DYNAMIC RECONFIGURATION AND LOAD DISTRIBUTION IN COMPONENT MIDDLEWARE

MAARTEN WEGDAM

The author of this thesis is employed at Bell Labs, the research arm of Lucent Technologies. The research presented in this thesis was done in cooperation with the Centre for Telematics and Information Technology (CTIT). Parts of the research were done in the context of two collaboration projects: AMIDST and EQUANET.

AMIDST is a Telematica Instituut project on next generation object-middleware platforms that can support Quality of Service (QoS).

EQUANET is an ICT-"Doorbraak" project sponsored by the Dutch Ministry of Economic Affairs on cost-effective realization of end-to-end QoS in multi-domain network environments.

This thesis is published as a part of both the CTIT PhD.-thesis series and the Telematica Instituut Fundamental Research Series.

CTIT (www.ctit.utwente.nl) is one of the key research institutes of the University of Twente. It conducts research on the design of complex ICT systems and their application in selected user domains.

Lucent Technologies (www.lucent.com) develops and delivers the systems, software and services for next-generation communications networks. Lucent focuses on high-growth areas such as broadband and mobile Internet infrastructures.

Telematica Instituut (www.telin.nl) is a partnership between the business community, research centers and government, to perform research in the field of telematics. The Dutch government supports Telematica Instituut under its 'leading technological institutes' scheme. Both CTIT and Lucent participate in the Telematica Instituut.

Dynamic Reconfiguration and Load Distribution in Component Middleware

Maarten Wegdam

Lucent Technologies
Bell Labs Innovations

Centre for
Telematics and
Information
Technology

Telematica
Instituut

Enschede, The Netherlands, 2003

CTIT PhD.-thesis series number 03-50
Telematica Instituut Fundamental Research Series, No. 009 (TI/FRS/009)

Cover Design: Studio Oude Vrielink, Losser and Jos Hendrix, Groningen
Cover Photo: Susan ten Buuren and Maarten Wegdam
Book Design: Lidwien van de Wijngaert and Henri ter Hofte
Printing: Universal Press, Veenendaal, The Netherlands

CTIT Ph.D.-thesis series, No. 03-50
ISSN 1381-3617; No. 03-50
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Telematica Instituut Fundamental Research Series, No. 009
ISSN 1388-1795; No. 009
Telematica Instituut
P.O. Box 589, 7500 AN Enschede, The Netherlands

ISBN 90-75176-36-8

DYNAMIC RECONFIGURATION AND LOAD DISTRIBUTION IN COMPONENT MIDDLEWARE

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 donderdag 26 juni 2003 om 16.45 uur

door
Maarten Wegdam
geboren op 12 februari 1973
te Almelo

Dit proefschrift is goedgekeurd door:

prof.dr.ir. L.J.M. Nieuwenhuis (promotor)

en

dr.ir. M.J. van Sinderen (assistent-promotor)

Abstract

The ability to control the Quality of Service (QoS) characteristics of distributed systems is essential for their success. The QoS characteristics that we consider in this thesis are performance characteristics (response time and throughput) and availability characteristics (uptime, mean-time-between-disruptions and mean-time-to-repair). Controlling QoS is a complex problem since it concerns all the functional layers we consider in this thesis, which are the application, middleware and resource layers. Controlling QoS is especially complex for large-scale systems such as telematics systems, due to heterogeneity and scalability issues.

QoS mechanisms control a certain QoS characteristic by allocating network and/or processing resources, or by adapting the application behavior. This thesis focuses on QoS mechanisms in the middleware layer.

Existing approaches to provide QoS are often static. Static approaches allocate resources either at startup or before startup of each instance of an application. This resource allocation remains fixed during the lifetime of the application instance. For static approaches this allocation has to be such that the highest resource usage that is expected during the lifetime of the application instance can be supported. This leads to a waste of resources since the actual resource usage of the application instance will vary during its lifetime. In addition, this also requires intimate knowledge about the resource needs of the application.

A dynamic approach varies the resource allocation based on the needs during the lifetime of the application instance. Contrary to the static QoS approach, a dynamic approach does not require calculations of the resource needs before startup of the application instance. In a dynamic approach the resource allocation is based on monitoring the achieved QoS during run-time, and adapting the resource allocation based on this. As a consequence, in a dynamic approach the QoS is not guaranteed, e.g., during run-time a dynamic QoS mechanism might discover a shortage of available processing resources. However, for a large class of distributed applications, such as telematics applications, the benefits of a more efficient resource usage outweighs the disadvantages of the inability to provide hard QoS guarantees. In this thesis, we only consider dynamic QoS mechanisms.

Large-scale distributed systems are often built using component-middleware technology (e.g., CORBA) because of the distribution transparency it offers. With distribution transparency, complexities related to the distribution are hidden from the application developers by embedding the distribution aspects in the middleware. We extend the distribution transparency concept by embedding the QoS mechanisms in the middleware layer, thereby hiding the complexities associated with QoS control from the application developer. In addition, this facilitates re-use of the QoS mechanisms.

Besides the transparency, a second aspect of our approach is that we make no assumptions with respect to the resource layer. Our middleware-layer QoS mechanisms do not rely on QoS functionality in the resource layer, such as, e.g., IntServ, DiffServ or real-time operating systems. The mechanisms we propose instead use the functionality of the middleware. For example, our QoS mechanisms can use the middleware to dynamically change the allocation of components to different nodes. The main challenges for the realization of these mechanisms are to preserve the correctness of the applications, and to minimize restrictions on application design.

We have applied our approach for dynamic QoS mechanisms in the middleware layer to two QoS mechanisms: a dynamic reconfiguration QoS mechanism, and a load distribution QoS mechanism. The dynamic reconfiguration QoS mechanism allows runtime upgrades of a component-middleware-based application. It can replace a component with a new version, migrate a component to another node, add a component or remove a component without taking the application instance as a whole offline. Since this prevents disruptions of the application, it increases the availability of the applications. An important characteristic of our dynamic reconfiguration mechanism is that it preserves correctness, viz., mutually consistent states, structural integrity and application invariants. Our dynamic reconfiguration mechanism drives a component to a reconfiguration-safe state in which there are no ongoing invocations by selectively queuing incoming invocations.

The load distribution QoS mechanism distributes the components over a set of nodes in such a way that the performance requirements are met. A load distribution strategy makes the distribution decisions, based on the available load information. The execution of the distribution decisions is done by distribution methods: initial placement, migration and replication of components. Since optimal load distribution depends on the specific characteristics of an application and of the environment the application is deployed in, we propose a framework-based solution in which it is possible to easily add new load distribution strategies and load information types. An important aspect of our load distribution mechanism is that it allows QoS differentiation, in addition to load sharing. We divide the available nodes into classes, and create, migrate and replicate components over these classes in such a way that the performance requirements are achieved.

For both QoS mechanisms we use message reflection techniques to achieve a clear separation between application code, the middleware code and the QoS mechanism code. The usage of reflection enhances the transparency and composability of our QoS mechanisms.

A CORBA-based prototype implements our Dynamic Reconfiguration Service and our Load Distribution Service. This prototype serves as a proof of concept for our approach and our QoS mechanisms. CORBA Portable Interceptors are used to implement the message reflection. Measurements with the prototype give insight into the performance aspects, and show that the overhead is acceptable for most applications.

The research presented in this thesis can be used to develop commercial QoS mechanisms that improve the QoS of component-middleware-based applications.

Acknowledgements

Although doing a PhD is mostly a solitary process, it is not something you can do on your own. Below I mention the people I am most indebted to for making it possible for me to do my PhD.

My promotor Bart Nieuwenhuis was there from start to finish to supervise me. He continued to supervise me when I left KPN Research to join Bell Labs Twente (Lucent), and also when he switched (principle) employer. I have a deep respect for his commitment to his part-time full professor job at the University of Twente, and his willingness to spend many evenings and weekends on reviewing draft versions of my thesis, and on contributing to papers we wrote together.

My assistant-promotor Marten van Sinderen got involved with my PhD when I had already done most of the research, but little of the writing of the thesis. His contribution to my PhD research proved to be essential. He reviewed more draft versions of my thesis than anyone else, and provided me with high-quality feedback on them.

Jeroen Schot is my supervisor at Bell Labs Twente, and I owe it mainly to him that Lucent supported my PhD research. He was also involved with the research itself, especially in the writing phase. His feedback taught me to be more strict in my reasoning.

Aart van Halteren was my 'promotiemaatje', certainly when we were both still at KPN Research, but also after that.

I supervised several master students who contributed to my PhD research. The message reflection research was done together with Dirk-Jaap Plas, who after his graduation became my colleague at Bell Labs Twente. The dynamic reconfiguration research was done together with João Paulo A. Almeida, who after his graduation became my room mate and fellow PhD student at the University of Twente. The load distribution research was done together with Erik Post.

Although not actually employed at the University of Twente (UT), I have been a visitor of Architecture of Distributed Systems group. Especially

during the last year of my PhD I have spent quite some time at the UT. This is a non-exhaustive list of people that made this a pleasant and productive environment: Giancarlo Guizzardi, Helen Kool, Remco van de Meent, Renata Silva Souza Guizzardi, Patrícia Dockhorn Costa, Diego Ríos, Valérie Gay, Remco Dijkman, Nikolay Diakov, Christian Tzolov, Dick Quartel, Luís Ferreira Pires, Marcos Salvador, Cléver R. Guareis de Farias, Marlous Weghorst, Aiko Pras (and, of course, João Paulo, Aart, Bart and Marten).

The people I have worked with have influenced my PhD research, and provided me with the context to do this research. I want to thank my colleagues at Bell Labs EMEA Twente (especially our secretary Sue Atkins and my 'cubicle mates' Bastien Peelen, Richa Malhotra and Dirk-Jaap) and Bell Labs EMEA Hilversum, and my fellow project members in the EURESCOM P715, EURESCOM P910, AMIDST and EQUANET projects.

Friends and family are very important in life in general, and possibly even more so while doing a PhD. Although I won't even try to be complete here, I want to mention some friends and family explicitly because I will remember them as extra supportive at one time or another (for as far as they are not already mentioned above): Mathijs Goldschmidt, Martijn Vloedbeld, Marijke Mayer, Femke Wegdam, Frank Oude Breuil, Merthe Oude Breuil, André Wegdam, Marga Wegdam-Fransen, Jorrit Wegdam, Michel Westenberg, Peter Leijdekkers, Frank Wegdam, Petra Demmenie, Edward Span, Harriët Span-Boersma, Cristian Hesselman, Miroslav Živković, and, especially, Susan ten Buuren.

Contents

Chapter 1

Introduction

This chapter presents the background, problem description, objectives, scope and approach for this thesis.

1.1 Background

Telematics

Telecommunication and computer technologies are converging [Janowiak03, P715], and telematics[1] forms the merger of the two. The telematics domain is the application domain we focus on in this thesis. We characterize *telematics* as the domain of distributed systems concerned with the support of the interactions between people or automated processes or both, by applying telecommunication and information technology [Visser00]. Telematics is also denoted by the term *Information and Communication Technology* (ICT). We elaborate on telematics systems by presenting an example.

Service platform

As a consequence of the unbundling of telecommunication networks and services, the operator is opening up its network to allow third-party service providers to provide end-user services that use the operator network [P715]. Examples of such end-user services are multimedia messaging, games and location-based services. This requires the operator to offer open interfaces that a third party can remotely access. These open interfaces are offered by *service platforms*. Standardization efforts in this area include TINA [Halteren99A, Sellin99], and more recently Parlay [Hellenthal01] and Open Service Access [Wegdam01B]. These standards enable third parties to provide end-user services by allowing these third parties access to the service platform that is located in the operator domain. The service platform communicates with the network and back-end systems in the operator domain, such as billing platforms, customer databases and Home

[1] The word “telematics” combines “telecommunication” and “informatics”.

Location Registers. Distribution is inherent for the realization of third-party end-user services; the end-user remotely accesses the end-user service: the end-user service remotely accesses the service platform, and the service platform remotely accesses the network and back-end systems. The communication networks that are used are both public networks (Internet) and private networks (local and wide area networks). There might be thousands if not millions of end-users and hundreds or thousands of third-party service providers, and these end-users and service providers can be involved in hundreds or thousands of parallel transactions. This makes service platforms large-scale distributed systems. Characteristics of a service platform, and of telematics systems in general, are as follows:

- the amount of end-users varies erratically in time;
- the end-users have soft real-time performance and availability requirements;
- there exists inherent heterogeneity in used networks, operating systems and computers due to the different parties that are involved. Because of this, standards play an important role to enable interoperability between the different parties.

Our focus on the telematics domain does not mean that the research described in this thesis cannot be applied to other domains as well, in fact we expect it can. But we will use the telematics domain to derive our requirements, and in this way ensure the applicability of our research to telematics systems.

Telematics systems are distributed systems for two reasons. The first is that the users of a telematics service are physically distributed, thus distribution of the service itself is inherent. The second reason is that telematics systems are often designed as distributed systems to be able to allocate more resources.

The quality of the telematics service is important because it is a way for a service provider to differentiate himself from his competitors. This Quality of Service (QoS) is often associated with the quality of the underlying communication networks. Network QoS is however only one aspect of the overall QoS as experienced by the end-user, and is not the focus for this thesis. This thesis focuses on the QoS that can be provided by component middleware, which is a generic software infrastructure that is used to support the telematics services.

We will first explain the role of component middleware, and then the relation between QoS and component middleware.

Component Middleware

Component middleware technologies facilitate the development of distributed applications by functionally bridging the gap between the application and the lower-level resources, and by enabling and simplifying integration of components developed by different parties [Schantz02]. We elaborate on component middleware by first addressing the component aspect, and then addressing the middleware aspect.

Component

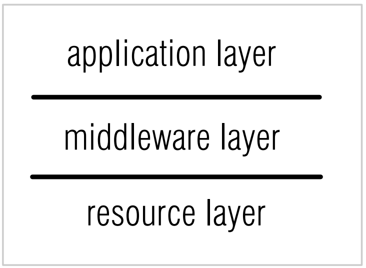
Distributed systems can be designed and implemented as a collection of collaborating and distributed components. A *component* encapsulates a piece of functionality, has a certain state, and offers services through one or more interfaces. A component can be developed independently from the rest of the system, and is subject to re-use and composition. A component has well-defined interfaces to interact with other components and its environment, and only interacts through these interfaces. The environment provides the component with the resources it needs to operate.

Middleware

Middleware is software that provides a supporting infrastructure for distributed applications that reduces costs by shifting common complexities of distributed systems from the application to the middleware [Raymond95]. Middleware hides the complexities related to distribution from the application developer. We refer to this property of hiding complexities as *distribution transparency*, or transparency for short.

Middleware is positioned as a software layer between the operating system (including the basic communication protocols) and the distributed applications that interact via the network [Geihs01A]. This results in a layered architecture consisting of an application layer, a middleware layer and a resource layer, see also *Figure 1-3*.

Figure 1-1 Three layered architecture



Component Middleware

Component middleware is a specific type of middleware that uses component concepts such as encapsulation and well-defined interfaces. Examples are the Common Object Request Broker Architecture [CORBA], and Enterprise JavaBeans [EJB]. *Figure 1-2* shows the resulting three-layered view on a distributed system. The application layer consists of a collection of interacting components. The resource layer consists of a collection of nodes that provide processing resources, connected by a network that provides network resources.

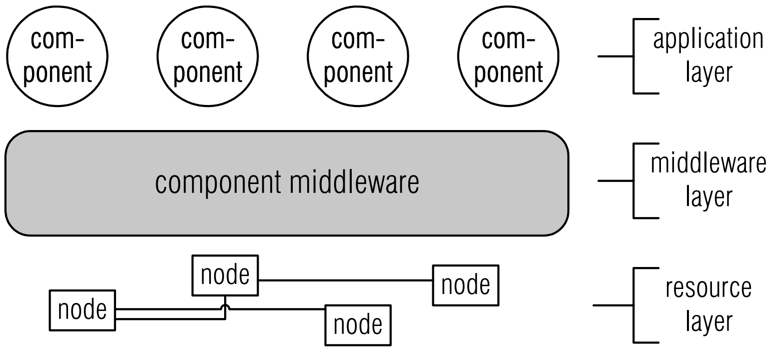


Figure 1-2 Layered view on a component-middleware-based distributed system

Quality of Service

Distributed systems offer a service, and this service has to fulfill certain Quality of Service (QoS) requirements. *Quality of Service* is defined as a set of qualities related to the collective behavior of one or more components [ISO-QoS]. This definition is quite broad, and in the context of this thesis we limit ourselves to performance and availability QoS characteristics. The *performance characteristics* are response time and throughput of interactions. The *availability characteristics* are the percentage of time that the system functions without disruptions, the mean-time-between-disruptions and the mean-time-to-repair. Disruptions can be caused be faults, e.g., network faults, but also by planned downtime of the system, e.g., caused by software upgrades.

Quality of Service

Performance and availability

The QoS that a component offers depends on the type and amount of resources that it has available. Resources can be *networking resources* such as bandwidth, or can be *processing resources* such as clock cycles. A QoS mechanism improves some QoS characteristic by, for example, reserving network resources, by prioritizing certain components or by prioritizing certain interactions between components.

Networking and processing resources

The research question we address is this thesis is how to provide QoS for component-middleware-based applications. This includes QoS differentiation, e.g., providing a different QoS to different users. We focus on two specific QoS mechanisms: dynamic reconfiguration and load distribution.

1.2 Problem Description

This section elaborates further on the problem of providing QoS for component-middleware-based applications, and describes some choices we make in our approach to solve this problem. These choices will be discussed in greater detail later in this thesis.

QoS is a Middleware Issue

A basic characteristic of the middleware layer is that it shields the components from the heterogeneity of the resource layer. Since the QoS as experienced by the components depends on the available resources, it is also the responsibility of the middleware layer to allocate the resources to the components in such a way that the application QoS requirements are met.

Best-effort QoS

Current component middleware technologies generally do not have a concept of QoS, and as a result offer only *best-effort QoS*. This means that there are no means to control the QoS, and the available resources are simply allocated to the components without considering specific QoS requirements. The achieved QoS depends solely on the available resources at a certain point in time. We need QoS mechanisms in the middleware layer to be able to control the allocation of resources to the components based on the QoS requirements.

Approaches to QoS Provisioning

We distinguish three approaches to QoS provisioning: over-dimensioning resources, static allocation of resources and dynamic allocation of resources.

Over-provisioning

The *over-provisioning approach* is based on providing abundant resources. Although straightforward and easy to implement, this does result in a waste of resources. In the *static approach* the required resources are calculated using some quantitative model, and these resources are reserved for the lifetime of the application instance. This approach is more efficient with resources than over-provisioning and it is possible to guarantee that QoS requirement are met. The static approach is, for example, used for real-time systems. The problem with this approach is that the quantitative model is based on detailed knowledge of the application, is dependent on the environment the application runs in and needs accurate predictions on the usage of the application. For telematics systems, and other large-scale systems with erratic usage, such a model-based approach is too static and difficult to implement. In the *dynamic approach* the resource allocation is adapted during runtime. This approach requires less detailed knowledge on the application and is better suited for telematics systems. We will adopt the dynamic approach in this thesis.

Static approach

Dynamic approach

Middleware-layer QoS Mechanisms

Middleware-layer QoS mechanisms can control a certain QoS characteristic by relying on resource-layer QoS mechanisms. An example of a resource-layer QoS mechanism is IntServ [IntServ94], which can be used to ensure that sufficient network resources are available to transport invocations between two components. Direct access by the application developer to the IntServ interface would violate the transparency that the middleware layer should offer. Middleware-layer QoS mechanism can be introduced to prevent this, as, for example, proposed in [Halteren03]. Middleware-layer QoS mechanisms that rely on resource-layer QoS mechanisms abstract the often low-level interface of resource-layer QoS mechanisms by offering an easier to use interface to the application developer. We refer to this category of mechanisms as *mapping mechanisms*, since they map higher-level QoS requirements to one or more lower-level resource-layer QoS mechanisms.

Mapping QoS mechanisms

A second category of QoS mechanisms uses the middleware functionality to improve the QoS. An example is a mechanism that uses the fact that middleware hides the precise location of a server component from the client to implement transparent migration or replication of components. This category of mechanisms would otherwise have to be implemented in the application layer, and as a consequence would be application specific and burden the application developer. We refer to this category of mechanisms as *middleware-layer-internal QoS mechanisms*.

Middleware-layer-internal QoS mechanisms

The mapping category of QoS mechanisms makes it easier to use resource-layer QoS mechanisms. The mapping category of QoS mechanisms excludes improvement of the QoS beyond what resource-layer QoS mechanisms provide. With middleware-layer-internal QoS mechanisms, however, it is possible to enhance the QoS beyond what is possible with resource-layer QoS mechanisms. For example, by replicating a component and dividing incoming requests over the different replicas we can enhance performance beyond what is possible with resource-layer QoS mechanisms alone. A second benefit of the middleware-layer-internal category is that since there is no dependency on resource-layer QoS mechanisms they allow more heterogeneity of the resources. These two categories are complementary, and they can be combined. The focus in this thesis is on middleware-layer-internal QoS mechanisms.

Dynamic Reconfiguration and Load Distribution

There are many possible middleware-layer-internal QoS mechanisms that improve availability or performance, such as, e.g., prioritizing certain invocations or replicating components. This thesis focuses on two mechanisms: a QoS mechanism that improves availability, and a QoS mechanism that improves performance:

Dynamic reconfiguration

– *Dynamic reconfiguration QoS mechanism* – This mechanism reconfigures or upgrades a running system without taking it off-line. This increases the availability QoS characteristics, viz., the percentage of time that the system functions without disruptions, the mean time between disruptions and the mean time to repair. Reconfiguration can be needed because the resources the system is using will no longer be available, or the behavior of the system needs to be adapted by replacing some of the components.

Load distribution

– *Load distribution QoS mechanism* – This mechanism improves the performance, viz., the response time and throughput. It does this by adapting the allocation of components to nodes by (i) migrating components to a different node, by (ii) instantiating components on the most suitable node or by (iii) replicating components over different nodes.

Both of these mechanisms fall into the category of QoS mechanisms that are internal to the middleware layer, i.e., they do not depend on resource-layer QoS mechanisms.

Reflection and Separation of the QoS Concern

Separation of concerns

Separation of concerns is a fundamental principle in software and system engineering to cope with the inherent complexity of designing a system. Applying the separation of concerns principle results in a system design that is split up into parts that each address a specific concern. These parts can, for example, be components or layers.

Formulated in these terms, the middleware layer handles the concerns that deal with the physical distribution of a system. The application layer is concerned with application specific concerns, i.e., the application logic. This application logic is separated in different components.

Middleware-layer QoS mechanisms

We want to develop *middleware-layer QoS mechanisms* and make QoS a middleware-layer concern. The problem is that these dynamic QoS mechanisms require monitoring and control functionality in the application layer [Molenkamp02], thereby potentially violating the separation of concerns principle.

Reflection

Reflection seems to be a promising technique to prevent this violation of separation of concerns, and to enhance the transparency for our QoS mechanisms. Reflection, or meta programming, is a technique that has its origin in programming languages and operating systems to introduce openness and flexibility [Yokote92]. It enhances the adaptability and composeability of a system. With *reflection* we mean the ability of a system to reason about itself, using some kind of self-representation. This reasoning is done at a meta-level where certain aspects of the system are represented or reified as meta-objects. Reflection enables both inspection and adaptation of systems at run time.

Message reflection

Message reflection is a specific type of reflection in which messages that are passed between different parts of the system are intercepted and reified. The adaptability and composeability properties of reflection fit very well with the separation of concerns requirements that form the basis of our approach. Message reflection especially fits very well with the type of distributed systems we are considering since these distributed systems consist of components that exchange messages, and we can implement message reflection with common-of-the-shelf middleware that is used to develop telematics systems.

1.3 Objectives and Scope

The main objective of the research presented in this thesis is

> *To develop middleware-layer QoS mechanisms that improve the availability and performance QoS characteristics of component-middleware-based applications.*

Essential characteristics of these QoS mechanisms are that they dynamically adapt the allocation of the available resources for a component and do not rely on resource-layer QoS mechanisms. The middleware-layer QoS mechanisms have to be as transparent as possible, which means that we aim for a maximum separation of concerns, and that the usage of these QoS mechanisms should not pose specific restrictions on the design of the application components. The responsibility of the application components is ideally limited to passing the QoS requirements to the middleware-layer QoS mechanisms, without requiring any knowledge on how these QoS mechanisms work.

The following objectives are considered to be part of the main objective:

- Propose an approach for the design of QoS mechanisms that dynamically adapt the resource allocation, that are transparent for the component developers, and that do not rely on resource-layer QoS mechanisms.
- Propose a new mechanism for dynamic reconfiguration. This mechanism makes it possible to adapt a running application. Adaptations that are supported include the replacement of a component with a newer version, and the migration of a component to another node. This improves the availability.
- Propose a new mechanism for load distribution. This mechanism distributes the components over a set of nodes in such a way that the performance requirements are met. This improves the performance.
- Investigate whether reflection can be used as a technique to achieve separation of concerns for middleware-layer QoS mechanisms, and if this is the case, how to do this.

Scope

We limit the scope of the research presented in this thesis to a single management domain. Hence, we do not consider federated QoS control.

In this thesis, we focus on operational interfaces, rather than streaming interfaces.

QoS is sometimes used to denote a wide range of characteristics, including, for example, image resolution, data integrity and security. We focus only on performance and availability.

We limit ourselves to QoS parameters that are meaningful at the middleware layer. The translation of high-level, end-user QoS parameters to QoS parameters on the middleware layer is the responsibility of the application developer, and is out of our scope.

We focus on the functional aspects of the QoS mechanisms. A quantitative analysis of QoS requirements, QoS mechanisms and the exact amount of required resources is out of our scope.

We limit ourselves to QoS mechanisms that can be used with common-off-the-shelf middleware, contrary to implementing our own middleware, or requiring source code changes to existing middleware.

Combining different QoS mechanisms can result in feature interaction issues, depending on the involved mechanisms. We do not consider this problem in the research presented in this thesis.

1.4 Approach and Structure

The approach to accomplish our objectives consists of the following steps:

1. Investigate state-of-the-art in component middleware technologies, relevant standards, component-based development and QoS provisioning for middleware-based applications (Chapter 2).
2. Define generalized component middleware concepts and terminology, including QoS for component-middleware-based applications (Chapter 2).
3. Identify requirements that QoS mechanisms have to fulfill in order to be suitable for large-scale telematics systems (Chapter 3).
4. Define our overall approach for the design of middleware-layer QoS mechanisms, which is based on separation of concerns, a dynamic QoS approach and no usage of resource-layer QoS mechanisms (Chapter 3).
5. Determine how to achieve transparency and separation of concerns using reflection techniques (Chapters 3, 4, 5).
6. Develop a QoS mechanism that uses dynamic reconfiguration to improve availability. This mechanism allows upgrades of component-middleware-based applications without taking them off-line (Chapter 4).
7. Develop a load distribution QoS mechanism that allows the distribution of load over a set of computers based on the QoS requirements, thus improving the performance (Chapter 5).

8. Implement a prototype to serve as a proof of concept for our approach for middleware-layer QoS mechanism, our dynamic reconfiguration mechanism and our load distribution mechanism (Chapter 6).

The structure of this thesis is depicted in *Figure 1-3*.

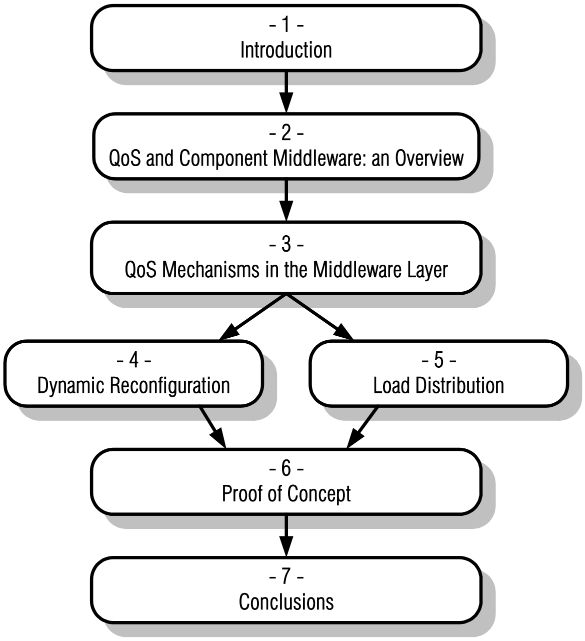


Figure 1-3 Structure of this thesis

Chapter 2

QoS and Component Middleware: an Overview

This chapter provides the context for this thesis. We do this by:

i. Presenting an overview of relevant standards and technologies in the area of component middleware.
ii. Presenting an overview of component-based systems, and how they can be designed.
iii. Defining QoS in the context of component-middleware-based systems.
iv. Defining our component middleware concepts.
v. Discussing related work in the area of QoS for middleware-based systems.

The standards and technologies in the area of component middleware we describe in this chapter are the Reference Model for Open Distributed Processing, Szyperski's work on component-based development, and state-of-the-art of current middleware technologies.

We used RM-ODP, Szyperski's work, the current middleware technologies as input to the definition of our generalized concepts and terminology for component middleware. We will use these generalized concepts and terminology throughout this thesis.

We describe relevant work in the area of QoS for component middleware, and relate it to our work. This does not include related work that is specific to one of the QoS mechanisms we developed, which we discuss when we discuss the specific mechanisms (in Chapters 4 and 5).

The overview of relevant standards and technologies in the area of component middleware in split up in three sections: Section 2.1 contains an overview of RM-ODP, Section 2.2 contains an overview of Szyperski's work and Section 2.3 describes the state-of-the-art of current component middleware technologies. Section 2.4 contains an overview of a design approach for component-based systems. Section 2.5 defines QoS for component-middleware-based systems. Section 2.6 contains our definition for component middleware concepts en terminology, and relates these to RM-ODP, Szyperski's work and current component-middleware technologies. Section 2.7 *describes related work in the area of QoS for middleware. Section 2.8 ends this chapter with concluding remarks.*

2.1 Reference Model for Open Distributed Processing

The *Reference Model for Open Distributed Processing (RM-ODP)* is a joint ISO and ITU-T standard that defines concepts and models to describe distributed systems. A lot of concepts that are used in the context of component middleware originate from RM-ODP. The standard is described in [RMODPPart1, RMODPPart2, RMODPPart3]. We will adopt some RM-ODP concepts in our work, as will be described later in this chapter. In this section, we give a short overview of RM-ODP.

2.1.1 Distribution Transparencies

Distribution transparency

An important concept of RM-ODP is distribution transparency. *Distribution transparency*, or transparency for short, is the property of hiding from a particular user the potential behavior of some parts of a distributed system. [RMODPPart2]. Users may for instance be end-users, application developers and function implementers. [RMODPPart1] and [Putman01] clarify what behavior is actually hidden by stating that transparency is the property of hiding from developers the details and the differences in mechanisms used to overcome problems caused by distribution. Examples of such problems are partial failure, heterogeneity and remoteness. The purpose of the distribution transparencies is to shift the complexities of distributed systems from applications programmers to the supporting infrastructure [Raymond95]. RM-ODP does not specify a complete set of distribution transparencies, but it does define a number of commonly required distribution transparencies, which are listed below:

Defined transparencies

- *access transparency* — hides the differences in data representation and procedure calling mechanism to enable interworking between heterogeneous computer systems
- *location transparency* — masks the use of physical addresses, including the distinction between local and remote
- *relocation transparency* — hides the relocation of an object and its interfaces from other objects and interfaces bound to it
- *migration transparency* — masks the relocation of an object from that object and the objects with which it interacts
- *persistence transparency* — masks the deactivation and reactivation of an object
- *failure transparency* — masks the failure and possible recovery of objects, to enhance fault tolerance

– *replication transparency* — hides the maintaining of consistency of a group of replica objects with a common interface
– *transaction transparency* — hides the coordination required to satisfy the transactional properties of operations

2.1.2 Viewpoints

Viewpoints

RM-ODP defines five so-called viewpoints. A *viewpoint* is a set of concepts, structures, and rules that are different for each viewpoint, providing a language for specifying distributed systems in that viewpoint. The five viewpoints are [Raymond95]:
– *Enterprise viewpoint* – for specifying purpose, scope and policies
– *Information viewpoint* – for specifying semantics of information and information processing
– *Computational viewpoint* – for specifying the functional decomposition
– *Engineering viewpoint* – for specifying the infrastructure required to support distribution
– *Technology viewpoint* – for specifying the choices of technology for the implementation

The different viewpoints focus on different aspects of the distributed system, and cannot be considered refinements. They are also not independent, and have to be consistent. RM-ODP specifies some concrete constraints on the relationship between computational and engineering viewpoint. We describe this relationship below when we describe the engineering viewpoint.

The enterprise language does not provide concepts we can use to specify QoS requirements or QoS mechanisms, and is therefore not relevant for our work. The information viewpoint is not relevant for us since it deals solely with the semantics and structure of the information that is exchanged, and not with QoS. The technology viewpoints is also not relevant for our work since it is too technology specific, and because it is not a very well defined or complete part of the RM-ODP standard. This leaves the computational and engineering viewpoints, which are relevant for our work. We describe the computational and engineering viewpoints more elaborately below, and explain their relevance.

Computational viewpoint

The *computational viewpoint* is object-based, that is:
– An object encapsulates data and processing (i.e., behavior)
– An object offers (possible multiple) interfaces for interaction with other objects

The distribution of the objects, and the distribution aspects are transparent in the computational viewpoint. Most objects in a computational specification describe application functionality, and these objects are linked by bindings through which interactions occur [Raymond95, Leijdekkers97]. There are three forms of interaction between objects: stream-oriented, signal-oriented and operational. The computational viewpoint is focused on the functionality of the system, i.e., the division in objects, their interfaces and the behavior. Although distribution aspects are not described in the computational viewpoint, the requirements for the distribution are part of the computational viewpoint. These include QoS requirements, which makes this viewpoint relevant for our work.

Engineering viewpoint

The *engineering viewpoint* is concerned with distribution aspects and not with the application semantics except to get the requirements for the distribution aspects. It defines the concepts needed to describe the distribution infrastructure that is required to support the distribution transparencies. The engineering viewpoint is not a refinement of the computational viewpoint as a whole, but only of the interactions. The fundamental entities in the engineering viewpoint are channels and objects. A channel provides the communication mechanism and contains or controls the transparency functions required by the basic engineering objects, as specified in the computational specification. Objects in the engineering viewpoint can be divided into two categories—basic engineering objects and infrastructure objects. Basic engineering objects correspond to objects in the computational specification. This correspondence defines the relationship between the computational and engineering viewpoints.

The engineering viewpoint prescribes the structure of an ODP system. The basic *units of structure* are:

Units of structure: node, cluster, capsule, nucleus

- *Node* – An independently managed computer system.
- *Cluster* – A set of related basic engineering objects that will always be co-located.
- *Capsule* – A set of clusters, a cluster manager for each cluster, a capsule manager, and the parts of the channels which connect to their interfaces. A typical example is a process in the operating system sense.
- *Nucleus* object – Controls a node, i.e., an (extended) operating system.

The engineering viewpoint deals with the mechanisms to implement the interactions, and with the resources. This makes the engineering viewpoint relevant for our QoS work.

In Section 2.6.2 we discuss which of the RM-ODP concepts we re-use in this thesis.

2.2 Component-Based Development

Component-based development is a design methodology that focuses on third-party composition of software. Component-based development and the technology to support this are becoming more and more popular in industry and academia.

There is a lot of literature on component-based development, but we consider Szyperski's work on components [Szyperski98] to be representative for component-based development. The component definitions, concepts and ideas presented in this section are therefore based on Szyperski's work.

The main idea behind component-based development is to allow better re-use of code than older techniques such as macros, libraries and object-orientation. Concrete manifestations of a component can vary from procedures, modules, classes, libraries to entire applications, as long as they are binary and independent. Szyperski defines a software *component* as:

Szyperski's component definition

> *"an unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Context dependencies

Explicit *context dependencies* specify what the deployment environment needs to provide to allow the component to function, including the required interfaces that other components need to offer.

Contract and interface

Ideally, the *contract* between the client of a component and the component specifies both functional and non-functional aspects. The functional part of the contract is typically an *interface* annotated with pre- and postconditions, and possibly invariants. This however does not cover the non-functional aspects, such as performance and resource requirements. Szyperski considers the question on how to include non-functional aspects in a contract as ongoing and future research.

Unit of deployment

Third-party composition and no persistent state

A component is a *unit of deployment*, meaning that the software component is what is actually deployed. A component needs to be well separated from its environment and from other components. A component is a unit of *third-party composition*, and has *no persistent state*.

Implementation inheritance

Interface inheritance

Components do not support *implementation inheritance*, i.e., it is not possible to make a subclass of a component that inherits its behavior. The reason behind this is that this would create a strong dependency of the subclass on the implementation of the component it inherits from. For example, implementation inheritance would have the undesirable requirement that both components have to be implemented in the same programming language, and this programming language would have to be object oriented. Components do support *interface inheritance*, i.e., the interface of the superclass is inherited without inheriting the behavior or implementation.

Objects versus components

Components are different from objects in the object oriented programming sense. In short, the characteristics of objects are encapsulation of state and behavior, polymorphism, and inheritance. According to Szyperski, a main difference is that objects do not support the notions of independent deployment and third-party composition. Using components however, does not exclude use of objects. A typical example of a component is a class, or a set of classes. Objects then form part of component instances, and can be instantiated as needed. A component however, does not have to be implemented using objects, it can just as well be implemented using other programming techniques.

In Section 2.6.3 we discuss which of Szyperski's concepts we adopt in this thesis.

2.3 State-of-the-Art in Component Middleware

In this section, we describe the most popular component middleware technologies and standards, which are:
- CORBA
- Enterprise JavaBeans
- CORBA Component Model
- SOAP and WSDL
- DCOM and COM+

2.3.1 CORBA

Common Object Request Broker Architecture (CORBA) is a middleware technology that is standardized by the Object Management Group (OMG) [CORBA]. There are different generations of CORBA specifications. In this section, we consider only the CORBA 2.x generation.

In CORBA, application objects have interfaces that objects running on other nodes can use. All interfaces are of the operational type, i.e., they support remote method invocations. Every object is identified by a unique object reference, called the Inter Object Reference (IOR).

The CORBA standard is implemented by different vendors. CORBA is operating system independent, and there are implementations available on different operating systems, including the family of Microsoft Windows operating systems, and different UNIX versions. To guarantee interoperability between CORBA implementations of different vendors, OMG standardized the General Inter-ORB Protocol (GIOP). This protocol can be mapped on different network protocols. The mapping on TCP/IP is called Internet Inter-ORB Protocol (IIOP) and is part of the CORBA standard.

The core of the CORBA standard is the Object Request Broker (ORB). The ORB provides the basic mechanisms by which objects can make requests and receive responses. The interface between an application object and the ORB is part of the standard.

All interfaces are defined in the Interface Definition Language (IDL). Interfaces can have an inheritance relationship to other interfaces. Each interface consists of a set of operations. For each operation the operation name, the parameters, the result value and possible exceptions are specified. IDL is implementation language independent. The specification contains mappings from IDL to different implementation languages, including Java and C++. An IDL compiler uses the IDL to generate stubs and skeletons in a specific implementation language for marshalling of the operations.

OMG specifies several services that provide some common functionality. Examples are the naming service, the notification service, the security service and the transaction service.

OMG also standardized Portable Interceptors [Wegdam00A], which provide an ORB independent way to intercept a request or reply, and inspect the content. There are also limited possibilities to alter a request or reply.

For the set of implementation languages that are part of the standard, CORBA ensures not only interoperability, but also portability of application objects. Portability here means that without changing an application object it can use a CORBA implementation from vendor X or from vendor Y. This is possible because both the interface between an application object and the ORB, and the language mappings are part of the standard.

2.3.2 Enterprise JavaBeans

The *Enterprise JavaBeans (EJB)* [EJB] specification is part of Sun's Java 2 Platform Enterprise Edition (J2EE). J2EE is a set of standards to develop multitier enterprise applications. EJB is a standard architecture for component-based distributed computing. EJB instances are server-side components, i.e., EJB instances are meant to be components running in the backend, contrary to the first tier that runs on the end-user system. EJB is not suitable for thin clients such as web browsers and mobile phones.

An important concept in the EJB specification is that of EJB container. A *container* provides the execution environment for *EJB instances*. The EJB instances and container interact through standardized local interfaces. An *EJB server* can contain multiple containers. Each container can accommodate multiple EJB classes. An *EJB class* corresponds to the type definition of an

EJB, and contains the implementation. Each EJB class has a *home interface* that can be regarded as a set of class methods that define factory methods by which EJB instances, or EJB objects, for that class can be created, deleted or located. A *class method* is comparable to static methods in C++ or Java. Multiple EJB instances of the same class can reside in the container. Each EJB instance has a *component interface* that lists the methods that local or remote clients can invoke.

There are restrictions on what an EJB instance is allowed to do. These restrictions make it possible for the container to manage the resources EJB instances use. An important restriction is that an EJB is *passive*. This means that EJBs are not allowed to perform any threading operations and they only respond to incoming invocations. Also the container serializes any incoming invocations, and only one thread is simultaneously active within one EJB instance (single thread model). The only exception is re-entrance, which is allowed in some cases.

The EJB specification distinguishes a separate deployer role in the development and deployment process. The deployer is responsible for deployment of EJB classes into a specific operational environment. This includes resolving any external dependencies declared by the developer of the EJB, e.g., other EJB instances it depends on, or database connections it requires. The benefit of this is that the EJB class itself will be more independent of the environment it will be deployed in.

J2EE specifies several *common services*. These include a directory service and a transaction service. The transaction service is compatible with the OMG Transaction Service [CORBA].

There are different categories of EJB classes. A developer can choose to implement an EJB class as one of three categories: session bean, entity bean and message-driven bean.

- *Session bean* – a session bean executes on behalf of a single client, and does not have a persistent state or identity. There are two sub-categories:
  - *stateless session bean* – no state between invocations
  - *stateful session bean* – conversational state on a per-client, per-session basis. Thus a different instance of a stateful session bean implementation class is used for each client, and its state is maintained between method invocations until the session is terminated.
- *Entity bean* – has persistent state. Entity beans can be shared between clients. Entity beans provide an object view of data in a database. Typically, an entity bean represents a single row of a query into a relational database.
- *Message-driven bean* – a more recent addition to the standard (since version 2.0). Communication is not based on remote invocations, but

based on exchanging messages. This is comparable to the OMG Notification Service. A message-driven bean is stateless, and does not have a persistent identity.

The container is responsible for the management of the lifecycle of an EJB. Stateful beans can be activated and passivated. *Activated* means that they are loaded from secondary storage, and the resources are allocated. *Passivate* is the opposite, the EJB is stored in secondary storage and the resources are released.

Session and entity beans use *Java Remote Method Invocations* (RMI) [RMI] to communicate with clients, or to interact with other EJB instances. RMI is a remote procedure call mechanism similar to CORBA, but meant for a Java-only environment. A major difference with CORBA is that RMI allows pass-by-value for objects. RMI uses the CORBA protocol IIOP to interoperate between EJB servers of different vendors, and to interoperate with CORBA-based systems.

2.3.3 CORBA Component Model

The *CORBA Component Model (CCM)* [CCM] is part of CORBA 3.0. The CMM extends and builds upon CORBA 2.x, and is standardized by OMG.

A *CORBA component* is a basic meta-type in CORBA. The component meta-type is an extension and specialization of the CORBA Object meta-type. A CORBA component has a component type that is specified in IDL. A CORBA component is denoted by a *component reference*, which is represented by an object reference. A component interacts with its environment through ports. There are different kinds of ports:

Ports

– *Facets* – named interfaces offered by the component. A component can offer more than one facet. The lifecycle of the facets is bound to the lifecycle of its component.
– *Receptacles* – the ability to accept object references upon which the component may invoke operations. This can thus indicate a dependency of the component on other components.
– *Event sources and event sinks* – sources and sinks supported by the component for the publish/subscribe event model that is part of CCM.
– *Attributes* – named values exposed through accessor and mutator operations, and primarily intended to be used for configuration.

Component reference

A component is identified by its *component reference*. The component reference supports the so-called *equivalent interface*, which can be considered as a special kind of facet. All ports can be reached through this equivalent interface.

There are two levels of components: *basic components* and *extended components*. Basic components are very similar to EJB (version 1.1), and according to the specification essentially provide a simple mechanism to "componentize" a regular CORBA object. Basic components are limited to offering one facet (the equivalence interface) and possibly attributes, but no other ports. Extended components can offer all kinds of ports. *Figure 2-1* depicts an extended CORBA component with the equivalent interfaces and two facets.

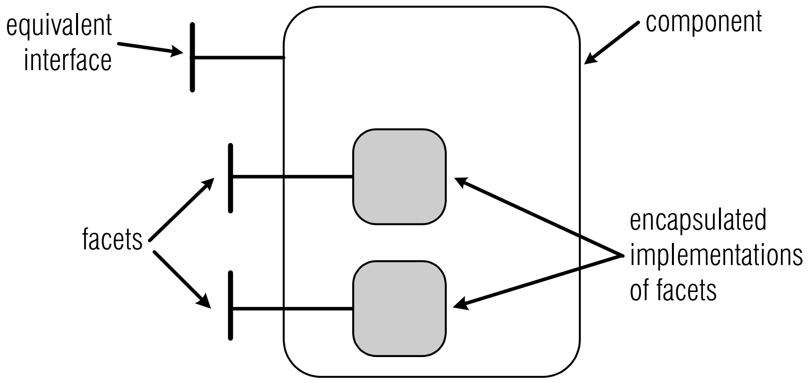


Figure 2-1 A CORBA Component with facets

Component home

A *component home* is a manager for all component instances of a certain type within the scope of a container. The concept of container is explained below. The component home controls the lifecycle of the component instances. This means that all components are created with a factory pattern through the component home. Components may also be associated with a primary key. These primary keys are exposed to the clients, and are maintained by the component home. A client can use this primary key to identify or get a reference to a specific component instance.

A *container* is the execution environment in which a component instance and a component home live. It provides via *local interfaces* a standard set of *services* to the component. These standard services are transactions, security, persistence and events. There are two types of containers: the session type and the entity type. Components that run in an entity container have a persistent reference, components that run in a session container have a transient reference. The specification mandates that a CORBA Container can also host Enterprise JavaBeans (EJB) (with EJB 1.1 version). An EJB is then exposed as if it was a CORBA Component.

Figure 2-2 depicts a CORBA Component with a container and the rest of the CCM entities.

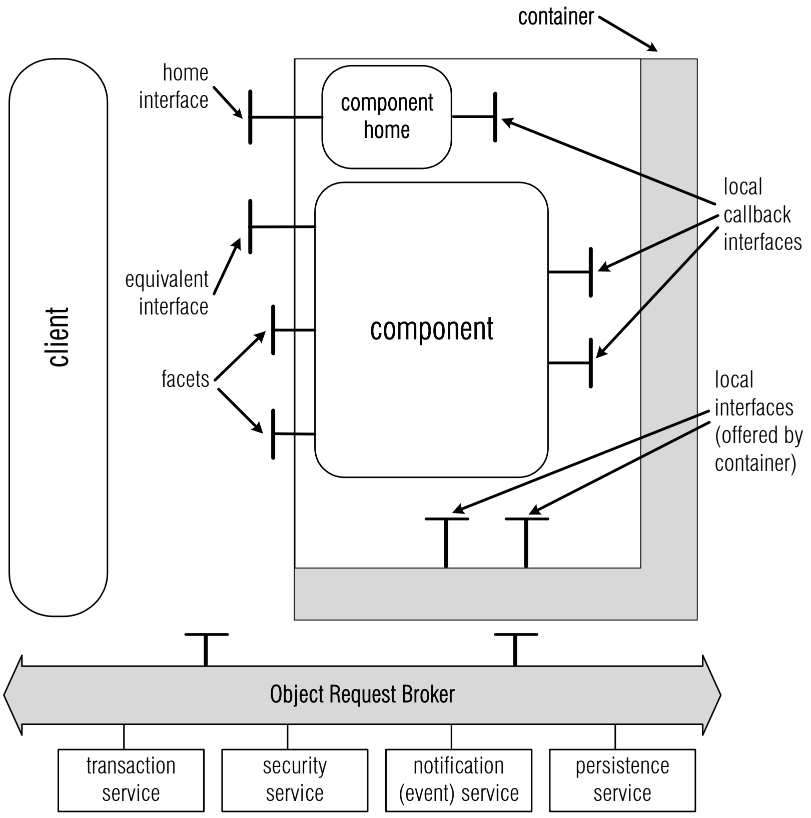


Figure 2-2 The CORBA Component Model

There are four different component categories:

– *Service component* – only state within the processing of one invocation and no identity. The lifecycle is equal to a single invocation. This is similar to the stateless EJB session bean.
– *Session component* – only state within a session (no persistent state). It has an identity, but this identity is not persistent. This is similar to the stateful EJB session bean.
– *Entity component* – persistent state and persistent identity. The identity is externally visible, thus the finder pattern is used, and an entity component has a key. This is similar to the EJB entity bean.
– *Process component* – persistent state and persistent identity. Contrary to the entity component, the identity is not exposed to the client, i.e., no finder pattern or keys are defined. There is no corresponding EJB bean.

The CCM can be considered a generalization of EJB [Szyperski99]. It borrowed a lot of concepts and ideas from EJB, and interoperability between the two is part of the CCM standard. The most notable feature CCM adds to EJB is language independence. Unfortunately, there are no complete and mature implementations of the CCM available at the moment of writing.

2.3.4 SOAP and WSDL

Simple Object Access Protocol (SOAP) [SOAP] is a W3C standard that specifies a mechanism to perform remote method invocation by exchanging XML [XML] messages between a web client and a web service. SOAP messages are commonly exchanged using HTTP as a transport protocol, but other protocols (e.g., SMTP) are also possible.

Web Services Description Language (WSDL) [WSDL] is also a W3C standard. It is an XML format describing the web service operations, protocol bindings, and protocol message formats. A WSDL definition usually also contains information about the service endpoint, i.e., the address where the service is deployed.

The combination of SOAP to do remote method invocations and WSDL to define an interface, guarantees interoperability between software entities.

SOAP and WSDL specify interfaces and a protocol, comparable to the combination of GIOP, IDL, IOR and the Common Data Representation in CORBA. CORBA, and the other component middleware technologies discussed in this section, however, specify much more. For a discussion on this, see [Lagerberg02]. We summarize the main differences here:

– SOAP/WSDL do not specify assumption on how software entities are implemented, i.e., there is no component model. For example, the WSDL/SOAP specifications do not consider concepts as unit of deployment and independency.
– SOAP/WSDL do not specify the APIs that the execution platform on which the software entities run should support. SOAP/WSDL thus provide no portability and every platform of every vendor will have different APIs between the platform and the software entities.
– SOAP/WSDL do not specify language mappings, marshalling functionality and common services.

Different vendors of platforms that use SOAP and WSDL do offer or plan additional functionality or assume a particular type of component model. Due to the proprietary nature of these platforms and the immaturity of WSDL, SOAP and related technologies, it is not possible to make proper generalizations about these platforms and the component functionality they (might) support.

2.3.5 DCOM and COM+

DCOM [DCOM] is an object middleware technology from Microsoft that borrowed a lot of concepts of CORBA [Emmerich02]. DCOM is programming language independent, but it depends on the Microsoft Windows family of operating systems. It is closely integrated with MicroSoft development tools. *COM+* [COM+, Eddon99] is a component middleware technology that uses DCOM. Microsoft more recently released *.NET*, which is Microsoft answer to Sun's J2EE. .Net re-uses parts of DCOM/COM+ [Bakken01]. A major change in .NET compared to DCOM/COM+ is that .NET is very Web Services (SOAP) oriented. The successor of DCOM/COM+ in .Net is called *.Net Remoting* [NetRemoting]. .Net Remoting is a component middleware technology with many similarities with DCOM, Enterprise JavaBeans and CORBA. It can interoperate with existing DCOM and COM+ applications.

2.4 Component-Based Systems and their Design

In this section, we give an overview of component-based distributed systems, and how they can be designed. We assume a *top-down design* approach here, thus start with a high level view of distributed systems, and refine this in several steps.

2.4.1 Stepwise Refinement

At the highest abstraction level, we distinguish the distributed system itself, and its interactions with the environment. This is depicted in *Figure 2-3*. We do not make any assumption on the type of environment and the type of interactions between the distributed system and the environment. The environment could for example be a human user, but also some automated process. The nature of the interactions between the distributed system and the environment will depend on the type of environment, e.g., graphical user interfaces, or analog signals.

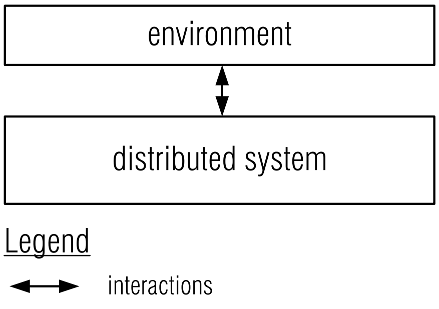


Figure 2-3 Model of a distributed system as a black-box

Stepwise refinement

A well-known methodology to design distributed systems, or systems in general, is *stepwise refinement* [IEEE610]. Refinement uses the *divide-and-*

conquer engineering principle in which a system is divided in several subsystems, and the *abstraction principle* in which irrelevant information is suppressed [RMODPPart2]. In iterative steps the distributed system is divided in subsystems that interact, as depicted in *Figure 2-4*. How this subdivision is done is part of the creative design process. Criteria for this division can be functional (functional decomposition) [IEEE610], but other criteria are also possible. For example, the fact that a system is geographically distributed can influence the decision of the designer on how to divide a system in subsystems. Or, if there is existing functionality that can be re-used, a designer might want to divide a system in such a way that reuse is possible. A third example is to meet certain QoS requirements. A concrete example of this is that some functionality must have a higher availability than the rest the system. Modeling this functionality as a separate subsystem that can be replicated can then ensure this higher availability.

For a refinement step to be correct, the combined external behavior of the subsystems has to provide the same external behavior as the original system. The process of stepwise refinement will often have cycles, because at a lower level of refinement you may discover that you are not able to sufficiently address certain concerns because of decisions taken at a higher level [Emmerich02].

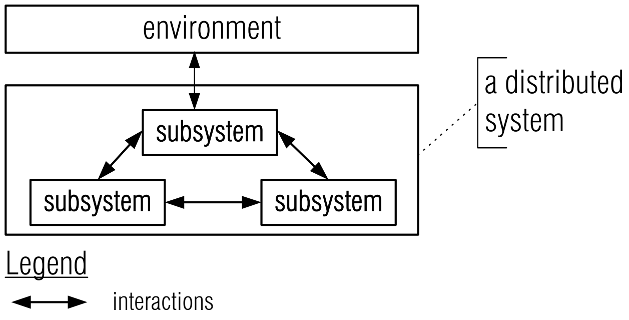


Figure 2-4 Example of refinement step

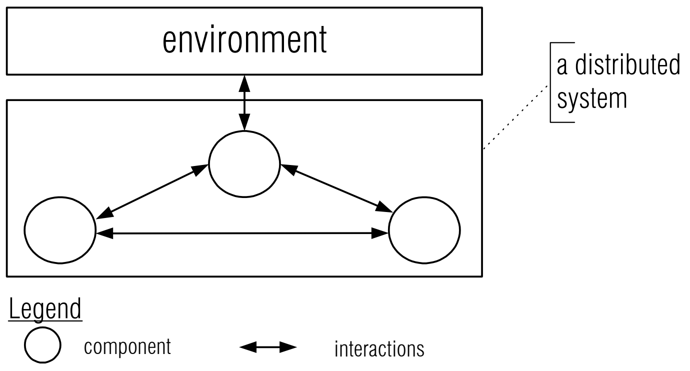
The stepwise refinement stops when the granularity of the subsystems is such that each subsystem can be implemented using the software artifacts that the chosen implementation technology offers, or if existing subsystems can be used. At this level, we refer to the subsystems as components. Thus, at the lowest level of stepwise refinement, we model a distributed system as a set of interacting components. The complete design methodology is not important for our thesis, and we will not cover it further.

Component

A *component* encapsulates its behavior and state, and has one or more interfaces. Components interact with each other via their interfaces, and only via their interfaces. There is a strong parallel here to object orientation, and a component is similar to an object. We prefer to use the term component to avoid confusion with object as a programming language artifact, and because components as we define them are similar to components as they exists in recent middleware technologies (see Section 2.3). We elaborate on the definition of component in Section 2.6

Figure 2-5 depicts an example of a distributed system consisting of three interacting components. We will refer to such a model as the computational viewpoint of a distributed system (see Section 2.1). The computational viewpoint does not model the behavior of the distributed system as such, it only describes how it is structured. Since in this thesis we focus on the QoS of components and the QoS of interactions between components, and not on the semantics of the system, we do not require a model that describes the complete behavior of the distributed system.

Figure 2-5 An example computational model of a distributed system



QoS of component interactions

The QoS requirements for the interactions between the distributed system and the environment determine the QoS requirements for the components and component interactions. We allow these interactions to be of any type, and thus cannot make any assumptions on them for the generic case. The issue of how to translate the QoS requirements of the interaction with the environment to QoS requirements for the components and the component interactions is therefore an application designer issue. We cannot provide generic support for it, and it is not a middleware issue. We therefore consider the QoS characteristics of the interactions between the distributed system and the environment, and the translation into QoS characteristics of components and component interactions, outside the scope of this thesis.

2.4.2 Remote Method Invocation

Servers and clients

In the common middleware technologies we consider in this thesis an interaction is always limited to two components. One component offers certain functionality, i.e., a service, to another component. The component that offers the service has a *server* role, the component using the service has a *client* role. A component can be involved in numerous interactions during its lifetime and can play both the client and the server role. Interacting components can be distributed over different nodes, but do not have to be. We abstract from this, we only indicate the potential to be distributed. In this thesis we limit ourselves to *operational interfaces* and do not consider streaming and signal interactions. An instance of an operational interaction consists of the sequence of activities starting when a client component issues a request for a server component, followed by the processing of this request by the server component and ending when a reply to the request is delivered to the client component. In the literature this is often referred as a *remote procedure call*, or *remote method invocation* [Bakken01]. A set of related remote method invocations between a specific client and server component is called a *session*. This session is implicitly created by the middleware with the first invocation.

Operational interfaces

Remote method invocation

Session

Figure 2-6.depicts two alternative representations of a single instance of interaction between a component in the server role and a component in the client role. In *Figure 2-6a* we explicitly show the request and the reply by using one-sided arrows. In *Figure 2-6b* we indicate the server component by drawing the interface on the server, and imply the request and reply.

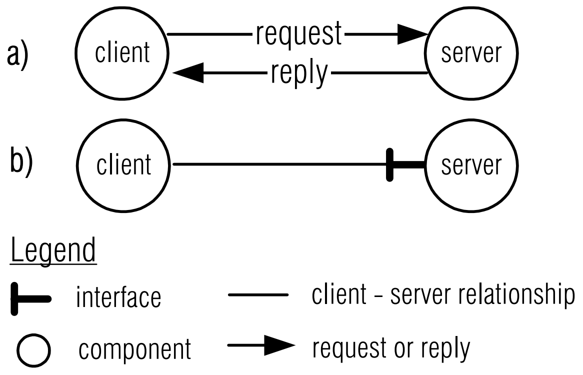


Figure 2-6 Two alternative ways to depict a client and server component

Remote invocations between components are enabled by a *distribution infrastructure*. The distribution infrastructure offers distribution transparencies to the components. The distribution infrastructure contains network and processing resources, and middleware. In Section 2.6 we will detail the distribution infrastructure further, for now we will consider this as a black box.

Distribution infrastructure

Every remote invocation between two components involves two local invocations between the components and the distribution infrastructure. This is depicted in *Figure 2-7*

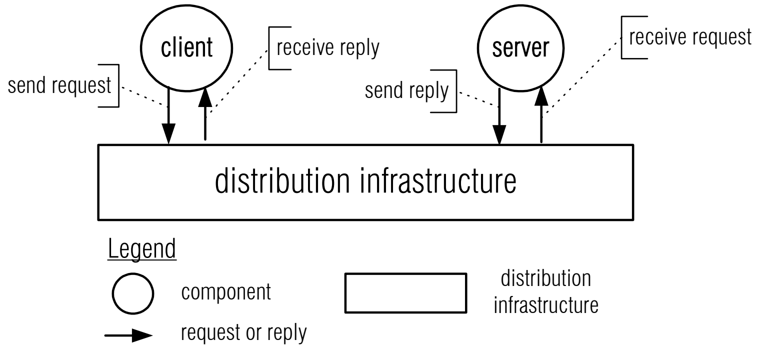


Figure 2-7 Instance of interaction supported by a distribution infrastructure

2.5 Quality of Service

This thesis is about QoS in component-middleware-based systems. We do not focus on the QoS as perceived by a human end-user, or the QoS of the interactions between the system and the environment, but instead consider the QoS as it is provided to a component when interacting with other components. The QoS as perceived by the environment, e.g., by a human user through some graphical user interface, depends on the QoS as provided by the different components. How the QoS as perceived by the environment is translated into QoS as provided to the different components is application specific, and we consider this the responsibility of the application developer.

Quality of Service

RM-ODP defines *Quality of Service* as a set of quality requirements on the collective behavior of one or more objects [RMODPPart2]. It does not define what the quality requirements could be, it only gives some examples, such as the rate of information transfer, the latency, the probability of a communication being disrupted, the probability of a system failure and the probability of a storage failure. The RM-ODP definition of QoS is also used in the *ISO QoS Framework* [ISO-QoS], which does list possible quality requirements. We will base our definitions of quality requirements on the definitions that the ISO QoS Framework provides.

ISO QoS Framework

The ISO QoS Framework describes QoS characteristics that can be used in communications and processing. The following categories are distinguished:

– *time-related* characteristics
– *coherence* characteristics
– *capacity-related* characteristics
– *integrity-related* characteristics

– *safety-related* characteristics
– *security-related* characteristics
– *reliability-related* characteristics
– other characteristics

We do not consider all these QoS characteristics in this thesis, but limit ourselves to:
– time delay (time-related characteristics)
– throughput (capacity-related characteristics)
– availability, reliability, and maintainability (reliability-related characteristics)

All other QoS characteristics defined by the ISO QoS Framework, and any other QoS characteristics not captured in the ISO QoS Framework, are outside the scope of this thesis.

We will define these characteristics, based on the definitions of the ISO QoS Framework but specialized for component-middleware-based systems.

Response time and throughput

The *response time* is the time delay from the perspective of the client between sending the request, and receiving the reply. The *throughput* is the amount of invocations per time interval. This is thus the rate of requests/replies, and can also be denoted with the more generic term capacity.

Failure, fault and error

Before defining the reliability-related characteristics, we first need to define failure, fault and error. A *failure* is a violation of a contract [Nieuwenh91, RMODPPart2, Avizienis00], i.e., a deviation of the correct behavior. A failure can be caused by an error, but an error does not necessarily lead to a failure. Corrective actions or internal redundancy may prevent an error to cause a failure. An *error* is a manifestation of a *fault*. Faults can be dormant or active. A fault is active when it produces errors.

Availability

Using this definition of failure, we define *availability* as the percentage of time that the application is functioning without failures.

Reliability

Reliability is the time between disruptions of the service, i.e., the mean time between failures. Disruptions can be caused by some fault in the hardware or software, but also by system maintenance, e.g., when upgrading some part of the application.

Maintainability

The *maintainability* is the duration of any continuous period for which the service is disrupted, i.e., the mean time to repair.

Performance and availability QoS categories

We group these five QoS characteristics into two categories: performance and availability. The *performance* category consists of the response time and the throughput characteristics. The *availability* category consists of the availability, reliability and maintainability QoS characteristics.

Statistical QoS characteristics

Statistical QoS characteristics are derived from a specific QoS characteristic. The ISO QoS framework defines some commonly used statistical QoS requirements: maximum, minimum, range, average, variance, standard deviation, n-percentile and statistical moments [ISO-QoS]. For example, for delay it is possible to state that the maximum response time should be less than 150 ms, the average 50 ms, and in 90% of the cases it should be less than 75 ms.

Offered, agreed and achieved QoS

The QoS characteristics are not only used to describe the required QoS, but also to describe the *offered* QoS, the *agreed* QoS and the *achieved* QoS. In a typical scenario the client will have some required QoS, which is matched with the offered QoS by one or more servers. This process is called the QoS negotiation, which will besides client and server also involve the supported infrastructure. The QoS negotiation results in an agreed QoS. The achieved QoS is the QoS that the client actually receives.

QoS mechanism

A *QoS mechanism* is a specific mechanism that may use protocol elements, QoS parameters or QoS context, possibly in conjunction with other QoS mechanisms, in order to support establishment, monitoring, maintenance, control, or enquiry of QoS [ISO-QoS]. A QoS mechanism enhances one or more QoS characteristics.

QoS as perceived by the client component

The *QoS as perceived by the client component* is a combination of the QoS of the transportation of the request and reply by the distribution infrastructure, and the QoS of the server component to process the invocation. *Figure 2-8* shows a message sequence diagram that illustrates this for the response time characteristic for a simple invocation. Between time t1 and time t2 the requests is transported from the client to the server, between t2 and t3 the server processes the invocation, and between t3 and t4 the reply is sent to the client. *Figure 2-9* shows the response time as provided to the client C1 in case of a more complex interaction in which C1 is a client of C2, and C2 interacts with C3 as part of processing the interaction with C1. This is called a *nested invocation*. We illustrate with this example that in case of a nested invocation the QoS depends on all involved components and interactions.

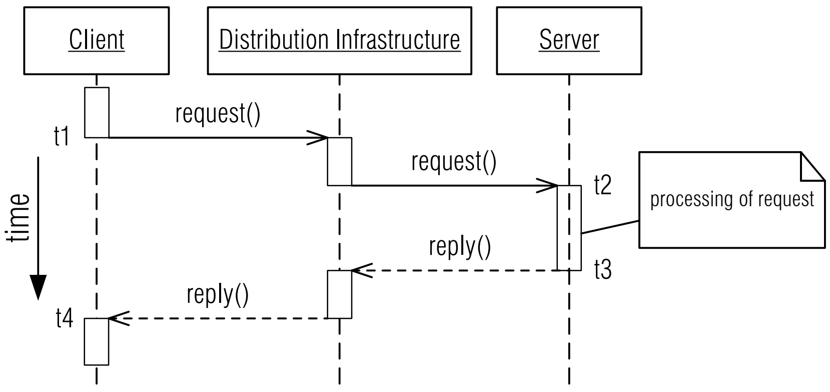
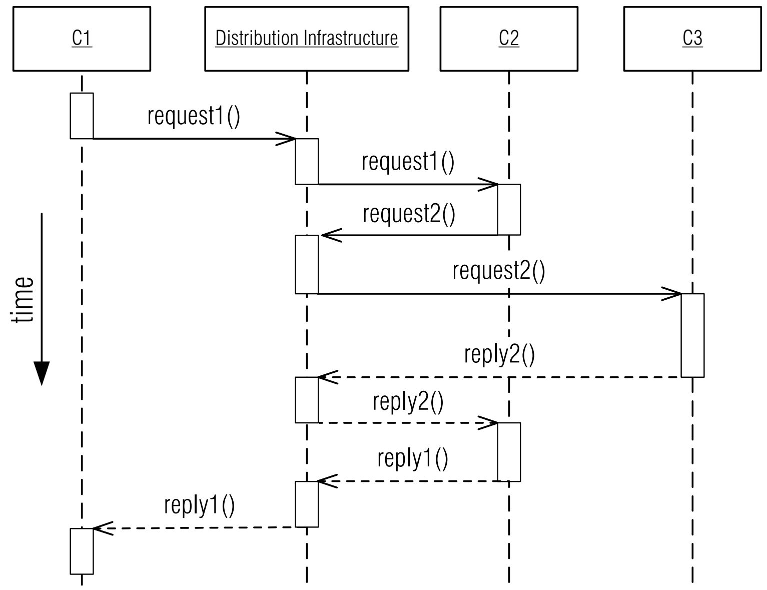


Figure 2-8 Response time for a simple invocation

Figure 2-9 Response time for a nested invocation



2.6 Component Middleware Concepts

In this section, we present our terminology and concepts to discuss component middleware and component-middleware-based applications. Our concepts and terminology are a generalization of concepts found in existing component middleware technologies as described in Section 2.3, and we selectively use concepts defined by RM-ODP (see Section 2.1) and Szyperski (see Section 2.2). In the end of this section we compare our concepts and terminology with RM-ODP, Szyperski's definitions and with current middleware technologies.

Layering

Resource layer

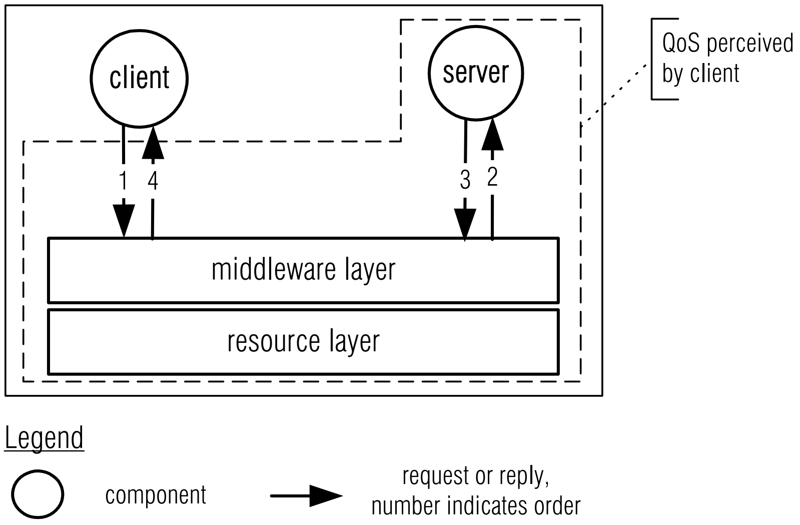
Middleware layer

As described in the previous sections, the QoS of a component based application depends on the distribution infrastructure. We use a common division (e.g., see [Bakken01]) of the infrastructure layer into a middleware layer and a resource layer. The *resource layer* is the layer that provides the resources that make it possible for the components to execute and interact, the *middleware layer* is the layer that shields the components from the particularities of the resources and provides the distribution transparencies.

Figure 2-10 illustrates the layering by providing a functional view on interacting components. We only show one invocation, although all types of nested invocations are possible, we can represent this by modeling one invocation. The interactions between the components and the distribution infrastructure are by definition local, and are transported by the middleware using available resources. With a *local interaction* we mean that no communication over a network takes place. This does not imply the client and server have to be distributed, but they can be. We indicate with the dashed line that the QoS as perceived by the client depends on the middleware, resources and the server component.

Local interactions

Figure 2-10 Functional view on middleware and components



To be able to reason about the QoS mechanisms that are part of the middleware, and to be able to reason about allocation of resources to components, we have to further refine resources and middleware.

Resources

We distinguish two types of resources; processing resources and network resources. *Processing resources* are provided by a node. Examples of processing resources are clock cycles on the CPU, or threads. Since processing resources are part of a node, they are shared by all the components located at that node. The QoS is dependent on the processing resources allocated to components.

Processing resources

Network resources are provided by the network and allow communication between nodes. Examples are bandwidth or buffer space. The QoS depends on the network resources to transport the request and reply. Nodes are connected by network connections that represent the network resources. The network resources are shared by the nodes connected to the network. How they are shared between the nodes depends on the network topology.

Network resources

The QoS as perceived by a client component for a specific interaction depends on the amount of allocated processing and network resources during that interaction. Every node has one *nucleus* that controls and

Nucleus

provides access to the processing resources of that node. A typical example of the nucleus is the operating system of a computer. The nucleus also provides access to the network resources. However, the control of the network resources is shared with the other nuclei because the networking resources themselves are shared with the nodes connected to the network. The amount of control and how this is divided over the nuclei depends on the type of network.

Component Middleware

Component, template, unit of composition, unit of deployment

A *component* is a software entity that encapsulates its *behavior* and *state*, and that has one or more interfaces (see also Section 2.4). Components interact with each other via their interfaces, and only via their interfaces. Components have a unique identity, and are instantiated from a component *template*. Characteristics of a component are that it is a *unit of composition* and a *unit of deployment*. A component template defines the behavior of the components that are initiated from the template.

Capsule

Middleware provides a component with distribution functionalities while shielding the component from the peculiarities of the nucleus and the resources that are used to provide the distribution. Components are grouped in capsules. A *capsule* is an encapsulation of processing resources. A typical example of a capsule is a process in an operating system. The middleware is present in every capsule, and requests resources from the nucleus for all the components that execute within the capsule. *Figure 2-11* depicts this. Contrary to *Figure 2-10* and earlier figures, *Figure 2-11* does no longer abstract from the distribution aspects, e.g., it shows that the components are distributed over two nodes. In addition, it shows that the middleware layer as we depicted it in earlier figures, is now split up in a set of middleware parts.

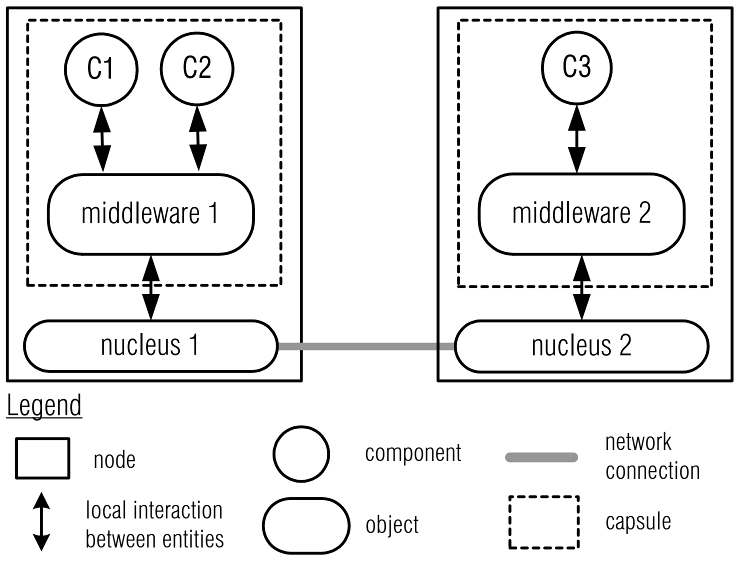


Figure 2-11 A view of a distributed system with middleware as a black-box

A node typically has several capsules, and a capsule typically contains several components. *Figure 2-12* shows the cardinality.

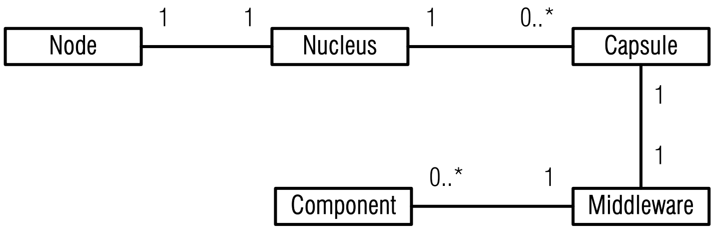


Figure 2-12 Cardinality of middleware, capsules etc

Since QoS mechanisms in the resource layer are out of the scope of this thesis, we do not need to refine the network or nodes further. Our focus is on QoS mechanisms in the middleware layer, and we therefore will refine the middleware part of the above model.

As is the case in current middleware technologies such as EJB and CCM, we divide middleware into 3 different parts; core middleware, common services and container.

Core middleware

The *core middleware* provides the basic invocation and communication functionality. It is responsible for locating the server component, transporting the request and replies and marshalling of the parameters.

Common service

The *common services* provide functionality that goes beyond basic communication and is commonly used by components. The purpose of this is two-fold: to relief the component developer of developing this functionality himself, and to enhance interoperability between components.

The interoperability is enhanced because the common services located at different nodes can interact using standard interfaces. For example, in case of a security service the same encryption algorithm will be used at client and at server side. Although there is no agreement in the literature which services to consider a common service, there is a consensus that this includes directory (or naming), transaction, security, persistence and event services [MDA01]. Another term sometimes used for common services is *basic services*, or *pervasive services* [MDA01].

Container

The *container* provides the context for the components to execute. There can be several containers within one capsule, each providing a different type of execution context. The container is responsible for lifecycle management functionality, deployment functionality and access to the common services. With *deployment functionality* we refer to the possibility to control the behavior of a component in a very late stage, i.e., at deployment time contrary to at development time. This enhances re-use of a component, and makes the component more independent of the environment it runs in compared to not having deployment functionality. The container has a separate interface, called deployment interface, to control the deployment functionality.

Deployment

Figure 2-13 shows the composite relationship between middleware and its parts, with the cardinality. Some middleware technologies do not support the container concept, e.g., CORBA, DCOM and RMI. This type of middleware technologies is sometimes also referred to as *object middleware*.

Object middleware

Figure 2-13 Core middleware, Container and Common Services

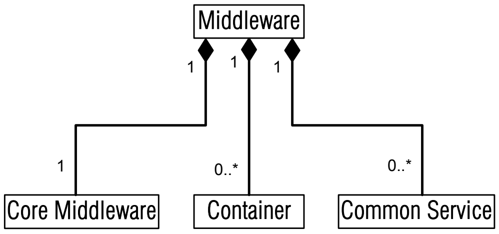


Figure 2-14 is a refinement of *Figure 2-11* in which middleware is no longer depicted as a black-box. The middleware is split into the container, common services and core middleware.

The concepts presented in this section are used throughout this thesis, and we will detail them further where necessary, for example to describe details of a specific QoS mechanism.

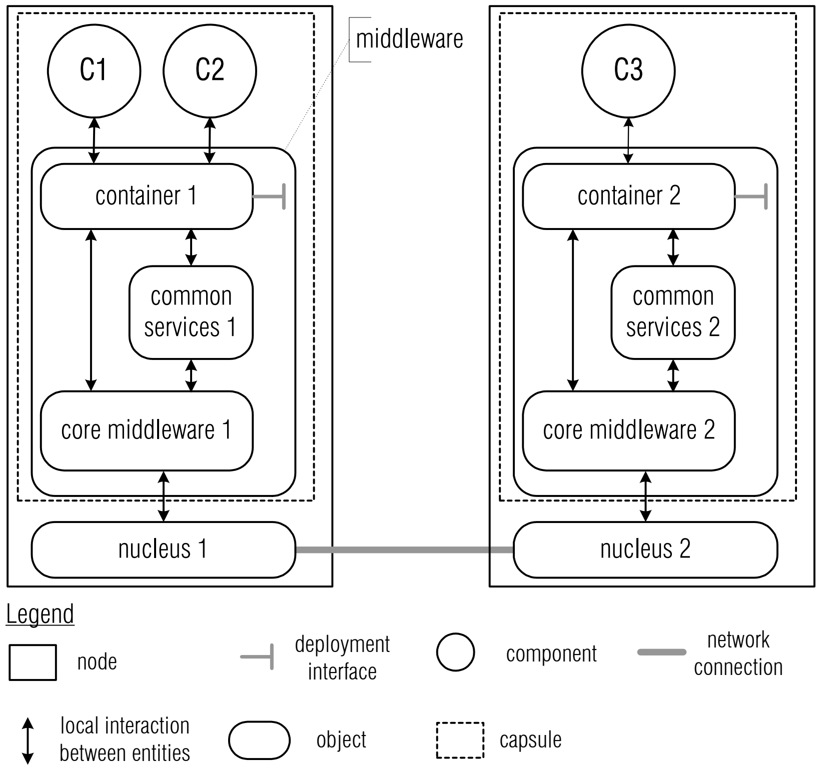


Figure 2-14 A view on middleware

2.6.2 Comparison with RM-ODP

Our terminology to describe distributed systems and component middleware has many similarities with the computational and engineering viewpoint languages of RM-ODP. Specifically, we also use object as a central modeling entity, and re-used capsule, node, nucleus, and distribution infrastructure. Also the concept of transparency is used throughout this thesis. However, we focus on component-middleware-based systems, and not on any type of distributed system, which makes some RM-ODP concepts superfluous for us. We will not list all RM-ODP concepts we do not use and explain why not, but for example we do not use engineering objects. These are not suitable to describe the middleware functionality. For example, the concept of container as it exists in current middleware technologies cannot be properly described using the RM-ODP engineering language.

2.6.3 Comparison with Szyperski's Definitions

There are similarities between the concept of component as described by Szyperski [Szyperski98] and how we define it. Notably concepts such as unit of deployment, unit of composition and explicit context dependencies are shared. Szyperski's component definition however is more focused on third party composition aspects, and thus software re-use, while we focus more on the distribution aspects of component-based applications.

Another difference is that our definition of component is stricter than Szyperski's. Contrary to Szyperski we assume that components are instances of a component template. Our components have an identity, have state, every interface has a unique identity, and the lifecycle of the interface is linked to the lifecycle of the component as a whole. A component in Szyperski's definition however cannot be instantiated, and only objects inside it can be instantiated and have an identity and state. Szyperski does not specify any correlation between the lifecycle of interfaces and of the objects, so anything is possible. Thus, Szyperski's definitions are broader, and the lifecycle of interfaces is not linked together. For example, a component in Szyperski's definition could be a library, which is not possible in our case. Our component corresponds with the simplest case of Szyperski's component in which a component is a class [Szyperski99]. With the same reasoning, Szyperski's component is similar to our component template.

We choose this more strict definition to make the definition of a component correspond better to current component middleware technologies, such as EJB and CCM.

2.6.4 Comparison with Component Middleware Technologies

Our definition of a component is compliant to the EJB and CCM definition of component. We generalized the EJB and CCM definitions of component, and abstract from specific technical peculiarities of these technologies. For example, we do not specify different types of ports, such as CCM. Contrary to EJB we do allow the possibility of several interfaces. Also we do not distinguish between session and service type of components, since this is not relevant for us. In our component definition a component always has state, we do not divide components into stateful and stateless types. When necessary, we will consider stateless component as a special type of component for which certain issues such as state consistency between replicas becomes trivial.

Our concepts of container, core middleware and common services are based on and can easily be mapped to EJB and CCM/CORBA. Our deployment interface is a generalization of the deployment descriptor, as it exists in CCM and EJB. Our definition of component also fits the older middleware technologies RMI, CORBA and DCOM. The main differences are that they do not have the concept of container, and do not have the unit of deployment characteristic of our definition of component. These older generation of middleware technologies do not enforce the encapsulation characteristic of our component definition and allows components to interact in other ways, such as, e.g., bypassing the middleware and calling co-located components directly.

2.7 Related Work

The current middleware technologies, as discussed in Section 2.3, do not support QoS, or only have a very limited support for QoS. There is research in academia however on QoS for component middleware, which we discuss in this section.

We limit ourselves here to work with an overall QoS approach, contrary to related work that focuses on one QoS mechanism or one QoS characteristic. Later in this thesis when we focus on specific QoS mechanisms we discuss related work specific to each of those QoS mechanisms.

QoS Modeling Language and the QoS Runtime Representation

The Quality of service Modeling Language (QML) [Frolund99] is a language for defining QoS specifications for distributed objects. QML originates from HP Laboratories, Palo Alto. QML has been designed to support QoS specification in a general way, encompassing QoS categories such as reliability, performance and security.

QML has three main language constructs that are used to construct a QoS specification:

– *Contract type* – A contract type specifies the QoS category, for example reliability or performance.
– *Contract* – A contract is an instance of a contract type, and represents a particular QoS specification.
– *Profile* – A profile associates contracts with interfaces entities. A common example of an interface entity is an operation.

For each QoS category, the contract type defines the QoS dimensions. A *QoS dimension* expresses the values that can be used to express a QoS contract. A dimension has a domain of values. A domain can be ordered, i.e., in an ordered domain it is possible to compare two values and know which value indicates a better QoS. There are three types of domains: set, enumerated and numeric. A set is unordered, enumerated domains have a user-defined ordering, numeric domains have a built-in ordering. For every ordered dimension you can specify if it is increasing or decreasing, which indicates whether a higher value is better or worse than a lower value. It is possible to indicate some statistical constraints in a contract, e.g., the latency has to be within certain boundaries in 90% of the cases.

QML supports the notion of conformance between contracts. Contract P conforms to contract Q if contract P is stronger than contract Q. This relieves developers of making exact matches between contract types. It is only necessary to find an operation whose specification is at least as strong as needed.

The *QoS Runtime Representation* (QRR) makes it possible to manipulate and create QoS contracts at runtime. QRR is purely focused on runtime manipulations of the QoS contracts, i.e., it does not prescribe how these contracts should be enforced or monitored.

Summarizing, QML offers a language for specifying QoS contracts and contract types, together with a runtime representation (QRR) of this language. QML does not prescribe any specific QoS categories, QoS dimensions or QoS contract types.

Quality Objects

Quality Objects (QuO) supports the specification of QoS contracts between clients and service providers, runtime monitoring of contracts, and adaptation to changing system conditions. It is based on CORBA. QuO is developed by BBN Technologies [Zinky97, Vanegas98].

The QuO framework consists of the following components:

- A local *delegate* of the remote object. This delegate has the same interface as the remote object, but it can trigger contract evaluation upon each call and return.
- A *QoS contract* written in a *Quality Description Language* (QDL) between client and object.
- System condition object interface, used to measure and control QoS.

When a client calls a remote method, the call is passed to the object's local delegate. The local delegate will then pass the call on to the remote object. While doing so the delegate is able to record the current system conditions. The method return will also pass through the local delegate and the delegate is so able to evaluate whether the QoS requirements were met or if it has to take action in order to fulfill the requirements for subsequent remote method invocations.

At runtime, client and server interact about the level of QoS they can provide. Clients and servers are notified of changes in QoS through callback interfaces. The QuO architecture provides objects that define these callbacks; the developer is responsible for implementing their behavior. The delegates will try to modify their behavior to maintain the systems current

QoS. For example, if the network throughput degrades to such a level that they may degrade below the agreed QoS specifications, the delegates can try to compress the data and thus to trade CPU cycles for throughput.

QDL is an extension of IDL that specifies an application's expected usage patterns and QoS requirements for a connection to an object. The QoS and usage specifications are at the object level (e.g., methods per second) and not at the communication level (e.g., bits per second). A client application can have many connections to the same server object, each with different system properties. QDL allows the object designer to specify QoS regions, which represent the status of the QoS agreement for an object connection. The application can adapt to changing conditions by changing its behavior based on the QoS regions of its object connections.

QuO has three focus areas; distributed systems that run over wide-area networks, embedded systems and security. It considers mainly network resources, processing resources are out of scope. A main difference between QuO and other research in this area is the concept that the application objects get feedback on the QoS characteristics through the QoS regions. In most approaches the QoS mechanisms just have to fulfill the QoS requirements, and there are no callback interfaces to inform the application objects to what extent the agreed QoS is actually achieved. Although with these callback interfaces QuO reduces the transparency since it impacts the application code, it does have the benefit that the application can adapt its behavior based on achieved QoS. The client side proxy concept that QuO uses is more common, the main difference with other approaches is the layered architecture that is proposed. An important issue with respect to the client side proxy is how transparent it is for the client developer. This is not clearly described. Also, the proxy concept can pose significant management issues, such as how to distribute the proxies and how to upgrade them. It is unclear how this is envisioned in QuO.

QuO is used by AQuA [Ren03], which implements fault tolerance for CORBA by using replication.

Quartz

Quartz [Siquera00] is a research project at Trinity College in Dublin, Ireland. Quartz translates high-level application specific QoS requirements into low-level, resource-layer QoS parameters. It uses a three-step translation for this. The application-specific QoS parameters are translated into a set of generic application-level QoS parameters, which are further translated into a set of generic system-level QoS parameters. These generic system-level QoS parameters are balanced between the network and the operating system, and translated into the system-specific QoS parameters.

The system-specific QoS parameters are specific for the resource-layer QoS mechanisms that are used. This idea is prototyped for the CORBA Audio/Video Service. As will be more elaborately explained in Chapter 3, our approach is more dynamic, and does not rely on resource-layer QoS mechanisms.

TAO

TAO [TAO] is a research C++ CORBA ORB that implements the real-time CORBA specification. It is developed by the University of Washington and University of California, Irvine, both in the USA. Other research groups also use TAO for prototyping QoS mechanisms. Examples of prototyped QoS mechanisms are a load balancing service based on replication of stateless CORBA objects (see Chapter 5), a fault tolerance service by Lucent called Doors [Man00B, Natarajan00], and pluggable protocols [Kuhns99]. We discuss the relation between TAO and our work in Chapter 3 when we discuss our approach in more detail, and in Chapter 5 we compare the TAO-based load balancing service with our load distribution service.

MASQ

MASQ uses aspect-oriented programming ideas to weave QoS with the application objects [Geihs01B]. This is done with the MICO CORBA ORB. The QoS parameters and QoS interfaces are specified in a special language called Quality IDL (QIDL). QIDL is an extension of CORBA IDL. The QIDL definitions are a part of IDL spec, thus the functional IDL specification cannot be implemented with different QoS characteristics. The IDL compiler functions as the aspect weaver. The work is very networking oriented, processing resources are not considered. There is a lot of focus on a QoS catalog of QoS mechanisms. The published papers do not make clear what the actual QoS mechanisms are, and how the weaving of QoS mechanisms with the application code takes place. Because of this it is not clear what the impact is on the application code. Examples have the typical `get_state()` and `set_state()` operations, which would require intimate knowledge of the application code and we do not see how a tool could weave this automatically.

Monet

The Monet group at University of Illinois at Urbana-Champaign (US) does research on QoS aware middleware [Nahrstedt01]. Their approach is based on calculation of required resources using a so-called QoS compiler. The QoS compiler compiles high level QoS requirements into low level resource QoS mechanisms. A second characteristic of their approach is that during run-time and based on available resources one of different preselected configurations of the application is chosen. How these configurations are selected, how they differ and how this impacts the design or implementation of the application is not clearly described. We discuss the relation between this work on our work in Chapter 3 when we describe our approach in more detail.

MULTE-ORB

The MULTE-ORB is a research orb for distributed multimedia applications from several groups in Norway [Eliassen02]. Goals include:

- The ability to provide dynamic QoS support, meaning that users can change their QoS requirements at any time.
- Evolution of QoS requirements, new media types and new applications might introduce new QoS characteristics.
- Transparency versus fine-grained control, MULTE should provide high level and lower level QoS requirements.
- To control the QoS mappings using policies.

Central in the design of MULTE-ORB is the concept of explicit binding, which is similar to the binding concept in RM-ODP. A stream model was developed to describe the QoS properties of a stream. Media gateways are used within a binding to modify the format and/or QoS of a media stream, when necessary. A trading service is used to locate a gateway with the appropriate capabilities. MULTE-ORB is streaming focused, contrary to the focus on operational interface we have in this thesis.

Globus

The Globus Project develops technologies needed to build computational grids. Grids are persistent environments that enable software applications to integrate instruments, displays, computational and information resources that are managed by diverse organizations in widespread locations. Part of Globus is GARA: Globus Architecture for Reservation and Adaptation [Foster00]. GARA allows resource reservations and adaptations in the resource allocation to enhance performance in the scientific computing domain. Typical examples of what GARA is used for and geared at are bulk data transport and distant visualization.

Contrary to the type of applications we consider, GARA is not remote procedure call based. Transparency does not seem to be a major concern for Globus. Although Globus tries to make life easier for the developer by providing resource reservation mechanisms, it does not hide the distribution or QoS aspects. Globus is focused on performance, not on availability.

Open ORB

The Open ORB Python Prototype (OOPP) [Andersen02] is an experimental middleware platform with QoS support for multimedia streaming. OOPP uses structural reflection, which allows inspection of the different parts that make up OOPP, e.g., to determine overflow of internal buffers. OOPP also uses structural reflection to effectuate a change of strategy if needed to achieve a certain QoS. OOPP is implemented in the programming language Python, which has reflective features.

We assume common middleware technologies, which do not support structural reflection. Also we focus on operation invocations, not on streams. We will discuss the usage of reflection, and the work done on reflection by the designers of Open ORB, in Chapter 3.

2.8 Concluding Remarks

The concepts we use in this thesis to describe component middleware are based on RM-ODP, on Szyperski's work on component-based development and on current component middleware technologies. RM-ODP is a standard for specifying distributed systems, and specifies concepts like distribution transparency and viewpoints. Szyperski's work on components defines concepts such as unit of deployment and explicit context dependencies. Current component middleware technologies such as CORBA and EJB have concepts such as container and common services, which are not part of RM-ODP or Szyperski's work, but are relevant for us. By defining our own generalized concepts, we avoid any technology dependencies, and can apply our research results to different component middleware technologies.

The concepts we use to describe component middleware include component, middleware, common services, container, core middleware, processing resources, network resources, capsule and node.

We limit QoS in this thesis to the performance and availability QoS categories. The performance QoS category consists of the response time and throughput QoS characteristics. The availability QoS category consists of the availability, reliability and maintainability QoS characteristics.

Current middleware technologies have no or only limited support for QoS and QoS differentiation. Related work in the area of QoS and middleware, such as QML and QuO, provides only partial solutions, or is very specific to certain application domains or middleware technologies.

Chapter 3

QoS Mechanisms in the Middleware Layer

This chapter[2] discusses our approach for QoS mechanisms for component-middleware-based applications.

We identify requirements for QoS mechanisms for component-middleware-based applications. These requirements are generic, i.e., they are independent of any specific QoS mechanism.

Based on these requirements, we define our approach for QoS mechanisms. We describe and compare static and dynamic approaches to QoS provisioning, and motivate our choice for a dynamic approach. We discuss the roles of separation of concerns and transparency in our approach. We compare middleware-layer QoS mechanisms that rely on resource-layer QoS mechanisms and those that do not, and motivate our choice for the second category.

We discuss QoS mechanisms that are possible within our approach, and select two concrete mechanisms that we will focus on in the remainder of this thesis.

A major issue for the design of our QoS mechanisms is how to maximize the transparency for the component developer who uses a QoS mechanism. We discuss reflection, and how we can use reflection and especially message reflection to tackle this issue. We also identify and evaluate different ways to implement message reflection in common middleware technologies.

The approach we describe in this thesis is applied to specific QoS mechanisms in the next two chapters. These chapters also list requirements specific for those QoS mechanisms.

Section 3.1 contains our generic requirements for QoS mechanisms for component-middleware-based applications. Section 3.2 describes and motivates our approach. Section 3.3 selects two mechanisms that are the subjects for the next chapters. Section

[2] Parts of this chapter have been published in papers [Wegdam00A], [Wegdam00B], [Wegdam01C] and [Kath00], which are co-authored by the author of this thesis.

3.4 discusses the usage of reflection to achieve a clear separation of concerns. Section 3.5 contains the conclusions.

3.1 Requirements for QoS Mechanisms

This section formulates requirements for QoS mechanisms for component-middleware-based applications. A QoS mechanism enhances the QoS, and the requirements formulated in this section define the conditions on the behavior of the QoS mechanism, the interactions it may have with its environment, and the conditions it poses on the application design and implementation. The requirements listed in this section are generic, i.e., they are not specific for a certain QoS mechanism.

Generic requirements for QoS mechanisms

We distinguish two stakeholders for a QoS mechanism, each with their own set of requirements:
- The component developer that implements application components that use the QoS mechanism.
- The middleware developer that implements QoS mechanism.

We first describe the requirements from the perspective of the component developer, and then the requirements from the perspective of the QoS mechanism developer. Each requirement is identified with a keyword that will be used in the remainder of this thesis to refer to the specific requirement.

Component developer's requirements
1. *Telematics* - The QoS mechanisms should at least be suitable for telematics systems. However, when possible the QoS mechanisms should not be restricted to a specific application domain at all. Telematics systems have these characteristics (see also Chapter 1):
   - They are inherently distributed.
   - The amount of users, and the usage in general, varies erratically in time.
   - The end-users have certain performance and availability expectations that have to be met, but there are no hard, real-time requirements.
   - Because of the different parties involved, there is inherent heterogeneity in used networks, operating systems and computers.
   - They are large scale.
   - They process large amounts of (parallel) interactions.

   These characteristics are not unique for telematics systems, other domains with large scale distributed system have similar characteristics, e.g., banking systems. We however pose this requirement here since the research takes place in a telematics context.

2. *Flexibility* – Since components are subject to third-party composition, the component developer does not know the QoS requirements for the component he is developing. Limiting the range of QoS requirements that can be supported limits the possibilities for third-party composition, and for re-use in general. The choice which QoS mechanisms are used determines the range of QoS requirements that can be supported. This choice should be made as late as possible in the development cycle to increase the flexibility on the QoS requirements that can be supported. Ideally, this choice is made at run-time, and design or implementation dependency on QoS mechanisms should be minimized. Design and implementation dependencies cannot always be avoided, e.g., some QoS mechanisms need access to the application state, which has to be implemented by the component developer.
3. *Time* - A component developer using the QoS mechanism should spent less time on the QoS aspects than would be the case if he did not use the QoS mechanism. This decreases the total development time that is needed, and a larger percentage of the development time is spent on the application logic. To make this more precise, we split up the development time needed for a component as follows:
   $$t = a + q + o$$
   with
   t = total development time needed to develop a component
   a = development time needed to develop the application logic
   q = development time needed to develop the QoS aspects
   o = development time needed to develop other aspects
   By decreasing the development time needed to develop the QoS aspects, the total development time needed also decreases. The time spent on the application logic and the other aspects is not affected by using the QoS mechanism. The responsibility of the component developer for the QoS aspects should be limited to specifying the required QoS, and not on enforcing this QoS. Because the component developer has to specify the QoS, he will still spent some time on the QoS aspects. As a result of the decrease in total development time for a component the time-to-market decreases and the development costs decrease.
4. *Expertise* - A related but separate requirement from the time requirement is that using a QoS mechanism should not require special expertise from the component developer on the implementation of that QoS mechanism. The time requirement already states that that component developers should only focus on the application logic, but are still expected to specify the QoS they need. The way the component developer has to specify QoS however should not require expertise on the QoS mechanism. Or put differently, any interface that a QoS mechanism exposes to the component developer should not reveal any

implementation details of the QoS mechanism. We do not want to require the exact way to specify QoS, but below we give an example for each of the five QoS characteristics we consider in this thesis:

- throughput – 25 invocations per second
- response time – 150 ms
- availability – 99%
- reliability – 16 days
- maintainability – 20 minutes.

A QoS mechanism that fulfills this requirement will, when used by a component developer, decrease development costs because more developers will be qualified to develop components.

5. *Heterogeneity* - The QoS mechanisms should be suitable to run in heterogeneous environments, i.e., on different types of networks and nodes. We want to minimize the assumptions we make on the nodes and networks that are used, including the resource-layer QoS mechanisms that these networks and nodes support. Minimizing the assumptions we make on the resource layer maximizes the number of different types of environments we can use the QoS mechanism in, e.g., different UNIX versions, different Windows versions, real-time and non-real-time operating systems, non-IP networks.

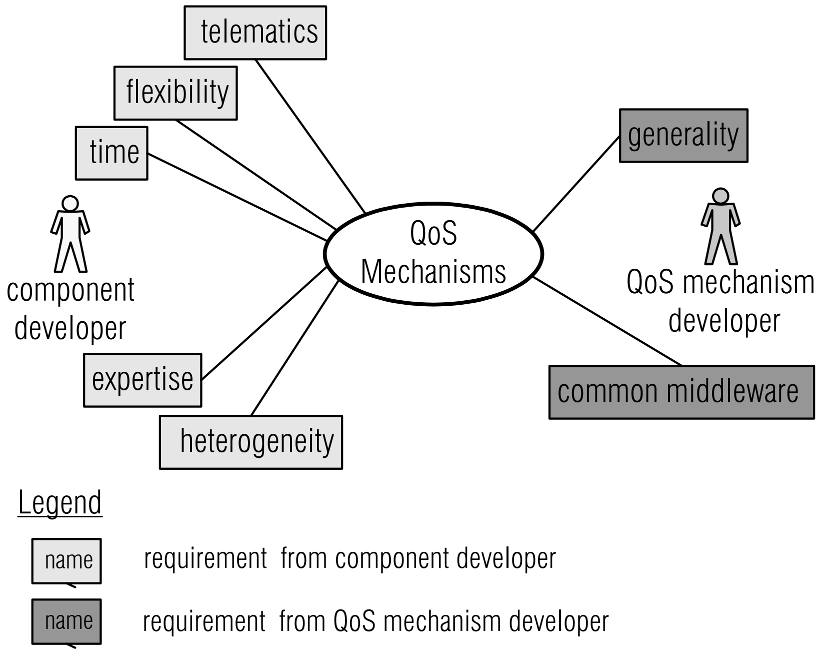
QoS mechanism developer's requirements

6. *Generality* - The QoS mechanisms should be generic, i.e., not be application specific. This allows re-use of QoS mechanisms. A QoS mechanism that fulfills this requirement can be used for a wide range of applications, and thus enlarges the potential market for this QoS mechanism. In fact, this requirement applies to middleware in general.
7. *Common middleware* - The QoS mechanisms should be applicable to common-off-the-shelf component middleware. The QoS mechanisms should not be specific for some research middleware, or require changes to the middleware source code. Common-off-the-shelf middleware represent the current consensus on what functionalities and concepts should be supported by the middleware, which we want to extend in our research. If we need changes to these functionalities and concepts to be able to implement a QoS mechanism, we limit the possibilities to use that mechanism. In addition, the impact of our research increases if this research can be applied to popular middleware technologies, including the possibilities to submit it for standardization. Using common middleware allows us to better position and compare our work to other work that is based on the same middleware technology. A last very practical reason it that we do not want to spent a lot of effort on re-implementing functionality that is already part of existing middleware.

Although this requirement poses some restrictions on our design choices, the above-mentioned benefits outweigh these restrictions.

Figure 3-1 gives an overview of the requirements and shows their origin.

Figure 3-1 The requirements and their stakeholder



3.2 Our Approach for QoS Mechanisms

This section defines our approach for QoS mechanisms for component-middleware-based applications.

3.2.1 Dynamic Approach

We can divide approaches for providing QoS into static and dynamic approaches. This is a common division, see for example this survey of QoS architectures [Aurrecoech98], or [Molenkamp01,Foster00]. We will introduce both approaches, then elaborate on them and discuss their suitability for QoS mechanisms based on our requirements.

The static approach is based on calculating the required resources needed for obtain a certain QoS, and reserving these resources for the lifetime of the application, or session. The dynamic approach is based on monitoring the achieved QoS during run-time, and adapting the application behavior or resource allocation if the achieved QoS does not fulfill the QoS requirements.

Static Approach

The static approach is a commonly used approach to achieve a certain level of QoS [Molenkamp01, Foster00, Aurrecoech98, Franken96]. In a static approach the maximum amount of resources that would be needed to obtain a certain level of QoS during the lifetime of an application is calculated, and this maximum amount of resources is then reserved for the whole lifetime of the application. In literature this approach is sometimes called *static QoS management* [Aurrecoech98], sometimes *reservation technique* [Foster00] or sometimes *static approach* [Molenkamp01]. We will use the latter term.

Below we identify four problems with the static approach:

– *Usage* - To be able to calculate the required resources, an estimation on the usage of the application is needed, e.g., the number of users, or the number of interactions. For certain application domains, for example single user applications or application for a fixed set of users, this might be possible, but as stated above for the telematics type of applications that we consider the number users can be vary erratically and is difficult to predict.

– *Resources per usage* - To be able to calculate the required resources, a quantification of the resources needed per usage, or per user, of the system is needed. An example of usage of the system is making a reservation for a movie, which will result in one or more remote invocations. To calculate the amount of resources per usage requires a lot of knowledge about the application, used systems, used programming language, used compiler etc. This conflicts with the heterogeneity that is inherent to telematics systems. And since this calculation is partly application dependent the application developer will have to do part of it, which is a burden we want to avoid. As an example, in the case of a Java-based CORBA application, the amount of resources to send an invocation will depend, among others, on the length of the parameters, the type of parameters, the threading model, the settings of the garbage collector, the used virtual machine and the operating system.

– *Shared resources* - A static approach requires knowledge on the usage of the resources by other applications. In the telematics domain, resources are typically shared, and other applications will have their own resource needs that vary in time. The amount of resources available to one application will thus depend on the amount of resources that other application will use. A trivial solution is not sharing the resources, but this leads to underutilization of resources since typically the average usage will be much lower than the peak usage.

– *Failures* - Failures are inherently unpredictable. Since with a static approach the amount of resources is fixed, it is not possible to change the resource allocating in case of resources failures (e.g., node failure). This is an undesirable limitation for QoS mechanisms that enhance the availability.

Dynamic Approach

The alternative to a static approach is a dynamic approach. A *dynamic approach* does not use a fixed resource allocation, but dynamically adapts during run-time the resource allocation and behavior of the application based on achieved and required QoS. This dynamic approach is in literature sometimes referred to as *dynamic QoS management* [Molenkamp01, Aurrecoech98] or *adaptation technique* [Foster02]. We will use the term dynamic approach.

A dynamic approach typically uses a so-called control system [Bergmans00], in which the achieved QoS is measured and compared to the required QoS. If the achieved QoS is insufficient, a control action is taken to correct this. A control action could be to change the resource allocation, or to change the application behavior.

If we apply this to QoS mechanisms to component-middleware-based applications, we need to address the following issues:

– *Monitoring functionality* - The monitoring of achieved QoS requires alterations to the application and/or to the core middleware. We want to control the QoS as perceived by the client component, this means we have to measure the actual QoS the client component receives. See also [Molenkamp02] for a discussion on this. For example, suppose the required response time should be less than 150 ms in 90% of the cases, and always less than 300 ms. To be able to determine if the achieved QoS fulfills this required QoS, we have to record the time that the client component starts the invocation, and the time it receives the reply. We can either do this at the application layer by requiring the component developer to alter his code with timing statements, or do this in the core middleware just before marshalling the request, and after de-marshalling the reply.

– *Control functionality* - A similar reasoning also applies to the control functionality that is required. To adapt the resource allocation and behavior of the applications extra functionality is needed in the middleware or application layer.

– *Decide on adaptation* - Although a dynamic approach does not require us to calculate the resource needs, we do need to make some assumptions on the resource usage to be able to decide which adaptation would improve the achieved QoS. We do not have to be able to quantify the change in QoS for every control action, it is sufficient to know what control action will positively influence what QoS characteristic. Fortunately, this does allow a simple heuristic algorithm, since this does not require specific knowledge on the application or on the resources that are used. Should we make an adaptation that does not achieve the desired improvement in QoS, we can correct or supplement it with new adaptations.

– *Statistical QoS requirements* - The dynamic approach basically tries different resource allocation to find one that fulfills the QoS requirements. The problems with this are that this resource allocation might not exists, or the heuristic algorithm might not find a suitable resource allocation. The inability to achieve the required QoS is then discovered during run-time. In addition, the adaptations can have a temporarily negative effect on the achieved QoS, e.g., the migration of a component to a node with more processing resources will make the component temporarily unavailable. Because of this a dynamic approach is best suitable for statistical QoS requirements in which some percentage of the time certain QoS requirements will not be met. Fortunately, for the telematics domain this is typically acceptable, e.g., a temporary and small degradation in performance is acceptable. This contrary to hard real-time systems, for which not fulfilling a QoS requirements can have disastrous consequences, and is therefore always considered a failure. For applications that have those hard QoS requirements, a dynamic approach is not suitable.

Conclusion

Based on the above discussion on the static and dynamic approach, we select the dynamic approach. The reasons for this are that a static approach is not suitable for the telematics domain because of the difficulty to predict usage, needed resources per usage, available resources due to the sharing of resources and inherent unpredictability of failures. In addition, a dynamic approach is more efficient with resources since resources are only allocated to an application when it actually uses them (contrary to allocating the resources for the whole lifetime of an application).

An issue with the dynamic approach is the need for monitoring and control functionality. This functionality has to be added to the application and/or middleware layer, and can potentially violate the flexibility, time, expertise and common middleware requirements. To prevent violation of these requirements, we have to provide the monitoring and control functionality without burdening the component developer, or requiring changes to the middleware. This is further discussed in the next subsection, and in Section 3.4.

3.2.2 Separation of Concerns

Separation of concerns is an important principle underlying the construction of complex systems. The viewpoints of RM-ODP are based on this principle. As stated in Chapter 2, a viewpoint consists of set of concepts, structures, and rules that are different for each viewpoint. The separation of concerns principle is used here because every viewpoint focuses on different concerns. Also the RM-ODP distribution transparencies are based on separation of concerns by allowing a developer designing a distributed application not to be concerned with the details on how this distribution is established.

Distribution transparency is the property of hiding from developers the details and the differences in mechanisms used to overcome problems caused by distribution (see Chapter 2). We give here two examples of how this transparency principle is implemented:

1. Access transparency hides the differences in computer architectures and programming languages to enable interworking across heterogeneous computer systems. The component developer thus does not need to be concerned with what programming language or operating system is used by the component he wants to interact with. The component developer however does need to be aware that the interactions are passed through some middleware platform, and there are restrictions on these interactions that local interactions in the same programming language do not have. For example, in CORBA a developer has to design the interface in IDL, and not in the programming language he is using. Also simple types in IDL that he can use might be quite different than the simple types in the programming language he is using. In Java RMI, the interface design is in the Java programming language, which makes it easier for the developer to design the interfaces, at the cost of being able to use other programming languages.
2. Failure transparency enhances the fault tolerance by masking the failure and possible recovery of objects. The component developer does not need to be concerned about the types of failures that are masked by the failure transparency. CORBA standardized fault-tolerant CORBA for this, and thus offers failure transparency. Fault tolerant CORBA however will always require the developer of a component that should be recoverable to implement state access methods. The component developer thus has to adapt his design and implementation be able to use fault tolerant CORBA.

As the above examples show, transparency hides the details, differences and implementation of the distribution, but the transparency has its limits. The distribution aspects are not completely hidden. Transparency is not a boolean value, and it is possible to compare two mechanisms on their transparency property. For example, "mechanisms A is more transparent than mechanism B", if mechanism A is less intrusive to the application code than mechanisms B. We can conclude that *full distribution transparency*, i.e., fully hiding the complexities of distribution for the component developer, is not achievable.

Full distribution transparency

A similar reasoning can be found in [Waldo94], where it is argued that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space.

With full distribution transparency as an unachievable goal, the challenge is to achieve a degree of transparency that is *as transparent as possible* by moving the complexity of the distribution aspects as much as possible into the middleware. This also applies to QoS support for component-middleware-based applications. We want to extend the existing distribution transparencies offered by component middleware with QoS transparencies. We define *QoS transparency* as the property of hiding from application developers the details and differences of the mechanisms that are used to achieve the required QoS for an application. The purpose of QoS transparencies is to shift the complexities of achieving a certain QoS for distributed systems from application layer to the supporting infrastructure (the middleware). This definition makes QoS transparencies a specialization of the distribution transparencies, i.e., the set of QoS transparencies is a subset of the set of distribution transparencies (see also *Figure 3-2*). Examples of QoS transparencies are load distribution transparency and replication transparency. An example of a distribution transparency that is not a QoS transparency is access transparency.

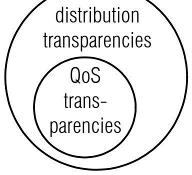


Figure 3-2 QoS transparencies and distribution transparencies

Using a QoS mechanism that offers some QoS transparency does not mean that the QoS aspects of the application are hidden from the application developer, or that the application developer can abstract from QoS. However, he will be able to focus on the application logic embedded in the component, and on application-layer QoS. The application developer does not need to be concerned with the details of how the QoS is achieved by the distribution infrastructure (middleware). It is desirable for the application developer to have a control interface to pass the QoS requirements to the middleware (see also our time requirement). This interface should be at the appropriate abstraction level, and should not require expertise of the application developer on the specific QoS mechanisms (see our expertise requirement). However, even at the right abstraction level, such an interface could be considered a violation of transparency, making full QoS transparency not only unachievable but also undesirable.

3.2.3 Middleware-layer-internal QoS Mechanisms

Our heterogeneity requirement states that middleware-layer QoS mechanism should minimize the dependency on resource-layer QoS mechanisms. In this section, we discuss dependency on resource-layer QoS mechanisms, divide middleware-layer QoS mechanisms into those that do rely on resource-layer QoS mechanisms and those and do not, and motivate our choice for the later category.

QoS mechanisms, independent if the are inside the middleware layer or not, can enhance QoS by providing functionality that improves one or more QoS characteristics, without relying on other QoS mechanism. This can be compared to TCP layer, which adds reliability to the IP layer by resending lost packages. The functionality that improves the QoS, in this case the resending of lost packages, is inside the TCP layer. There are also QoS mechanisms that rely on QoS mechanisms in a lower layer to actually provide the functionality that enhances the QoS. An example is a QoS mechanism that would add reliability by changing the standard Ethernet parameter to increase the number of retransmissions.

The reason we do consider QoS mechanisms that rely on lower- layer QoS mechanism to enhance QoS is because such a QoS mechanism does offer a different interface to control the QoS that is typically easier to use than the interface of the lower-layer QoS mechanism it relies on.

When we apply this distinction to middleware-layer QoS mechanisms, we can divide these mechanisms into mechanisms that rely on the resource-layer QoS mechanisms, and mechanisms that do not rely on resource-layer QoS mechanisms:

Mapping QoS mechanism

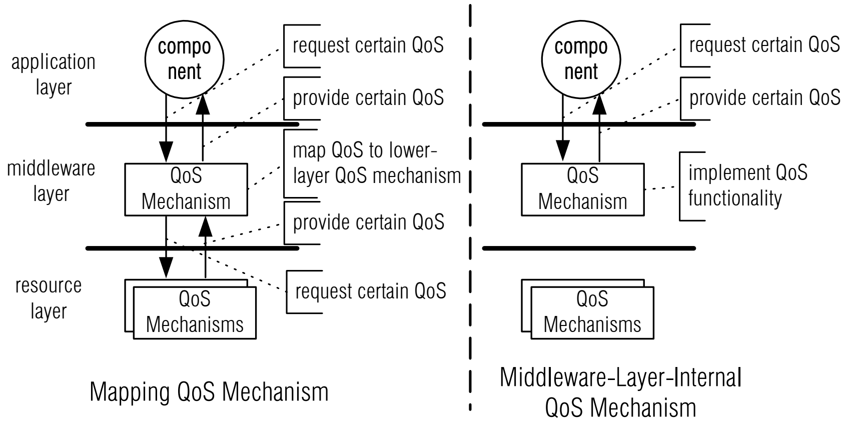
– *Mapping QoS mechanisms* – There are middleware-layer QoS mechanisms that map to resource-layer QoS mechanisms, e.g., pluggable protocols [Halteren99B], QIOP [Halteren03], Monet [Nahrstedt01] and Quartz [Siquera00]. These mapping mechanisms offer a higher level interface to the application developer that is easier to use than the typically difficult to use lower level interfaces that the resource-layer QoS mechanisms provide. Mapping mechanisms also enhance the portability of the application because the mapping-mechanisms can use several different resource-layer mechanisms without exposing this to the component developer.

Middleware-layer-internal QoS mechanism

– *Middleware-layer-internal QoS mechanisms* – This are middleware-layer QoS mechanisms that do not use resource-layer QoS mechanisms, and instead enhance the QoS by using the functions that the middleware provides. Example of these middleware-layer-internal middleware category of QoS mechanisms are active replication mechanisms that multi-cast requests to a group of replicas, or passive replication mechanisms and load distribution mechanisms that direct invocations to another node during a session.

Figure 3-3 depicts the two categories of QoS mechanisms. It shows that the mapping mechanisms map the QoS requirements to resource-layer QoS mechanisms, contrary to the middleware-layer-internal QoS mechanism that do not rely on any resource-layer QoS mechanisms.

Figure 3-3 Mapping versus middleware-layer-internal QoS mechanisms



Middleware-layer-internal mechanisms can be used in a wider range of environments than mapping mechanisms because they do not require the support of specific resource-layer QoS mechanisms. This means middleware-layer-internal QoS mechanisms fulfill the heterogeneity requirement.

A second benefit of middleware-layer-internal mechanisms is that they can enhance the QoS beyond what the resource layer provides, while mapping mechanisms are by definition limited to whatever QoS mechanisms the resource layer provides.

Based on the above reasoning, we select middleware-layer-internal QoS mechanisms as the focus for thesis.

The two categories of mechanisms are complementary and they can be used together to provide the required QoS. For example, a mapping mechanism that uses DiffServ can be used to obtain sufficient network resources, together with an middleware-layer-internal mechanism that uses load distribution to obtain sufficient processing resources. There can be feature interactions issues between the mechanisms, this depends on the mechanisms involved. This is out of scope for this thesis.

3.3 Possible QoS Mechanisms

This section discusses what different QoS mechanisms are possible within our dynamic, middleware-layer-internal approach. We first discuss mechanisms that enhance the performance characteristics, and then discuss mechanisms that enhance the availability characteristics. In addition, we select two mechanisms that will be the focus for the remainder of this thesis.

3.3.1 Performance Mechanisms

As stated in Chapter 2, we consider two performance related QoS characteristics:
- response time
- throughput.

To have a basis for discussing how we could enhance these characteristics with middleware-layer-internal QoS mechanisms, we first discuss performance in middleware-based systems.

A remote invocation sequentially passes through the following steps:
1. client middleware – marshal the request
2. client nucleus – pass the request to the network
3. network – transport the request
4. server nucleus – pass the request to the server middleware
5. server middleware – unmarshal the request
6. component – actual processing of the request
7. server middleware – marshal the reply
8. server nucleus – pass the reply to the network
9. network – transport the reply
10. client nucleus – pass the reply to the client middleware
11. client middleware – unmarshal the reply

These steps are depicted in *Figure 3-4*. For a local invocation, i.e., an invocation between components that are located on the same node, or even in the same capsule, some of the above steps will be skipped. Since this does not affect the line of reasoning presented here, there is no need to distinguish this as a separate case.

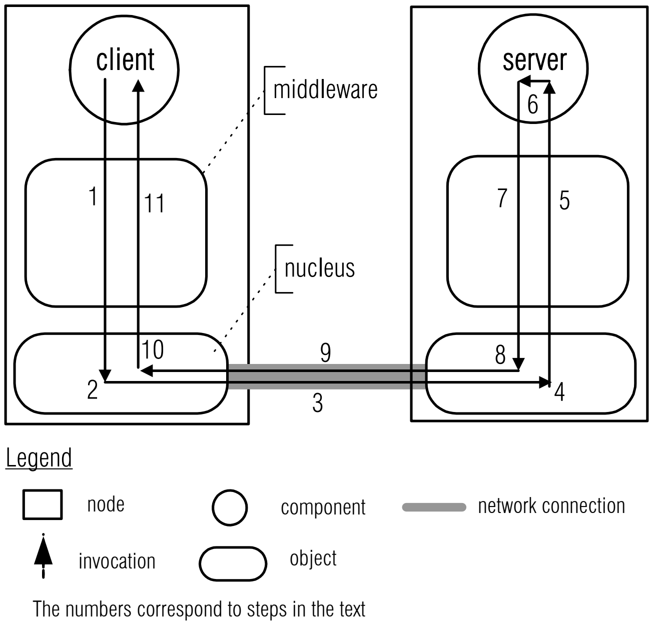


Figure 3-4 Steps for a remote invocation

The total response time is the sum of the times that each of these steps take. The time each step takes depends on the available resources to perform this step. The throughput depends on the available resources, but also on the usage of parallelism of the application and the middleware. How parallelism is used, and how this influences the throughput, depends on the implementation of the middleware, application and nucleus.

Network Related Steps

Elaborating on the network related steps, the required amount of network resources for a single invocation depends on the size of the request and of the reply. The size of the request and reply depends on the size of the parameters, plus some overhead to encode method name, identify the target component etc. The amount of time required to send the request or reply, depends on the amount of network resources that is allocated to the (typically TCP) connection that is used to send them. In the normal case, there is no QoS control mechanism to control this allocation, and the available bandwidth is equally shared over all connections, i.e., a best-effort network. If the network does support some QoS mechanisms, such as IntServ [IntServ94] or DiffServ [DiffServ98], the middleware can request a certain amount of resources, or a certain priority class, for a connection. Commercial middleware has little or no support for network resource reservation, but in some research or prototype middleware implementations do support this (e.g., [Halteren01, Halteren99B]). Also the middleware can prioritize certain request over others, and thus influence response times for certain requests. Prioritizing requests does not influence overall throughput, but the throughput for certain components, or for certain component-component interactions can be influenced.

Processing Related Steps

The processing in the client and server middleware, and the processing of the request in the server component will require a certain amount of processing resources. The available processing resource for this determines the amount of time this takes. The nucleus divides the available resources of the node over the different capsules, including the capsules the client and server component execute in. Controlling the scheduling algorithms of the nucleus is not possible in all operating systems, or if possible can be quite limited. Besides, as we stated already, we do not use resource-layer QoS mechanisms.

Within the capsules, given the amount of processing resources the nucleus allocates to the capsule, the middleware can have its own division and prioritization of the resources over the tasks it has to perform. This offers possibilities at the server side where the middleware will typically have a queue for incoming requests and assigns requests to threads to be processed. For more background on this, see [Schmidt98] for a comparison of multi-threading strategies in CORBA. The scheduling algorithm for assigning requests to threads can be used to influence the response time by prioritizing certain requests over others (for an example see Real-Time CORBA [RTCORBA]). The most common scheduling algorithm does not prioritize requests, but uses a first-in-first-out queue.

The scheduling algorithm is part of the core middleware, and the middleware developer determines whether or not it is possible to control or parameterize this scheduling algorithm. Trying to change this scheduling behavior beyond what has been made possible by the middleware developer requires changing the code, which would violate the common middleware requirement.

If we assume the distribution of the components over the different nodes as fixed, the only way to control QoS is by controlling the allocation of the (local) resources over the components located at a certain node. As an alternative, if we do not assume the distribution of the components to be fixed, we can control the achieved QoS by (re)distributing components to nodes, or over several nodes. Since the amount of available and needed resources differs per node, we can use this to control performance. And by distributing one component over several nodes, we increase the amount of resources available to this component. Controlling the QoS by controlling the distribution of the components, and thus the workload of these components, fits our approach because it does not require any resource-layer QoS mechanism. These two approaches are complementary.

We distinguish three ways to adapt the distribution of components:

- *initial placement* – create the component on a node that has enough resources available;
- *migration* – migrate a component to a node with more resources available;
- *replication* – replicate a component to increase the amount of resource that it has available.

We have chosen to apply our approach for QoS mechanisms to a mechanism that is based on dynamically changing the distribution of components over the nodes. This is the subject of Chapter 5 where we describe our Load Distribution mechanism.

3.3.2 Availability Mechanisms

We distinguish three QoS characteristics in the availability category (see Chapter 2):

- availability
- reliability
- maintainability.

We discuss two types of mechanisms that are possible within our approach and improve one or more of these characteristics. The first type of mechanisms we discuss can mask faults by creating redundancy by replicating components. This improves availability and reliability.

The second type of mechanism improves availability by allowing a distributed system to be reconfigured without taking it off-line. This includes on-the-fly migration of components to other nodes, and on-the-fly upgrading the functionality of a component. This improves maintainability, reliability and availability.

Replication Mechanisms

Creating redundancy can prevent a fault in some part of the system to lead to a failure of the system. This increases the uptime of a system (availability characteristic) and the time between failures (reliability characteristic).

In the case of a component-based application, redundancy translates to replicating components, and locating the replicas on different nodes. In case of node or network failures, the distributed system can continue functioning. Typically each replica is created from the same template. Hence replication has limited protection against software faults in the component. The replicas together form one component from the perspective of the rest of the distributed system. E.g., it is not visible for a client that a server component is actually implemented as a group of replicas. For each replica to be able to respond equally to a request of a client, their states need to be synchronized. Only in case of stateless components this is not required. The state synchronization requires exposure of the state, i.e., the component developer will have to provide the functionality to get and set the state. This is a violation of the encapsulation of state principle of components. Breaking encapsulation to expose the state is discussed in more detail in Chapter 4.

We distinguish two types of replication. With active replication each replica receives and processes all requests [Halteren99B, FTCORBA]. With passive replication only one replica, called the primary replica, receives and processes the requests, the other replicas are passive. Should a failure occur that affects the primary replica, one of the other replicas becomes active and assumes the role of primary replica [Man00B, FTCORBA].

At the moment of writing, commercial middleware has little or no facilities for replication, but literature does describe several approaches. See [Halteren99B] for a paper on multicasting request to a group of replicas. See [Man00B, Natarajan00] for papers on using a prototype CORBA fault-tolerance service called Doors to implement passive replication for CORBA Components. See [FTCORBA] for the Fault-Tolerant CORBA specification, and [Natarajan00] for a discussion on state synchronization and other issues involving fault-tolerant CORBA.

Dynamic Reconfiguration Mechanisms

System reconfiguration can cause a distributed system to become unavailable. Common reconfigurations include (i) reconfigurations that upgrade components to a newer version, and (ii) reconfigurations to (not) use certain nodes by migrating components to other nodes. Migrations of components can for example be needed in case a certain node has to be taken offline.

A reconfiguration can cause the system to become unavailable, i.e., it has to be taken offline. Even a reconfiguration on a single part of the distributed system may cause the distributed system as a whole to be unavailable, for example, because existing bindings between components may be broken, and those bindings may require a complete restart of the system to be re-established. In addition, partial restarts of the system can cause state inconsistencies which causes incorrect behavior of the system.

A dynamic reconfiguration mechanism allows runtime reconfiguration of a system, without causing the system to become unavailable. Dynamic reconfiguration is sometimes also referred to as *online upgrades* [Wegdam01A].

We have chosen to apply our approach for QoS Mechanisms in this PhD thesis to a dynamic reconfiguration mechanism. This is the subject of Chapter 4.

3.4 Using Reflection

As discussed in Section 3.2.1, a dynamic approach to QoS provisioning requires monitoring and control functionality in the application and/or middleware layer. We refer to this monitoring and control functionality as *QoS* instrumentation. The QoS instrumentation potentially violates our flexibility, time, expertise and common middleware requirements (as identified in Section 3.1).

QoS instrumentation

In this section, we discuss the usage of reflection techniques to cope with the issue of separating the QoS instrumentation from the (application) components and the core middleware.

QoS instrumentation is needed on different layers (application and middleware), and cannot be located in separate components or even in an additional layer. This makes QoS instrumentation a so-called *cross-cutting concern* [Bergmans01, Kiczales97]. The risk with cross-cutting concerns it that the code may end up being tangled with the code that deals with other concerns, violating the separation of concerns principle.

Cross-cutting concern

A technique to prevent tangled code is reflection. Reflection allows us to separate concerns by offering openness to the implementations details [Blair98A]. Reflection can prevent code that deals with different concerns to become tangled by handling certain concerns at the so-called meta level. Although reflection is often applied at a programming language level, this is not necessarily the case. In particular, reflection can be applied to middleware-based applications to reflect on the middleware and/or components. In addition, depending on the implementation, this does not require access to source code.

Reflection

Reflection and Aspect-Oriented Software Development

Before elaborating on reflection, we discuss a related programming technique that is getting more and more attention in the research community the last few years. This is Aspect-Oriented Programming (AOP) [Miller01], and the more general Aspect-Oriented Software Development (AOSD). Reflection can actually be used to implement AOP [Kiczales97, Elrad01].

Aspect-Oriented Programming

In AOP, a component is a piece of functionality that can be cleanly encapsulated in some programming language construct, e.g., an object or procedure [Kiczalis97]. Components tend to be units of the system's functional decomposition. An *aspect* cannot be cleanly separated in some programming language construct, but rather tends to be properties that affect the performance or semantics of components in systemic ways. An example of an aspect is synchronization of concurrent objects. Aspects are represented in an aspect language that is different from the component language. An aspect weaver combines the aspects and components into one program. This is done at the join points, which are those elements of the component language semantics that the aspects coordinate with. This weaving can be done at compile time or at run time.

Although we could view our QoS instrumentation as aspects, which can be weaved with the application components, we do not believe this be done in a straightforward manner using AOSD. Especially, we did not find any aspect language in which we could implement our QoS instrumentation, and that can be used with common middleware (especially not in 1999 and 2000 when this part of the research took place). We therefore dismiss AOSD as an option to develop our QoS instrumentation. However, with AOSD techniques becoming more mature, we do consider this a potential opportunity for the future. For example, more recent work by Filman and others [Filman02] combines AOP with CORBA, and uses instrumented stubs and skeletons to intercepts the request and reply (similar to middleware interceptors, as we will discuss below). By intercepting the requests and replies, Filman et al. are able to insert predefined aspects to the invocation path. An aspect language controls which aspects are inserted, and the aspect language is used to pass parameters to the aspects.

In the remainder of this section we give a more elaborate overview of reflection, discuss the usage of a special type of reflection called message reflection to implement QoS instrumentation, and identify and compare different ways to implement message reflection.

Overview of Reflection

The term reflection (or meta-level programming) is generally used for systems that have the ability to reason about them selves, using some kind of self-representation. This reasoning is done at a meta-level where certain aspects of the system are represented or reified as meta-objects. Reification is the process of transforming objects from a certain level to a meta-level, e.g., the transformation of method invocations to first-class objects [Plas99, Blair98A, Ferber89].

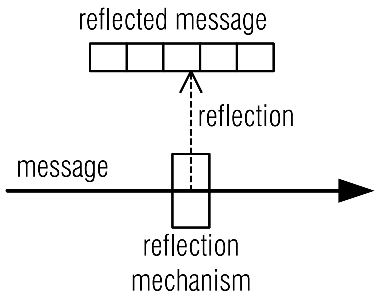
Reflection offers a principled, as opposed to ad hoc, means to expose certain implementation details [Andersen02]. It can be used for both inspection of a system, in which certain aspect of the system are revealed, or for adaptation in which the behavior is changed by modifying or adding a feature. Reflection is a technique to reach more flexibility and openness.

We believe that reflection is a promising way to implement QoS instrumentation, and allows us to fulfill the flexibility, time, expertise and common middleware requirements. The monitoring and adaptation functionality we need can be separated from the other concerns by locating them at the meta-level.

Reflective middleware is becoming an active area of research [Blair98A, Blair98B, Andersen01, Wang00, Halteren99B]. The focus of this work is on how to make middleware better configurable to be able to better adapt the middleware to environment or specific application requirements. Examples are changing the used protocol [Halteren99B, Wang00], or better support for streaming [Blair98B].

We distinguish two different types of reflection. Structural reflection is concerned with the structural dimensions of a system, such as classes, inheritance and instantiation relationships in an OO-system. Behavioral reflection is concerned with observing or changing the behavior of a system. A specific form of behavioral reflection is message reflection. Message reflection transforms a message to a meta-object. This meta-object can be read or modified to respectively obtain information or change the behavior of the system. This is depicted in *Figure 3-5*.

Figure 3-5 Message reflection



Current middleware technologies do not support reflection, or have very limited support for it [Blair98A]. Adding reflective capabilities to a system will have a great impact on both design and code, and we consider it very unlikely that this can be done without re-implementing it completely, which violates the common middleware requirement. However, what we can do without violating the common middleware requirement is to exploit the fact that components exchange message, and implement message reflection by intercepting these messages.

3.4.2 Message Reflection in Middleware

In this section, we list different methods that can be used to implement message reflection. The issue here is to intercept the messages going into or out of the components, and then be able to inspect and possibly alter the message. Criteria for a good method are that it is transparent for the application developer, the amount of overhead, the simplicity of instantiating and managing them.

Sniffing

A very straightforward method for intercepting messages is network sniffing. This is typically done by filtering out non-relevant TCP/IP messages, and parsing the relevant messages to extract the relevant information, effectively de-marshalling the requests. The obvious advantage of this method is that it is completely non-intrusive and transparent for the client, the server and the middleware. Disadvantages are that only messages actually passing through the network segment will be sniffed, excluding messages sent between clients and servers on the same host. A second problem is that this method is only practical on a network that uses broadcast technology, such as Ethernet. It would otherwise require a sniffer for each host which does not scale very well. A third limitation of this method is that it does not allow messages to be altered.

Instrumented Stubs and Skeletons

In most middleware technologies and for most distributed systems, stubs and skeletons are generated during the development or deployment phase by some tool provided by the middleware vendor. Stubs implement the proxy pattern [Gamma94], and marshal method invocations, such as parameters, at the client side into a standardized protocol format. Skeletons do the opposite by implementing the adapter pattern [Gamma94], and demarshal the method invocations at the server side. Typically stubs and skeletons are generated based on a description of the interfaces in some interface definition language. For example, in CORBA stubs and skeletons are generated by the IDL compiler. These stubs and skeletons can be instrumented to read or even alter messages that pass through them. Modified stubs are sometimes referred to as smart proxies [Wang00]. Filman et al. [Filman02] use instrumented stubs and skeletons to implement the join points for the Aspect Oriented Programming approach.

The main disadvantage of this method is that it is very dependent on the specific middleware implementation. For example in CORBA there is no standard for instrumented stubs or skeletons, although some ORBs have a proprietary way to do this. Another disadvantage is that messages that do not use stubs and skeletons, for example in the case of dynamic invocations in CORBA, are not intercepted.

Wrapping

Wrapping is a well-known pattern [Gamma94] to add functionality to an existing component or object. Wrapping can be used to intercept messages going to and from objects in a distributed application. Although dependent on the implementation technology, the main advantage of this method is that it is completely or at least mostly transparent to the server object. The problem is that the client has to send requests to the wrapper object instead of the actual object, which is especially difficult when object references are passed between clients. This problem requires a lot of administration and thus introduces a management issue. Also, it introduces a delay that can be unacceptable for certain applications. But in a system with a fixed number of objects on fixed locations this can be a good solution.

Inheritance and Delegation

At first glance, it might seem like a good idea to use implementation inheritance to add intercepting capabilities to a component. One can introduce a new class at the top of the inheritance tree that all other classes inherit from, or one can do the opposite and create a subclass of a component to do the intercepting. The first approach is not suitable for intercepting messages without requiring major changes to the middleware, since the instrumentation will not be in the invocation path. It can be used to intercept lifecycle events on an object. The second approach could be a solution, but introduces so-called inheritance anomalies [Bergmans96]. It is also quite intrusive to the application object and requires the usage of an object-oriented implementation language. Of course, inheritance is only be possible for object oriented programming languages that support inheritance, which is severe limitation. Delegation has similar disadvantages as inheritance, especially since it is intrusive to the application object.

Composition Filters

Composition filters [Bergmans96] is a modeling concept in which the actual object has explicit incoming and outgoing filters that can manipulate messages, e.g., to delay or to dispatch messages. It allows separation of concerns, and solves the problem of inheritance anomalies. It is a form of Aspect Orientated Programming (AOP) [Elrad01A], in which certain concerns are weaved together into a coherent program. Difference with other AOP approaches is that with composition filters the concerns are attached to messages, which allows the concerns to be modular extensions to the object. Because of this composition filters are less dependent on the implementation details of the object and more implementation language independent than other AOP approaches [Elrad01B].

The objects that composition filters encapsulate are (typically) programming language objects, but a component is typically implemented as a collection of objects, thus the granularity of the composition filters is too small. To be directly usable composition filters should have the granularity of a component, i.e., have middleware support. Unfortunately there is no support for it in common middleware technologies, and also limited support for composition filters in most implementation languages.

Middleware Interceptors

Middleware interceptors can intercept requests at defined points inside the middleware. All request are intercepted, including those between components that are co-located, i.e., requests between components in the same capsule. OMG standardized these type of interceptors for CORBA, and named them request interceptors [CORBA].

Middleware interceptors can intercept in- and outgoing requests on both the client and the server-side, resulting in a total of four interception points, see *Figure 3-6*.

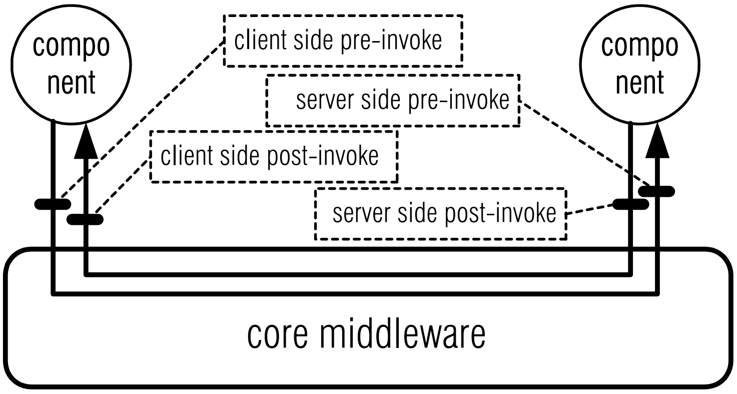


Figure 3-6 Middleware interceptors

The exact capabilities of the middleware interceptors depend on the middleware technology. For example in the case of CORBA a middleware interceptor can affect the outcome of a request by raising a system exception at any of the interception points, or directing a request to a different location. The target and parameters of a request can be inspected, but not altered. Several interceptor instances can be registered for one interception point, in which case they run in sequence. A request interceptor can inspect and alter implicit request parameters.

Other middleware technologies offer similar interception mechanisms as CORBA. For example COM+ [COM+] has interceptors that offers functionality that is comparable to the CORBA interceptors [Kath00].

Since the components are unaware of the middleware interceptors, this is a transparent solution. Depending on the middleware technology, no or very minimal code changes are required to instantiate the interceptors.

Operating System Interceptors

Operating System (OS) interceptors are positioned between the middleware and the OS-level interface to the network. Outgoing messages are intercepted after they leave the middleware, just before they enter the TCP/IP library. And incoming messages are intercepted just before they enter the middleware. This approach is used in Eternal [Narasimh99A]. The major benefit of this approach is that it is completely transparent to the component programmer and to the middleware. There are however several disadvantages. A major one is that since the intercepted messages are already marshaled, the messages have to be parsed (i.e., de-marshalling) to obtain request information (as is the case with sniffing). Besides this, the method depends on the usage of dynamically linked libraries, and is dependent on the OS and network. Last but not least, requests between components that reside in the same capsule or even on the same host cannot be intercepted, since they usually bypass the TCP/IP library.

Other

There are more interception mechanisms that are more on the programming language level. These are unsuitable because they are linked too much to the computational model of the programming language. The same argument is valid for Virtual Machine type of interception mechanisms such as the Java Virtual Machine Debugger Interface [Java], or the Java Virtual Machine Profiling Interface [Java]. The information is also too fined grained: we are only interested in messages that go in or out of the components, internal method calls are not relevant for our purpose. Filtering the relevant information is too complex and causes too much overhead.

Comparison

We consider middleware interceptors are best currently available technique to implement the message reflection functionality. Middleware interceptors can intercept incoming and outgoing requests and replies. The exact functionality depends on the used middleware technology, but each of the current major middleware technology has a similar feature. Benefits of middleware interceptors are that they can be added at deployment or even run-time, they also work for components located in one capsule or on one node, and they do not require changes in the middleware code.

This does not mean that the other interception mechanisms cannot be used. Depending on development environment, used middleware, access to source code and acceptable performance degradation, they may qualify as well or even can be preferred. For example, Filman et al. [Filman02] use instrumented stubs and skeletons, probably because this offers more flexibility than middleware interceptors. We will however use middleware interceptors in our design of QoS mechanisms.

3.5 Conclusions

Requirements

Our requirements for QoS mechanisms in component middleware can be separated in requirements from the perspectives of:
- the component developer that implements (application) components that use the QoS mechanism, and
- the middleware developer that implements the QoS mechanism.

The requirements from the perspective of the component developer are:
1. *Telematics* – the QoS mechanisms should be suitable for telematics systems, and in general for large-scale systems.
2. *Flexibility* – to allow third-party composition and enhance re-use, it should be possible to decide on the usage of QoS mechanisms and the QoS requirements as late as possible in the development cycle.
3. *Time* – the QoS mechanisms should require little development time to use them.
4. *Expertise* – the QoS mechanisms should require little expertise on the complexities of enforcing QoS from the component developer.
5. *Heterogeneity* – it should be possible to use the QoS mechanisms on a large variety of different types of node and networks.

From the perspective of the developer of the QoS mechanisms the requirements are:
6. *Generality* – the QoS mechanisms should be application independent, and thus be useable for a wide range of applications.
7. *Common middleware* – the QoS mechanisms should be useable together with common middleware technologies.

Dynamic Approach
A static approach towards QoS provisioning is based on calculating the required resources needed for obtaining a certain QoS, and reserving these resources for the lifetime of the application or session. A dynamic approach is based on monitoring the achieved QoS during run-time, and adapting the application behavior or resource allocation if the achieved QoS does not fulfill the QoS requirements.

We select a dynamic approach for our QoS mechanisms. The reasons for this are that a static approach is not suitable for the telematics domain because of the difficulty to predict usage, needed resources per usage, available resources due to the sharing of resources and inherent unpredictability of failures. In addition, a dynamic approach is more efficient with resources since resources are only allocated to an application when it actually uses them (contrary to allocated the resources for the whole lifetime of an application).

Separation of Concerns

Central in our approach in our aim for a strict separation of concerns, where we want to minimize the involvement of the component developer in the complexities of QoS provisioning. The role of the component developer is ideally limited to providing QoS requirements. We proposed to extend the distribution mechanisms as they are now provided by middleware technologies with new QoS transparencies. These QoS transparencies hide the complexities of the QoS provisioning from the component developer.

Middleware-layer-internal QoS Mechanisms

We can divide middleware-layer QoS mechanisms into mapping and middleware-layer-internal QoS mechanisms based on whether or not they rely on resource-layer QoS mechanisms. Mapping QoS mechanisms are middleware-layer QoS mechanisms that map to resource-layer QoS mechanisms. Middleware-layer-internal QoS mechanisms are middleware-layer QoS mechanisms that do not use resource-layer QoS mechanisms, and instead enhance the QoS by using the functions that the middleware provides.

Middleware-layer-internal mechanisms can be used in a wider range of environments than mapping mechanisms because they do not require the support of specific resource-layer QoS mechanisms. In addition, middleware-layer-internal mechanisms can enhance the QoS beyond what the resource layer provides, while mapping mechanisms are by definition limited to whatever QoS mechanisms the resource layer provides.

Dynamic Reconfiguration and Load Distribution

There are different types of QoS mechanisms possible within our approach. We choose to apply our approach to a dynamic reconfiguration mechanism (which improves availability), and to a load distribution mechanism (which improves performance). These mechanisms will be the focus on the remainder of this thesis.

Message Reflection

Message reflection can help us with one of the main challenges of our research: how to implement the monitoring and adaptation functionality that is required for our QoS mechanisms and still have a strict separation of concerns. Based on an evaluation and comparison of different ways to implement message reflection, we consider middleware interceptors as the best way that is currently available to implement message reflection.

Chapter 4

Dynamic Reconfiguration

This chapter[3] describes a QoS mechanism that improves the availability characteristics of a distributed system by making it possible to create, upgrade, migrate or remove components at run-time.

This chapter is structured as follows: Section 4.1 presents our model of dynamic reconfiguration, a new classification of approaches to dynamic reconfiguration and definitions of concepts and terminology in the area of dynamic reconfiguration. The model, classification, concepts and terminology presented in Section 4.1 are used in Section 4.2 to present, evaluate and compare the state-of-the-art in the area of dynamic reconfiguration, both approaches that are middleware based and approaches that are not middleware based. Section 4.3 discusses our middleware-based mechanism for dynamic reconfiguration, and compares our mechanism to the related work. Section 4.4 presents a high level design of our Dynamic Reconfiguration Service, which implements our mechanism. Section 4.5 presents the major conclusions of this chapter.

A description of the prototype of the Dynamic Reconfiguration Service can be found in Chapter 6.

4.1 A Model of Dynamic Reconfiguration

This section develops a model of dynamic reconfiguration that we adopt in this theses and a classification of approaches to dynamic reconfiguration. In addition, we give an overview of important concepts and terminology that are used in the area of dynamic reconfiguration. We discuss the importance of consistency preservation, and the three consistency preservation requirements: structural integrity, mutually consistent state and application

[3] Parts of this chapter have been published in the papers [Almeida01A], [Almeida01B] and [Wegdam03A], which are co-authored by the author of this PhD thesis, the Lucent Technologies' response [Wegdam01A] to OMG's *Request For Information on Online Upgrades* and in a master thesis that was supervised by the author of this PhD thesis [Almeida01C].

state invariants. We conclude this section by discussing the impact of reconfiguration on execution.

4.1.1 Introduction

The aim of dynamic reconfiguration [Bidan98, Bloom93, Endler94, Hofmeister93, Kramer85, Kramer90, Goudarzi99, Oreizy98, Wermel99] is to allow a system to evolve incrementally from one configuration to another at run-time, as opposed to at design-time, while introducing little (or ideally no) impact on the system's execution. In this way, systems do not have to be taken off-line, rebooted or restarted to accommodate changes.

The reconfiguration will have some impact on execution. Typically at least part of the system will be suspended during the reconfiguration, thus potentially violating some performance requirements. It depends on the QoS requirements and the extent of the impact whether this will be considered a disruption. In case the impact of a reconfiguration is such that it is not considered a disruption of the system, dynamic reconfiguration improves the reliability (mean time between failures or disruptions) and availability (uptime) QoS characteristics (see Chapter 2 for definition of these QoS characteristics). And even if the impact on execution of a reconfiguration is such that that this is considered a disruption, this disruption will typically be much shorter than a complete restart of the system. So also in this case dynamic reconfiguration improves availability and maintainability (time to repair) QoS characteristics. We will discuss the impact on execution of a dynamic reconfiguration more elaborately later this chapter.

4.1.2 Process and Activities Overview

The purpose of *dynamic reconfiguration* is to make a system evolve incrementally from its current configuration to another configuration without disrupting the system. A *system configuration* is defined as a set of software entities, and how they are related to each other. The definition of entity depends on the level of granularity of reconfiguration. Examples of entities include objects, groups of objects, components, groups of components, sub-systems, modules, bindings and groups of bindings. Reconfiguration is specified in terms of entities and operations on these entities. Typical operations on entities are replacement, migration, creation and removal. Dynamic reconfiguration should introduce as little impact as possible (ideally no impact at all) on the system execution.

Figure 4-1 depicts our dynamic reconfiguration model. This model is based on [Kramer85, Kramer90].

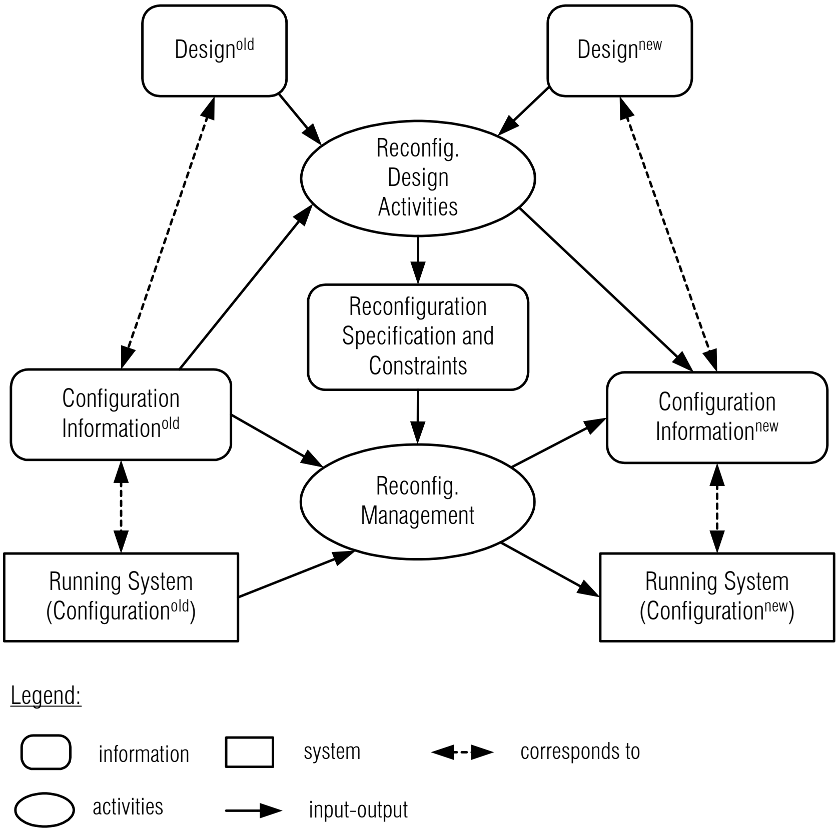

Design^old
Design^new
Reconfig. Design Activities
Reconfiguration Specification and Constraints
Configuration Information^old
Configuration Information^new
Reconfig. Management
Running System (Configuration^old)
Running System (Configuration^new)
Legend:
information
system
corresponds to
activities
input-output

Figure 4-1 Model of Dynamic Reconfiguration

In this model, *reconfiguration design activities* are the activities that prepare the reconfiguration. The reconfiguration design activities have as input the old design, the new design and configuration information on the current system. *Configuration information* refers to the relationship between entities. The reconfiguration design activities produce updated configuration information, and the specification of well-defined changes and constraints that have to be preserved during reconfiguration. Changes are specified in terms of *entities* and *operations* on these entities, and are applied under the control of reconfiguration management functionality. Reconfiguration constraints are predicates on the reconfiguration process that restrict its execution, e.g., "the reconfiguration process must be completed within 10s", or "entity A should be available during the whole reconfiguration process".

Reconfiguration management functionality [Kramer85, Goudarzi99, Oreizy98] controls the reconfiguration process of a distributed system. This functionality makes the system evolve from its current configuration to a new configuration. It has as input the current system and its configuration information, and the reconfiguration specification and reconfiguration constraints. It produces the new system, and updated configuration information.

The reconfiguration management functionality must guarantee that (i) specified changes are eventually applied to a system, (ii) a (useful) correct system is obtained, and (iii) reconfiguration constraints are satisfied.

Reconfiguration Design Activities

Reconfiguration (or change) design activities are part of the design activities that are executed during the lifetime of a system, and relate to a specific reconfiguration. These activities succeed system deployment, in case of unforeseen changes, and precede the application of a reconfiguration to a system. They are performed by reconfiguration (or change) designers.

Reconfiguration designers make use of the initial system configuration and the new configuration, identifying modifications introduced, to produce a well-defined set of changes to be applied to a system.

Changes are specified in terms of entities and operations on these entities. The definition of entity depends on the level of granularity of reconfiguration. Examples of entities include objects, groups of objects, components, groups of components, sub-systems, modules, bindings and groups of bindings. Examples of operations on entities are replacement, migration, creation and removal.

The procedures for obtaining a new system configuration are beyond the scope of this work. These procedures are performed subsequently to design activities and may include transformations on the initial system, re-specification, re-design and re-implementation, re-validation, re-test of parts of the system, acquisition and integration of new system parts, etc.

4.1.3 Correctness

Operating systems, middleware platforms and programming languages have mechanisms that facilitate system evolution, by allowing modules to be located, loaded and executed during run-time. However, these mechanisms normally do not ensure correctness, or desired properties of run-time change. Therefore, the sole use of these mechanisms to perform reconfiguration is error-prone [Oreizy98].

Performing reconfiguration on a running system is an intrusive process [Goudarzi99]. Reconfiguration may interfere with ongoing interactions between entities. Reconfiguration management must assure that system parts that interact with entities under reconfiguration do not fail because of the reconfiguration.

Preservation of system consistency is a major reconfiguration requirement. A system can become useless in case the preservation of consistency is ignored. The system under reconfiguration must be left in a "correct" state after reconfiguration. In order to support the notion of correctness of a distributed system, three aspects of *correctness* requirements are identified [Goudarzi99]. A system is said to be correct state if:

Correctness

1. The system satisfies its *structural integrity* requirements,
2. The entities in the system are in *mutually consistent states*, and
3. The *application state invariants* hold.

A resulting running system S_{i+1} is said to be a correct incremental evolution of a running system S_i, if S_{i+1} is in correct state, and if the behavior of the affected entities complies with the behavior expected by the unaffected system parts in case the reconfiguration had not taken place. Each aspect of the correctness notion is addressed in the remainder of this section.

Structural Integrity

Structural integrity requirements constrain the structure of a system, i.e., they constrain how entities are related [Gouradzi99].

Reconfiguration may affect the structural integrity of the whole system, so that corrective measures must be taken. For example, let us consider the replacement of one component by a new version of this component in a component-middleware-based system. Clients of the component being replaced should be capable of invoking the operations of this component during reconfiguration and after reconfiguration has taken place. This implies that two conditions on the structural integrity of the system must hold: (i) the new version of the component must satisfy the interface definitions of the original component, and (ii) the clients should have a valid reference to the new version of the component.

Mutually Consistent States

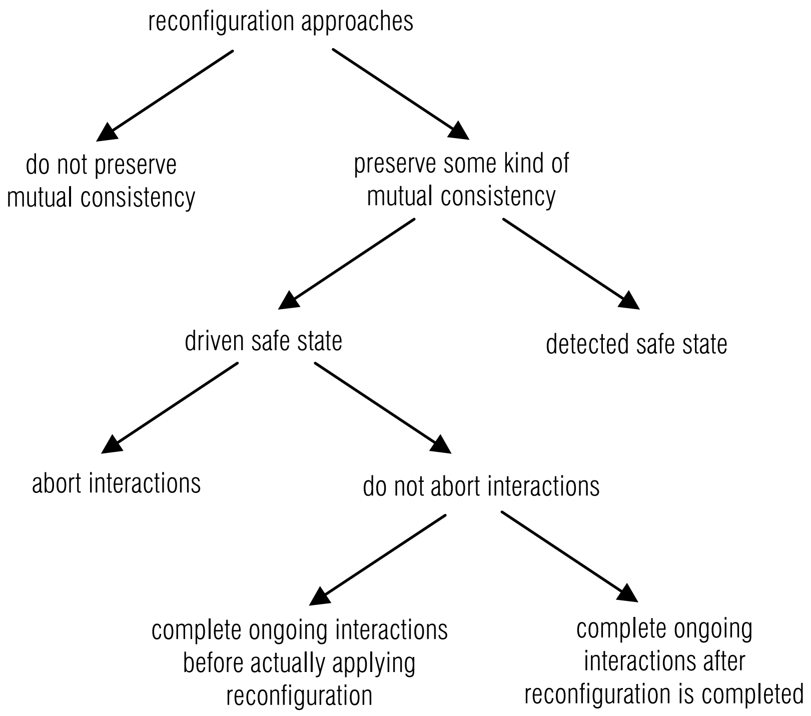
Entities in a distributed system need to be in mutually consistent states if they are to interact successfully with each other. Entities are said to be in *mutually consistent states*, if each interaction between them, on completion, results in a transition between well-defined and consistent states for the parts involved [Goudarzi99]. Interactions are the only means by which entities can affect each other's state.

For example, in a system with two components, component A invokes an operation on B. Components A and B are said to be in mutually consistent states if A and B have the same assumptions on the result of the interactions between them. To be more specific, either both of them perceive that an invocation has occurred successfully, or both of them perceive that the invocation has failed. Suppose the change manager decides to replace B by B' after A initiated an operation invocation on B. For the resulting system to be in a consistent state, either (i) the invocation is aborted, A is informed and synchronization is maintained; or (ii) B receives the request, finishes processing it and sends the response, and then is replaced by B'; or, (iii) B is replaced by B', and B' has to honor the invocation, by processing the request and sending a response to A. In case none of these alternatives occur, A might be waiting forever for a response.

Reconfiguration-safe state

Reconfiguration approaches provide mechanisms to transform systems with entities in mutually consistent states into resulting systems that maintain this mutual consistency. This is done by defining a *reconfiguration-safe state* (or shortly safe state) in which reconfiguration can be applied while maintaining mutual consistency. *Figure 4-2* shows a classification of reconfiguration approaches according to their choices regarding the preservation of mutual consistency.

Figure 4-2 Classification of reconfiguration approaches


reconfiguration approaches
do not preserve mutual consistency
preserve some kind of mutual consistency
driven safe state
detected safe state
abort interactions
do not abort interactions
complete ongoing interactions before actually applying reconfiguration
complete ongoing interactions after reconfiguration is completed

In this classification, approaches that preserve some form of mutual consistency fall into two categories: the ones that reach the reconfiguration-safe state by observing the system execution, and the ones that reach the reconfiguration-safe state by driving the system to it. In the former case, the reachability of the safe state depends on the behavior of the application. For systems in which entities may interact continuously because of parallel interactions that are interleaved, there is no guarantee that reconfiguration will ever take place. If at all times there are interactions in progress, reconfiguration is postponed indefinitely. In case the system is driven to a safe state, it is the role of the reconfiguration algorithm to guarantee the reachability of the safe state.

Existing approaches that work with a driven safe state fall into two major categories [Goudarzi99]: those in which during reconfiguration interactions are aborted and that rely on entities to recover from abortions, and those which avoid interactions to be aborted. Mechanisms based on interaction abortion (e.g., [Bloom93]) require the application developer to provide rollback mechanisms to recover from abortions without proceeding to errors. Therefore, the range of applications to which these mechanisms can be used is quite limited.

Abortion-based approaches

Mechanisms that do not abort interactions are designed to assure that interactions in progress are eventually completed, either before reconfiguration has started or after reconfiguration has finished.
In case of an approach in which ongoing interactions are interrupted (suspended) and completed (resumed) when reconfiguration has finished, the application developer has to implement functionality to restore the control state of the reconfigured entities, allowing the interrupted interactions to continue after reconfiguration. This control state typically includes the state of the invocation stack, program counter or thread context information. This information is closely tied to specific characteristics of the implementation code, and it is typically language- and operating system-dependent. The mapping of the control state from one implementation to the implementation of the new version would require deep knowledge of both implementations and would hardly be manageable by the reconfiguration designer. Therefore most approaches to reconfiguration do not consider this alternative. An exception is [Hofmeister93].

In this chapter, we propose an approach that drives the system to a safe state without aborting interactions and that allows ongoing interactions to complete before reconfiguration is applied. This mechanism is discussed in Section 4.3.

Application-State Invariants

Application-state invariants are predicates involving the state (of a subset) of the entities in a system. The preservation of safety and liveness properties of a system depends on the satisfaction of these invariants [Goudarzi99].

For example, let us consider a component that generates unique identifiers. An application-state invariant could be “all identifiers generated by the component are unique within the lifetime of the system”. In order to preserve this invariant, the new version of the component must be initialized in a state that prevents it from generating identifiers that have been already used by the original version. So, either (i) the set of all used identifiers is provided to the new version of the component, or (ii) the last used identifier is provided to the new version of the component. The latter alternative would require knowledge of the assignment mechanism used by the original version.

If dynamic reconfiguration is to be useful in a broad range of scenarios, it ought to provide mechanisms to allow the re-establishment of application-state invariants.

Most existing reconfiguration approaches rely on embedding the extra functionality for dealing with invalidated invariants into reconfigurable entities [Goudarzi99]. In this way, the responsibility to re-establish application invariants is solely delegated to application entities, which must determine what course of actions is needed to re-establish application invariants. For example, in Conic [Kramer90], application designers are required to supply modules with embedded routines (initialization and finalization) that are called whenever a reconfiguration operation is executed. The complexity of these routines depends largely on the nature of the application.

As pointed out in [Goudarzi99], this approach has serious drawbacks. Due to the generality of possible changes to a system, individual entities are rarely in a position to determine the course of actions to re-establish application-state invariants. This is especially true when, as is often the case, invariants are expressed over the combined state of a number of entities of the system.

Application entities that are developed to re-establish application-state invariants are likely to lose their potential generality, since they embed configuration specific concerns that prevent them from being used in other configurations. This is hardly acceptable since it reduces the potential for re-use and third-party composition.

Since embedding the necessary functionality to deal with invalidated invariants into application entities is undesirable, the support platform should provide mechanisms for change designers to specify how to re-establish application-state invariants.

[Goudarzi99] proposes a scheme whereby invalidated invariants can be identified and re-established by the change designer with little assistance from the application developer. This scheme consists of requiring reconfigurable entities to provide general-purpose state access-methods that can be invoked by a third party to query or adjust the state of entities. These are called state-access methods, and would be invoked by the change designer to query and alter a selected subset of an entity's internal state at runtime. The particular subset of the state that is exposed by these access-methods is decided upon by the application designer. In general, entities should provide "get" and "set" methods for state variables that control synchronization and computational behavior of the entity. One might argue that this scheme breaks encapsulation, as it allows external access to a component's internal state. Nevertheless, some form of introspection is necessary anyway for the manipulation of run-time aspects of an entity.

The nature of the safe state, as discussed in the beginning of this section, should be such that in the safe state the invocation of state-access methods yields meaningful results. Thus, a reconfigurable entity in a reconfiguration-safe state must have a consistent, self-contained state that can be accessed from outside the entity.

4.1.4 Impact on Execution

Reconfiguration is an intrusive process, since during reconfiguration, some system entities may temporarily become partially or totally unavailable, which can affect the performance of the system as a whole. Determining to what extent a system is affected during reconfiguration is relevant to assess the risks and costs in performing dynamic reconfiguration. If the system during reconfiguration fails to satisfy some QoS requirements (e.g., hard response times), it may not be feasible to reconfigure during run-time. For instance, dynamic reconfiguration may be shown to be unacceptable due to safety reasons. This may be the case for process control, where a failure to perform a critical activity within a bounded time can put people's lives in danger.

The quantification of the impact of reconfiguration on system execution is not trivial. Some reconfiguration approaches [Kramer85, Goudarzi99] quantify the impact on system execution as proportional to the number of system entities affected by reconfiguration. These entities become idle or partially idle due to reconfiguration and would otherwise execute normally. In [Bidan98] a more fine grained quantification is proposed in which impact is said to be minimal if the reconfiguration affects the smallest possible set of execution threads in system objects. In [Wermel99], it is argued that more attention should be given to the period of time during which system entities are affected by reconfiguration.

Application characteristics are important when evaluating the impact of reconfiguration on execution. The impact of reconfiguration, and which reconfiguration algorithm would result in the least impact, cannot be evaluated if we do not consider, for example, the level of coupling between system parts and the duration of the interactions between these parts.

In order to better understand the implications of application characteristics to system execution during reconfiguration, let us consider an application where interactions might take up to some hours to complete. Further, let us consider the replacement of an entity that has just initiated an interaction, using a reconfiguration approach based on a driven safe state. If we choose for an approach that allows on-going interactions to complete before reconfiguration, the reconfiguration will have a large impact on system execution, as it might take hours before the safe state is reached. During this time period, the affected system parts may not initiate new interactions, which might prevent the rest of the system from functioning. In contrast, if we choose an approach that aborts interactions, the reconfiguration time can be reduced drastically.

Ultimately, the maximum acceptable level of disturbance on the QoS during reconfiguration is determined by the QoS requirement of the application.

4.2 State-of-the-Art in Dynamic Reconfiguration

This section describes some available approaches to dynamic reconfiguration reported in the literature, extending the survey presented in [Goudarzi99]. The selected approaches preserve mutual consistency without aborting interactions and strive to minimize impact on system's execution.

For each approach we present its overall considerations on reconfigurable distributed systems, as well as the way it structures reconfiguration functionality, and its specific mechanisms to guarantee correctness of the resulting system. We give special attention to the level of transparency for the application developer, including the configuration information required to allow reconfiguration.

We do not cover the following approaches further in this section, but we do mention them here for completeness:

- [Rodriguez99] describes how the interpreted language Lua can be used in combination with CORBA Dynamic Invocation Interface and the Dynamic Skeleton Interface as a substitute to declarative configuration files to control the linking between clients and servers. The actual CORBA objects can be programmed in more conventional compiled languages. Also minor changes in the interfaces are possible without recompilation, e.g. to change a parameter from short to long. None of the consistency guarantees are discussed, and this thesis does not cover run-time reconfiguration. We will therefore not consider it further.
- [Hofmeister93] proposes an approach in which the state of a module that has to be reconfigured can be captured even if the module is not in a quiescent state, i.e. it is possible to have active threads and ongoing interactions. Because of the required access to the low-level control

state (e.g., stack and heap), and the intrusiveness to the application code, we do not consider this approach further.

4.2.1 Kramer and Magee

The early work of Kramer and Magee [Kramer85, Kramer90] has influenced the subsequent works of many others on dynamic reconfiguration. The concepts and terminology presented in Section 4.1 stem mostly from their work. Kramer and Magee promote a strict separation between the structural description of a system and the description of individual nodes. The first realization of their approach could be seen in the Conic environment [Kramer85], and led to the development of the approach called Configuration Programming and a configuration language named Darwin [Magee95].

In the *Configuration Programming approach*, a system is seen as a directed graph consisting of nodes and connections between the nodes. A node is defined here as a processing entity. The model assumes at most one connection between any pair of nodes. Nodes can only affect each other states via transactions. A transaction is defined in this approach as an instance of information exchange between two and only two nodes, initiated by one of the nodes, and consisting of a sequence of one or more message exchanges between the two connected nodes. The model also assumes that transactions complete in bounded time and that the initiator of a transaction is aware of its completion. *Figure 4-3* shows an example of a simple system, in which nodes A1, A2 and A3 are able to initiate transactions on a node B.

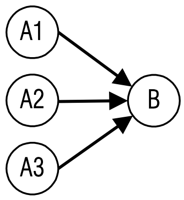


Figure 4-3 A simple system

In this approach, a change is described in terms of modifications to the structure (configuration) of the application system. Changes take the form of node creation and deletion, and connection establishment and removal, and are applied by a Configuration Manager.

Reconfiguration-Safe State

This approach has been the first to propose an avoidance-based mechanism to ensure that reconfigurations do not result in mutually inconsistent node states.

Kramer and Magee's approach uses the description of the changes and the current system configuration:

1. to identify the set of nodes whose activities must be restricted if reconfiguration is to proceed without leaving them in mutually inconsistent states, and;

2. to instruct these nodes to restrict their behavior by becoming passive, so that the reconfiguration safe state is brought about over the affected nodes.

Dependent and independent transactions

In this approach, node interactions are bounded transactions which are assumed to be the only means through which connected node can affect each other's states. Both parties involved in a transaction are informed of its completion. A transaction t is said to be *dependent* on the *consequent transactions* t1, t2,… tn (written t/t1t2..tn), if t can complete only after t1, t2,… tn complete, and *independent* otherwise. This approach supports reconfiguration in systems with independent and dependent transactions.

Reachability of the Safe State

The safe state for reconfiguration is reachable in finite time. This is discussed below for systems with only independent transactions and then generalized for systems with dependent transactions.

Passive state

For systems with independent transactions a node is said to be in the *passive state* if it:

a) continues to accept and service transactions, but
b) does not initiate new transactions, and
c) any transactions it has already initiated have completed.

A node reaches the passive state by refraining from starting new transactions and waiting for all the transactions it has started to terminate. A node is said to be *passive* if it is in a passive state. Passive nodes are not necessarily in a reconfiguration safe state, since they continue to accept and service transactions. Therefore, the notion of quiescence is relevant. A node is said to be *quiescent* if it is passive, and

Quiescent

d) it is not currently engaged in servicing any transactions (self initiated or otherwise), and
e) no transactions have been or will be initiated by other nodes which require service from this node.

The passive state can be brought about by nodes unilaterally. The quiescent state, in contrast, can only be brought about by nodes in cooperation with other nodes in the system. A node N becomes quiescent if and only if all nodes in its passive set PS, denoted PS(N) are in the passive state. For systems with only independent transactions, the membership of PS(N) is as follows:

a) the node N, and
b) all nodes that can directly initiate transactions on N, i.e., all nodes directly connected to N.

If all nodes in PS(N) are passive, N as well as all nodes that can initiate transactions on N are passive. Therefore, all transactions involving N are complete and new transactions will not be initiated, satisfying the quiescence requirements d) and e). As the approach assumes transactions to complete in bounded time, it follows that quiescence is reachable within bounded time.

For systems with dependent transactions the situation is more complicated and the definition of passive and PS(N) need to be amended to allow for the initiation and service of consequent transactions. Consider the system depicted in *Figure 4-4*, consisting of three nodes N1, N2 and N3. Suppose that N3 is in the passive state and N1 has initiated transaction a. In this situation, transaction a cannot complete because it depends on consequent transaction b. Transaction b cannot complete because it depends on a consequent transaction c, which N3 cannot initiate since it is passive. This means that neither N1 nor N2 will be able to move into the passive state if we would apply the algorithm as suggested above.

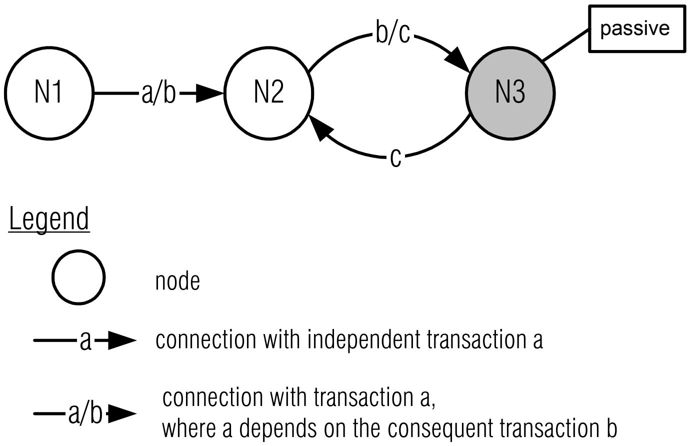

N1
a/b
N2
b/c
c
N3
passive
Legend
node
a
connection with independent transaction a
a/b
connection with transaction a,
where a depends on the consequent transaction b

Figure 4-4 A system with dependent transactions.

To ensure the reachability of the passive state and consequently the reachability of the quiescent state for systems with dependent transactions, the requirements of the passive state have been modified as follows. For a system with dependent transactions a node is said to be in the passive state if it:

a) continues to accept and service transactions and initiate consequent transactions, but
b) does not initiate new (non-consequent) transactions, and
c) any (non-consequent) transactions it has already initiated have terminated.

The set of passive nodes is extended to include all the nodes which are capable of initiating transactions indirectly on N. The enlarged passive set for a node N is called EPS(N) and is defined as follows:

a) all nodes in PS(N) are in EPS(N), and
b) all nodes that can initiate dependent transactions that result in consequent transactions on N are in EPS(N).

This extension guarantees that node N reaches a quiescent state in finite time.

Reconfiguration Rules

So far, we have discussed how nodes can reach quiescent states. Nevertheless, we have not discussed which set of nodes should be in the quiescent state for reconfiguration. In Kramer and Magee's approach, reconfiguration actions are node deletion, node linking and unlinking, and node creation. For each of these actions, reconfiguration rules in *Table 4-1* apply:

Table 4-1
Reconfiguration rules and justification.

Actions	Rule and justification
Node removal	*Rule* - The node targeted for removal must be quiescent and isolated, where isolated means that no connections are directed to it from other nodes or from it to other nodes.
	Justification - An isolated node cannot affect the system and therefore can be independently removed.
Node linking and unlinking	*Rule* - The node N from which the connection is directed must be in the quiescent state.
	Justification - Quiescence of the initiator node ensures that its state is consistent and frozen with respect to that connection, and all transactions involving this node are complete.
Node creation	*Rule* - The node should be quiescent.
	Justification - Trivially true, a newly created node is initially isolated and can neither respond to nor initiate transactions.

Using these rules it is possible to obtain the order in which nodes should be made passive, removed, created, connected and disconnected. Kramer and Magee also define an algorithm that allows multiple reconfiguration operations to be conducted simultaneously (see [Kramer90]).

Some criticisms to Kramer and Magee's approach are:

- It places a heavy burden on the application programmer who must write all nodes of the system such that they respond correctly to the command to drive to a passive state [Bidan98, Goudarzi99]. Kramer and Magee's approach thus requires substantial effort from the application developer to make the system reconfigurable, and it also requires expertise from the application developer. This respectively violates the time and expertise requirements, as identified Chapter 3 (generic requirements for QoS Mechanisms).
- Since all entities capable of initiating a transaction directly or indirectly with an affected entity have to be passive to reach the safe state, even small reconfigurations involving a few nodes result in substantial disruptions to the system [Goudarzi99, Wermel99].
- The re-establishment of application invariants is done through routines embedded in nodes [Goudarzi99].

4.2.2 Goudarzi

In [Goudarzi99], Moazami-Goudarzi proposes a framework that identifies the basic elements of a change management subsystem and establishes a separation between the responsibilities of the objects that implement this subsystem. The framework consists of a Reconfiguration Manager, a Reconfiguration Database, the Consistency Manager and a number of runtime hooks in the application.

The Reconfiguration Manager selects and executes reconfiguration scripts upon the arrival of triggering messages from an Event Composition Service. The Reconfiguration Manager coordinates the execution of the scripts with the Consistency Manager and Configuration Database, such that reconfiguration operations do not interfere with each other and leave objects in mutually consistent states.

The Consistency Manager encapsulates the safety mechanism necessary to ensure that objects are left in mutually consistent states after reconfiguration. Thus reconfigurations only proceed after the Consistency Manager has been consulted and has signaled that they can proceed safely.

The Configuration Database maintains and affects changes to the system configuration. It exports an interface that can be used to query and modify the system configuration. Interactions with the Configuration Database are transaction-based and are performed through an internal concurrency control module that coordinates concurrent access to the system configuration.

The Event Composition Service evaluates the triggering conditions written by the change designers and generates messages that trigger the execution of the reconfiguration scripts. In this framework, reconfiguration scripts are written in a reconfiguration language.

The re-establishment of the application invariants is controlled from within the reconfiguration program, with the aid of specialized runtime hooks.

Preserving Consistency

Moazami-Goudarzi's approach presents an alternative to Kramer and Magee's approach to reach a reconfiguration safe state. It assumes that objects in the system do not interleave transactions, i.e., while a transaction is in progress, an object does not participate in any new one. In this way, it is possible to drive an object to a quiescent state by blocking its execution when no transactions are being serviced. As in Kramer and Magee's approach, for an object to block within finite time (therefore reaching quiescence), transactions are assumed to complete within finite time.

The basic algorithm is to request objects in the quiescent set (called BSet, short for blocking set) to block their execution. Consider that an object Q is to be driven to the quiescent state. Since some of the objects that depend on Q may also have to block, Q must temporarily unblock to service some requests. However, the mechanism must guarantee that at some point no more such requests arrive and Q remains blocked. Therefore, a blocked object should be selective when serving transactions. A blocked object cannot process just any incoming transaction, since the transaction might come from an object that is not affected by the reconfiguration and thus is not blocked and allowed to initiate a new transaction any time. If all incoming transactions would be processed, a blocked object would unblock unpredictably and the safe state needed for reconfiguration to begin would never be reached. At least the transactions initiated by other BSet members will have to be serviced in order for them to become blocked. However, not every request from a non-BSet member can be ignored, since this might indirectly prevent another object in the BSet from blocking.

Figure 4-5 gives an example of a situation in which blocked object N1 does not accept incoming transaction c from non-blocked object N4, but has to accept an incoming transaction b from non-blocked object N3. Transaction b has to be processed because transaction a depends on it, and without completion of a affected object N2 will never reach a quiescent state.

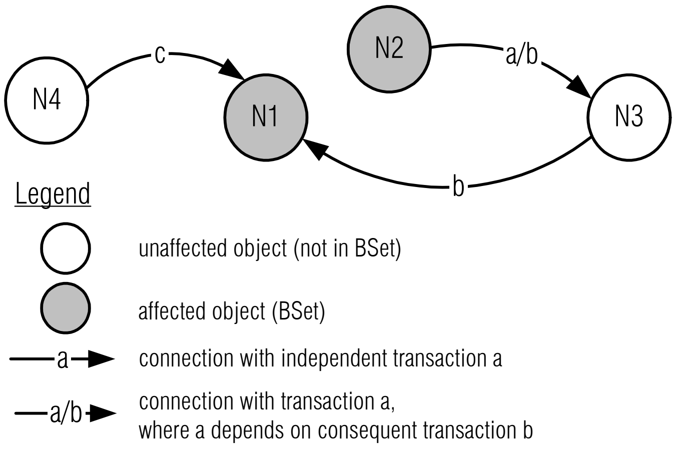

N4
c
N1
N2
a/b
N3
b
Legend
unaffected object (not in BSet)
affected object (BSet)
a
connection with independent transaction a
a/b
connection with transaction a, where a depends on consequent transaction b

Figure 4-5 Situation in which BSet members have to accept transactions from non-BSet members

The BSet grows dynamically with outgoing transactions. When an object gets an incoming transaction from a BSet member, it becomes a member too, and only transactions from BSet members are attended; all other are queued and serviced after reconfiguration. In the above example, N3 thus becomes a member of BSet when it gets a transaction from N2.

A distinction is made between members of the original BSet and members of the extended BSet. Members of the original BSet are affected directly by reconfiguration. Members of the extended BSet are those that have blocked in order to let the members of the original BSet get blocked. When all the members of the original BSet are blocked, the objects in the extended BSet can be unblocked. The BSet thus first grows and then shrinks.

This alternative addresses some of the criticisms to Kramer and Magee's work mentioned before. Nevertheless, the class of distributed systems to which this alternative can be applied is much more limited than in the case of Kramer and Magee, since objects in this approach cannot treat more than one transaction simultaneously. In component-middleware-based applications, such as CORBA-based applications, components can be multi-threaded, and it is possible to have re-entrant invocations. A re-entrant invocation is a special type of nested (or consequent) invocation that invokes a component that is in its invocation path. *Figure 4-6* shows the simplest case of re-entrance in which the nested invocation b has to be processed before invocation a can finish. Both multi-threading and re-entrance cannot be supported using Goudarzi's approach.

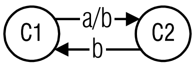


Figure 4-6 Example of re-entrance

4.2.3 Bidan et al.

In [Bidan98], the implementation of a reconfiguration service in CORBA is considered. A distributed system in this case consists of a number of objects that communicate over an ORB. The reconfigurable entity is a CORBA object and the configuration information consists of a directed graph of objects connected through links. Objects A and B are said to be linked if A can potentially initiate a CORBA invocation on a target object B. Links are therefore similar to connections in Kramer and Magee's approach.

This approach offers node consistency, i.e., it is primarily concerned with preserving mutual consistent states, refraining from addressing application consistency. More specifically, they provide Remote Procedure Call (RPC)-integrity, which is defined as "all RPCs initiated will be completed before the changes are effected." By providing only node consistency they do not address application state invariants and state transfer issues.

The reconfiguration service is designed for CORBA applications with multi-threaded objects, following the thread-per-request execution model, and extends the LifeCycle service [CORBA] to support dynamic reconfiguration of a CORBA application. It provides the primitives create and remove to respectively create and remove objects, and the primitives `link`, `unlink`, `transferLink` and `transferState` to respectively create and destroy a link, transfer the requests pending on a passivated link to another existing link, and to transfer the state from one object to another.

Reconfigurable objects should implement functionality to passivate a link, i.e., to block the thread that may use the specific link.

Preserving Mutual Consistent States

In [Bidan98], the algorithm to guarantee mutual consistent states works at a finer granularity level than the approaches previously presented. This approach considers the passivation of links instead of quiescence, passivation or blocking of objects. The advantage of this approach is that multi-threaded objects can continue functioning partially, since only threads that may use the passivated links are required to block. Nevertheless, this implies additional burden to the application developer, which must provide functionality to restrict individual threads that use a specified link.

Unlike the approaches in [Kramer85, Wermel99], this algorithm is not suitable for a system with re-entrant transactions. Since Bidan et al.'s work has focused on CORBA-based distributed systems, this means that the reconfiguration of systems with re-entrant invocations is not supported.

Another major limitation of this approach is that it does not support multiple simultaneous object replacements.

4.2.4 Wermelinger

Wermelinger's approach [Wermel99] considers link passivation, as in [Bidan98]. Nevertheless, more fine-grained information on the objects is used than in [Bidan98].

A system is defined as a set of connected nodes, where a connection is given by an initiator port and a recipient port. For each node, port dependencies are specified. A port dependency is defined by a recipient port and an initiator port. Port I is said to be dependent of port R, if upon reception of a transaction in R, a transaction is initiated in connections leaving from I. This makes it possible to relate transactions and derive transaction dependencies.

This approach requires an object to be shipped with a description of the object's internal port dependencies. However, this sort of specification has to be made by the application developer, and violates the time and expertise requirements, as identified in Chapter 3 (generic requirements for QoS Mechanisms). Wermelinger's work is presented at a theoretical level, and so far it has not been implemented.

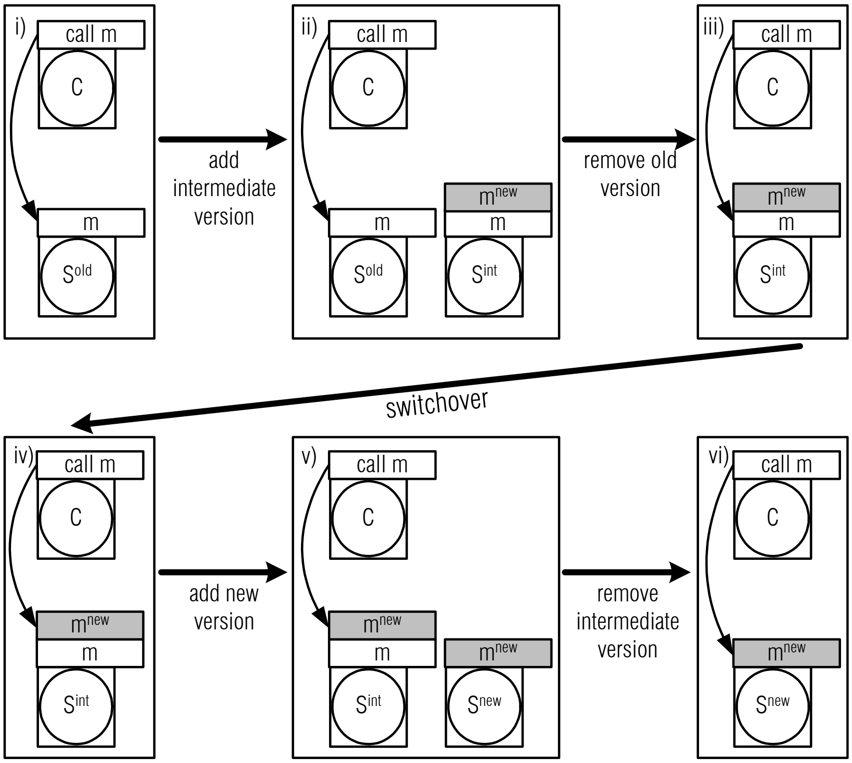
4.2.5 Tewksbury et al.

Parallel and independent research of Tewksbury et al. [Tewksb01A, Tewksb01B, Tewksb01C, Tewksb01D, Moser00] propose an approach that extends a Fault Tolerant CORBA [FTCORBA] implementation named Eternal [Narasimh99B] with dynamic reconfiguration capabilities. This approach exploits the replication functionality provided by Eternal.

The basic idea behind this approach is to replace old versions of the objects by an intermediate version that implements both the behaviour of the old version, and of the new version of the object. The intermediate version of the object is usually generated based on the code for the old and the code for the new object. Although not explicitly mentioned in the literature, this implies that the old and the new object are written in the same (version of the) programming language. After all old objects have been replaced by intermediate objects, and all intermediate objects are in a quiescent state, all intermediate objects simultaneously switchover to the new behaviour using a special method that is implemented by the intermediate objects. The intermediate version can then be replaced by the actual new versions of the objects.

Figure 4-7 shows the process as we sketched it. It shows a client object (C), that invokes a method m on server object S. We want to upgrade server object S, from version S^{old} to version S^{new}. i) shows the start situation before the reconfiguration activities start. In situation ii) the intermediate version of server object (S^{int}) is instantiated, and forms an object group with S^{old}. Invocation could be multi-casted if Eternal uses active replication, or in case of passive replication the state is periodically synchronized, and the invocations between synchronization are logged. In situation iii) the old version of S is removed. The normal replication functionality of Eternal makes sure that the client now uses S^{int}. In situation iv) the switchover has taken place, and the new code is now used. In situation v) the actual new version of S is added, and forms an object group with the intermediate version of S. In vi) the intermediate version has been removed, and Eternal made sure this was transparent for the client.

Figure 4-7 Stepwise upgrade using Eternal



Consistency

A mutual consistent state is preserved by requiring an object to be in a quiescent state. In the quiescent state, the state can be transferred using the usual get and set state operations that the application developer has to implement. Objects have to be quiescent three times, once for the replacement of the old with the intermediate version, once for the switchover of the intermediate version from the old to the new behaviour, and once to replace the intermediate version with the new version.

Messages are queued to get the objects in a quiescent state. The quiescence algorithm that controls the queuing makes it possible to have nested invocations. Eternal does not allow multi-threading and re-entrance [Narasimh99B], and the quiescence algorithm also assumes single-threaded objects.

State Access

To assist a component developer with implementing the state access method, [Tewksb01A] describes a tool that parses the source code and generates the `get_state()` and `set_state()` methods. Benefits of generating these methods is that is can save the component developer considerable time, and it can prevent some errors that manual state access methods might contain. A disadvantage is that some state that is captured in this way might not be relevant, or can easier be re-created in the upgraded component. Also the component developer cannot rely completely on this generated state access code, for example some of application state can be location dependent. A concrete example is the usage of logging component that resides on each node for efficiency reasons. After a migration the migrated component will, for efficiency reasons, want to use the local logging component and not the logging component on the node it used to reside. A third disadvantage is that this generated code does not work in case of an upgrade to another programming language. A developer has to modify the generated state access code in these cases. [Tewksb01A] mentions a graphical interface to assist the component developer with this. If the state actually has to be converted because of changes, this conversion can typically not be generated, and has to be done manually.

Group IOR

This approach uses the group Interoperable Object Reference (IOR), as specified by Fault Tolerant CORBA, to maintain structural integrity. A group IOR is basically the list of IORs of the replicas that are part of the group. Which replicas are part of the group, and changes in this group, are transparent to the clients. If the list of replicas changes, the client gets an updated group IOR the first time he makes an invocation. It is thus not pushed to the client. An issue with using the group IOR is that clients send requests to the old IOR, so either the updated object has to use the same IOR (and thus be on the same node), or some object should use the same IOR and send the clients the updated group IOR. This however means that the node cannot be taken offline.

Some criticisms on this approach are:

- Eternal is based on operating system interceptors, which are ORB independent, but cannot intercepts invocations between co-located objects [Wegdam00C].
- The requirements for strong replica consistency presented in [Narasimh99B] imply that an object can only service one invocation at a time. Similarly to Goudarzi et. al., re-entrant invocations are also ruled out.
- Because the intermediate object mixes the old and new implementation code, access to the source code is required, and the old and new version of the code have to be in the same (version of) programming language.

4.2.6 Observations

A common characteristic of the approaches we have studied is the definition of a reconfiguration-safe state. A system is driven into this safe state by algorithms that interfere with the execution of the system.

All the approaches studied, except for Tewksbury et al., use some formal representation of the system. These representations are described in configuration languages [Kramer85, Wermel99] or in configuration graphs [Bidan98, Goudarzi99]. The representations are used to identify which activities of the system should be deferred in order to reach the safe state. The use of these representations may have implications on the scalability of the solutions since for large-scale systems providing such information can be problematic.

The approaches studied do not assume the same computation model. For example, in the computation model assumed by –Goudarzi and Tewksbury et al., an application entity cannot be involved in several interactions simultaneously, while in the computation model assumed by Kramer and Magee this restriction does not apply. The computation model assumed by an approach has direct implications on the definition of a safe state and the algorithms to reach this safe state.

The approaches proposed by Tewksbury et al. and Bidan are aimed at middleware-based applications (CORBA in both cases). The other approaches assume that the application developer implements all the distribution functionality himself, and has full control over this. Also in Bidan's approach, the application developer has to implement functionality to passivate a link himself, making this approach also not very transparent.

Table 4-2 summarizes this comparison between the approaches to dynamic reconfiguration.

Table 4-2 Comparison of studied approaches to dynamic reconfiguration

Approach	Uses Formalism	Simultaneous interactions	Re-entrance	Middleware-based	Implemented	Driven safe state
Kramer-Magee	yes	yes	yes	no	yes	yes
Goudarzi	yes	no	no	no	yes	yes
Bidan	yes	yes	no	yes	yes	yes
Wermelinger	yes	no	no	no	no	yes
Tewksbury	no	no	no	yes	yes	yes

4.3 A New Dynamic Reconfiguration Mechanism

This section describes our mechanism for dynamic reconfiguration of component-middleware-based applications. Our mechanism addresses each of the generic requirements for QoS mechanisms (as identified in Chapter 3) and specific requirements for dynamic reconfiguration, which we will also list in this section. These specific requirements include the correctness aspects for a reconfiguration as identified in Section 4.1.

This section is further structured as follows: Section 4.3.1 motivates the need for a new mechanism for dynamic configuration of component-middleware-based applications, Section 4.3.2 states the specific requirements for such a mechanism, Section 4.3.3 presents the reconfiguration possibilities supported by our mechanism and Section 4.3.4 describes the mechanisms prescribed for change management. Finally, Section 4.3.5 discusses the limitations of our mechanism and Section 4.3.6 compares it to the approaches found in the literature.

4.3.1 Motivation

Most of the approaches found in the literature do not address component-middleware-based applications specifically. As a consequence, either they consider a computing model that is limited with respect to our component model, e.g. ruling out multi-threading or re-entrance, or they fail to address issues that are particularly relevant for component middleware systems, such as, e.g., interface evolution.

While some dynamic reconfiguration approaches that have not been originally developed for middleware platforms may be used in distributed component applications, only a component-middleware-based approach is able to profit from particular characteristics of component middleware.

As explained in Chapter 3, component middleware provides functions to locating components, relocating components, multi-casting requests and replies, holding requests and replies and change the order of request and replies. These functions provide an opportunity for the provision of reconfiguration transparency. Application developers can profit from reconfiguration functionality with the benefits of a middleware-supported service, e.g., interoperability, application portability, language independence, and wide support, requiring minimal expertise in the field of dynamic reconfiguration.

4.3.2 Requirements

The following requirements have been considered in the conception of our approach for a dynamic reconfiguration QoS mechanism, in addition to the generic requirements identified in Chapter 3:

1. *Correct incremental evolution* – The mechanism must include functionality to obtain a correct incremental evolution of a system, as defined in Section 4.1.3. The integrity of the component model must be preserved under normal operation, i.e., when reconfiguration is not taking place, and during reconfiguration.
2. *Impact on execution* – The mechanism should minimize impact on execution during reconfiguration, and it should account for little overhead during normal operation.
3. *General applicability* – The mechanism should minimize restrictions on applications. In particular, it should be suitable for systems with off-the-shelf, multi-threaded, stateful, re-entrant and active components. This requirement is somewhat overlapping with the flexibility and generality requirements for QoS Mechanisms, as identified in Chapter 3 (generic requirements for QoS Mechanisms). We want to stress this point since most approaches do not support re-entrant components and multi-threaded execution models.
4. *No additional formalisms* – The mechanism must not require the use of additional formalisms for application development. This is a specialization of time and expertise requirements, as identified in Chapter 3. We want to make this requirement explicit because most approaches use additional formalisms.
5. *Composite reconfiguration steps* – The mechanism should allow reconfigurations that involve multiple entities.

4.3.3 Supported Reconfigurations

In our dynamic reconfiguration mechanism, entities subject to reconfiguration are called reconfigurable components. A reconfigurable component is a component that can be manipulated through reconfiguration operations, namely *creation, replacement, migration* and *removal*.

Component Creation

Component creation allows an application to create a component at run-time. From the moment a component is created, references to its interfaces are used to communicate with it. Component creation is a trivial case from the perspective of change management, since applications are expected to cope with it.

Figure 4-8 depicts component creation from an abstract perspective.

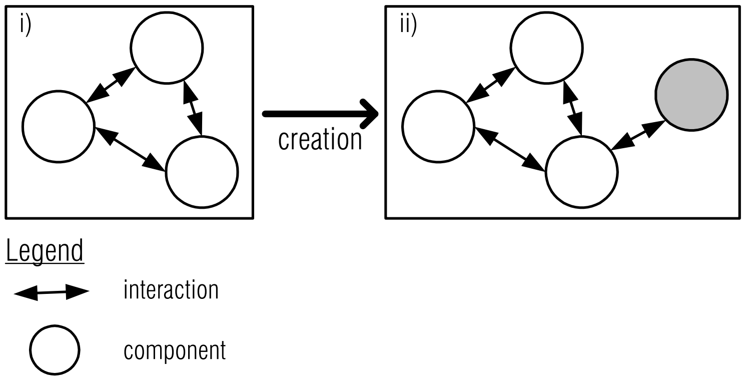


Figure 4-8 Component creation

Component Replacement

Component replacement allows one version of a component to be replaced by another version. We use the term version of a component to denote a set of implementation constructs that realizes a component (a template).

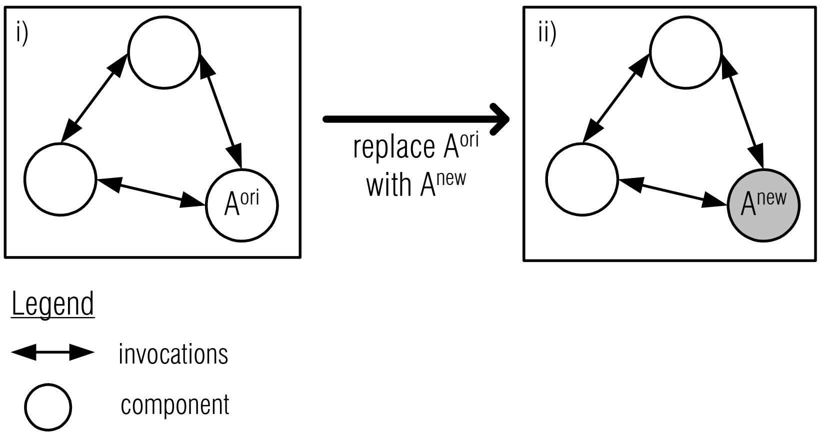
The new version of a component may have functional and Quality-of-Service (QoS) properties that differ from the old version. For example, the new version may correct faults in the original version, or implement additional functionality. The reconfiguration designer is responsible for assuring that the new version of a component satisfies both the functional and QoS requirements of the environment in which the component its inserted.

In addition, in our definition of replacement, the new version of component may run in another location or in another type of execution environment supported by the component-middleware platform, e.g., a different programming language and/or operating system.

Component replacement requires special attention from the perspective of change management, since it threatens application consistency.

Figure 4-9 depicts component replacement from an abstract perspective. Component A is replaced, substituting its original version A^{ori} with a new version A^{new}.

Figure 4-9 Component replacement



Replacement with Interface Changes

We define a version A^{new} of a component A conforming with a version A^{ori} if the interfaces of A^{new} are identical to the interfaces of A^{ori} or are a subtype from it [Liskov88], and non-conforming otherwise.

Replacement of a current version by a non-conforming version is called non-conforming replacement. Our mechanism only supports non-conforming replacements in special cases that are explained later in Section 4.3.4.

Component Migration

Migration means that a component is moved from its current location to a new location. A component preserves its identity and state. Because in our mechanism replacement can involve changing the location, we consider migration a special type of replacement in which the version of the component does not change.

Figure 4-10 depicts component migration from an abstract perspective. Component A migrates from its original location X to a new location Y.

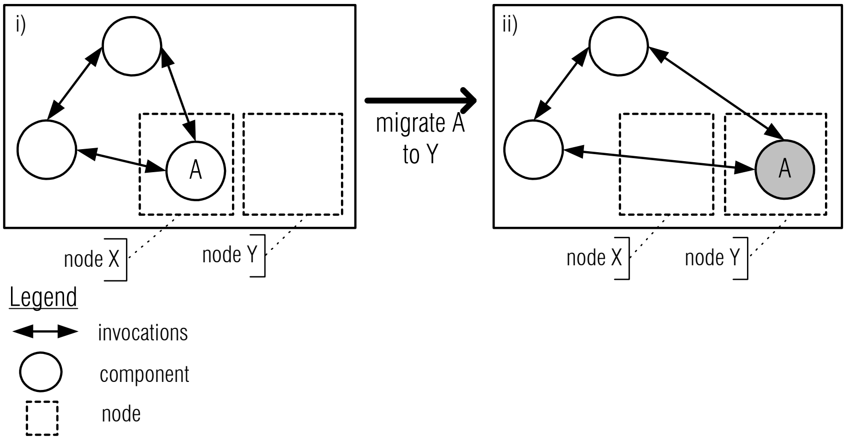


Figure 4-10 Component migration

Component Removal
Component removal allows an application to remove a reconfigurable component at run-time. From the moment a component is removed, the reference to its interface becomes invalid. Component removal is a trivial case from the perspective of reconfiguration management, since applications are expected to cope with it.

Figure 4-11 depicts component removal from an abstract perspective.

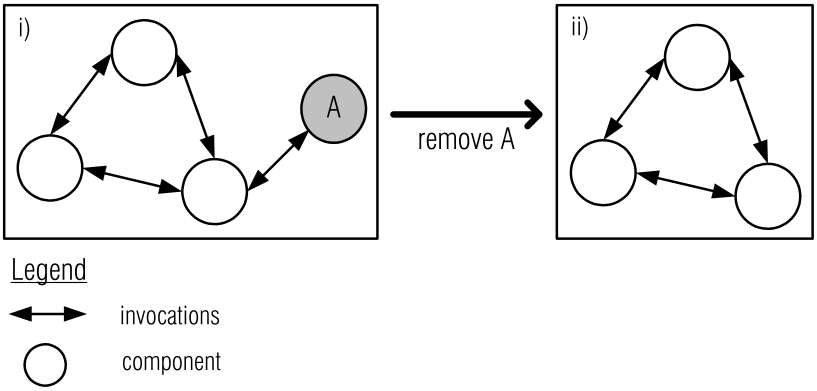


Figure 4-11 Component removal

Reconfiguration Steps
A system evolves incrementally from its current configuration to a resulting configuration in a reconfiguration step. A reconfiguration step is perceived as an atomic action from the perspective of the application.

A reconfiguration step consists of:

1. the execution of a reconfiguration operation in a component, in which case it is called *a simple reconfiguration step*; or
2. the execution of reconfiguration operations in several distinct components, in which case it is called a *composite reconfiguration step*.

Composite reconfiguration steps are often required for reconfiguration of sets of related components. In a set of related components, a change to one component A may require changes to other components that depend on A's behavior or other characteristics.

Figure 4-12 depicts a composite reconfiguration step from an abstract perspective, where component D is removed, components A and B replaced, and component E is created.

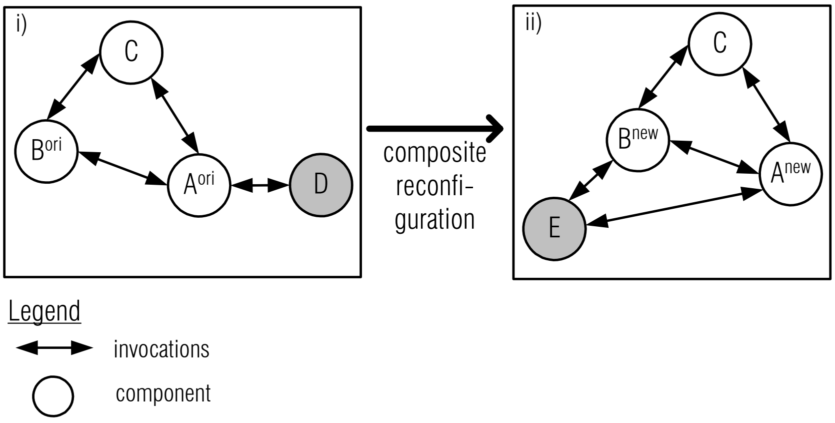


Figure 4-12 Composite reconfiguration step

A particular case of composite reconfiguration step is the replacement of multiple components. A common usage example is to replace all components of the same type with a new version.

4.3.4 Change Management

This section describes how we ensure consistency in our mechanism by addressing each of the correctness aspects identified in Section 4.1.3.

Structural Integrity

The main issues that have to be dealt with for a component-middleware-based system with respect to preserving structural integrity are referential integrity and interface compatibility.

Referential integrity becomes an issue whenever a component reference changes. A component reference is defined as a value that denotes a particular component, and is used by the middleware infrastructure to identify and locate the component. References acquired by clients prior to reconfiguration may be invalidated due to reconfiguration. If a reference points to a component that no longer exists, the established logical binding between a client and a target component is broken. In order to re-establish the binding after reconfiguration, we provide a logically central point of contact for clients to find the component with invalidated component references.

To preserve *interface compatibility*, a new version of a component must satisfy the original interfaces. In component-middleware technologies, interfaces satisfy the Liskov substitution principle [Liskov88]:

> *"If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T."*

This means that to satisfy the original interfaces, the new interfaces have to implement the original interfaces, or implement interfaces that are subtypes of the original interfaces. We refer to this as a *conformant replacement*. A *non-conformant replacement* is possible by introducing a wrapper component that is conformant, or by replacing all the clients of the replaced component as part of the same (composite) reconfiguration step.

It is possible to apply non-conforming replacements that promote arbitrary changes to an interface of a reconfigurable component if either one of the following conditions is satisfied:

1. all clients of the reconfigurable component that use the interface that is changed are also reconfigurable components,
2. the reconfiguration designer supplies a wrapper version of the component that is capable of translating requests to the new version.

Both cases require the use of a composite reconfiguration step. In the first case, the reconfigurable component and all the clients that use the interface that is changed are replaced. In the second case, the new version is created and the reconfigurable component is replaced by the wrapper version.

Mutually Consistent States

Drive to safe state

We propose an algorithm to drive the system to the safe state that uses information obtained from the middleware platform at run-time and freezes system interactions on-demand. This algorithm consists of these three stages:

1. Drive the system to the safe state by *deferring invocations* that would prevent the system from reaching the safe state;
2. *Detect that the safe state* has been reached; and
3. *Apply reconfiguration*;

Affected component

Safe state

We use the term *affected component* to denote a reconfigurable component that is replaced, migrated or removed as a consequence of reconfiguration. The system is said to be in the *reconfiguration-safe state* when each affected component (i) is not currently involved in invocations and (ii) will not be involved in invocations until after reconfiguration. As defined in Chapter 2, an invocation can be split up in the client component issuing a request, the server component processing the request, and sending the reply back to the client component. This means that when the system is in a reconfiguration-safe state none of the affected components is processing requests or waiting for replies.

Active and reactive components

We distinguish components in general as active and reactive. *Reactive components* are components that only initiate requests that are causally related to incoming requests. *Active components* may initiate requests that do not depend on incoming requests, e.g., they may initiate requests as a result of the elapsing of a time-out.

An active component should have capabilities for driving itself to a reactive state, in which it refrains from initiating requests that are not causally related to an incoming request. The implementation of the operation for forcing reactive behavior is a responsibility of the component developer. Once the set of affected system components is defined, all active components in the set are requested to exhibit reactive behavior.

Since the component developer has to implement this, we violate transparency here. We however prefer this to not allowing the reconfiguration of active components since this would violate the general applicability requirement we identified earlier this chapter.

Reaching the safe state

We guarantee the reachability of the safe state by interfering with the activities of the system. All affected components are requested to exhibit reactive behavior, and then pending invocations in the affected components are allowed to complete.

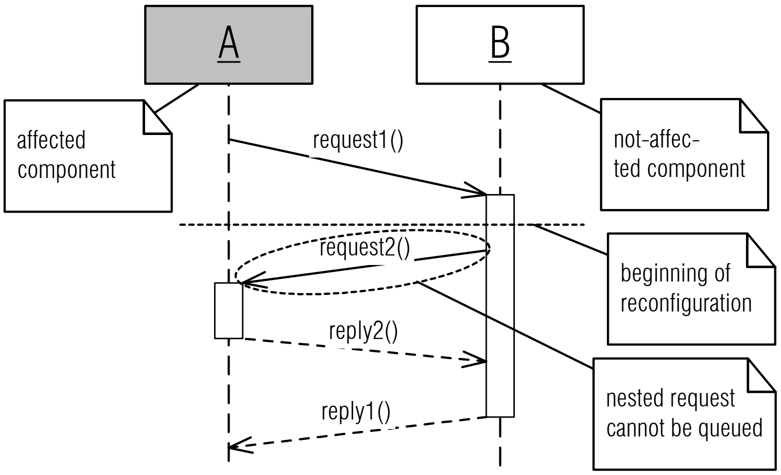
In the case of a simple reconfiguration step, with the replacement or migration of a single non re-entrant component, all requests issued to this component are queued by the middleware platform before they reach the component. Queuing of requests is one of the middleware functions we identified in Chapter 3.

In this way, new invocations are prevented from being served before the reconfiguration, and the component gets the chance to finish handling ongoing invocations. When all ongoing invocations have been treated, the system is in the safe state. Since all invocations are guaranteed to finish within bounded time, the safe state is reachable within bounded time.

There is only one restriction we have here. The completion of an ongoing invocation might depend on another incoming invocation, which we queue and therefore never reaches the object. The case where this is a re-entrant invocation we discuss below. This leaves the case of a non-re-entrant invocation, i.e., an invocation that is not dependent on the ongoing invocation but still needs to be processed by the object before it can complete the ongoing invocation. These types of dependencies are not common, but they are possible. An example is an object with two methods: `some_event_occurred()` and `wait_for_some_event()`. The second method blocks waiting for the first method to be called. An ongoing `wait_for_some_event()` would never complete since we queue all invocations including the `some_event_occurred()` invocation that is needed for the completion of `wait_for_some_event()`. We cannot detect such cases and therefore do not allow them. Since such dependencies are very rare, we do not consider this a major disadvantage.

In the case of replacement or migration of a single re-entrant component, we should not queue up re-entrant invocations. A re-entrant invocation is not queued, since otherwise the affected component would have a pending outgoing invocation that would never complete. Consequently, the system would never reach the safe state. *Figure 4-13* shows a re-entrant invocation (`request2`) that must be allowed to complete for reconfiguration to proceed.

Figure 4-13 Requests that must not be queued in case of a simple reconfiguration step



In the case of composite reconfiguration steps, several affected components have to be driven to the safe-state. In this case, we should neither queue up requests issued by an affected component nor the nested requests that are a consequence of invocations issued by an affected component. If one of these requests would be queued, there would always be a pending outgoing request in the set of affected components that would never complete, and the system would never reach the safe state. *Figure 4-14* shows invocations that must be allowed to complete for reconfiguration to proceed.

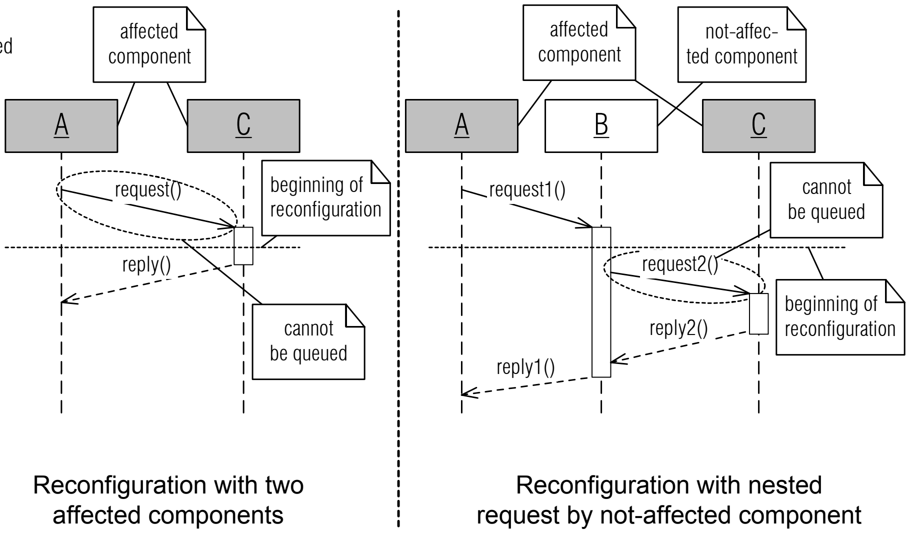

affected component
A
C
request()
beginning of reconfiguration
reply()
cannot be queued
Reconfiguration with two affected components
affected component
not-affected component
A
B
C
request1()
cannot be queued
request2()
beginning of reconfiguration
reply2()
reply1()
Reconfiguration with nested request by not-affected component

Figure 4-14 Requests that must not be queued in case of composite reconfiguration step

Therefore, in a system under reconfiguration, we can distinguish three sets of invocations:

Laissez-passer set

Blocking set

1. invocations whose processing is necessary for the system to reach the reconfiguration-safe state, the *'laissez-passer' set* ('laissez-passer' is French for 'letting pass'),
2. invocations whose processing could prevent the affected components from reaching the reconfiguration-safe state (*blocking set*), and
3. invocations that do not involve any affected component.

In our mechanism, the middleware platform is responsible for selectively queuing requests that belong to the blocking set and for allowing requests in the 'laissez-passer' set to be processed. This is done transparently for the application components.

In order to identify requests that belong to the 'laissez-passer' set, we use the propagation of *implicit parameters* along invocation paths. For every reconfigurable component in an invocation path the middleware infrastructure adds the component's identification to the request as an implicit parameter.

Given a request and the set of affected components, it is possible to determine if a request belongs to the 'laissez-passer' set by inspecting its implicit parameters. If at least one of the affected components has been included in the request's implicit parameters, the request belongs to the 'laissez-passer' set, and should be allowed to complete.

Applying reconfiguration

When all affected components are idle, reconfiguration can proceed. The affected components' state can be inspected and used to derive the state of the components being introduced. Once newly created components, new versions and relocated components have been instantiated (which may already be done before driving the system to the safe state), their state is properly modified. After their state is modified, they are allowed to exhibit active behavior. Queued requests and further new requests are redirected to the new or relocated version of a component after reconfiguration.

Application-state invariants

Each reconfigurable component must provide operations to access its state. These operations are used to inspect and modify a selected subset of the component's internal state. The component developer is responsible for deciding on the particular subset of the components' state that is exposed by these *state-access operations*. In general, a component should provide operations to inspect and modify its control and data state. These operations are only invoked in the safe state. Since in the safe state a component is idle, the amount of control state to be externalized is minimized.

State access operations

When the system to be reconfigured is in the safe state, the state of the affected components can be accessed consistently through these state-access operations, and can be used as input to a state translation function supplied by the change designer. The *state translation function* determines the state of the new version of each affected component so as to guarantee that application invariants are not violated. Furthermore, the state translation function may have to adjust the state of an affected component so that its behavior is compatible with the behavior expected by its environment.

State translation function

It is possible, however, that such state translation function does not exist for two given versions, preventing reconfiguration. This situation can be illustrated in a simple component replacement. It is possible that the new version of the component does not have a state that corresponds to the state of the original version. For example, let us consider a component that generates unique identifiers, with an initial version that generates identifiers counting up from A to B. The state variable of this implementation is the counter containing the last generated number S. Let us suppose that the new version also produces unique identifiers, but does it counting down from B to A. While the identifiers produced so far are known (all values smaller or equal to S), there is no value for the internal counter in the new version that can be derived so as to preserve the expected behavior of the generator. With any starting state, the new version would end up producing identifiers that have already been used in the previous version of the component, introducing inconsistency in the application.

Impact on Execution

While some reconfiguration approaches [Kramer85, Goudarzi99] quantify the impact on execution as proportional to the number of affected reconfigurable entities, or proportional to the number of blocked execution threads in application components [Bidan98], we intend to estimate the increase in response time QoS characteristic experienced by clients of affected components during reconfiguration.

Clients of an affected component may observe an increase in the response time of operations invoked on an affected component during reconfiguration. This increase only applies to invocations initiated by the clients of a target component after the beginning of the reconfiguration and before the end of the reconfiguration.

The increase in response time during reconfiguration is highly dependent on the application. Since in our mechanism we wait for ongoing interactions involving the affected components to finish, the expected increase in response time is proportional to the expected duration of those interactions. Therefore, this increase is higher for systems with long-lived interactions. The increase is limited by the duration of the longest pending invocation in the set of affected components at the moment the reconfiguration starts. For active components, the amount of time taken for the component to exhibit reactive behavior should also be considered in the calculation of the upper bound of the increase in response time. We expect however this increase to be insignificant for most applications.

Considering the absolute increase in response time, the mechanism seems to be best suited for applications with short-lived interactions. Ultimately, the maximum acceptable increase in response time during reconfiguration is determined by the QoS requirements for the application.

4.3.5 Limitations of our Mechanism

Our mechanism does not use any formal representation of the system, contrary to the approaches presented in [Kramer85, Goudarzi99, Oreizy98, Wermel99]. Instead, we discover during run-time the configuration information that we need to preserve consistency. There are application invariants that cannot be discovered during run-time, and therefore cannot automatically be preserved by our dynamic reconfiguration mechanism. We rely on the reconfiguration designer to preserve these application invariants. As an example, suppose a system has the property that it has to be organized as a ring. A removal of one component should preserve the ring. The fact that the components have to be organized in a ring cannot be discovered during run-time, and therefore the reconfiguration designer is responsible for preserving this ring property.

Since we have opted for complete transparency for clients of reconfigurable components, we can only support non-conforming replacements in restricted cases, as identified in Section 4.3.4. A less restrictive support to non-conforming replacements would require clients of reconfigurable components to be developed with mechanisms to cope with arbitrary interface change.

Another limitation refers to the externalization of state. In the component model adopted, relationships between components may be buried in the implementation of a component, in the form of component references. These component references cannot be easily manipulated externally. This forces the externalized state of a component to include all the component references that are still required for the component to continue operating and that otherwise would not be recovered from the system after reconfiguration.

4.3.6 Comparison with Studied Approaches

This section presents distinctive features of our mechanism and compares it with the reconfiguration approaches studied.

Application-description Models

Our mechanism does not require the use of specific description formalisms for application development. Some of the dynamic reconfiguration approaches studied [Kramer90, Goudarzi99, Oreizy98, Wermel99] prescribe the use of formal architectural or configuration models to describe an application. These models are produced by the application designer during the development process, and are described in Architecture Description Languages (ADLs) or Configuration Languages (CLs).

These models are used by dynamic reconfiguration approaches to derive how to apply changes to a system under reconfiguration. For example, in Kramer and Magee's approach [Kramer90] an application is represented as a directed graph, whose nodes are system entities and whose arcs are connections between entities. An entity A is connected to an entity B if A can initiate a transaction with B. For an entity Q to be replaced, all the entities that are capable of initiating transactions directly or indirectly on Q should exhibit passive behavior, as well as Q itself. In this case, the configuration graph is used to identify which entities must exhibit passive behavior for the system to reach the reconfiguration-safe state. In Wermelinger's approach [Wermel99], application entities must be supplied with a description of internal port dependencies, which relate input ports and output ports.

A drawback of prescribing an ADL or CL for application design is that the conventional development process has to incorporate the production of a description of the application using the specific formalism or language. Our mechanism differs from these in the sense that it does not prescribe the use of an ADL or CL. The configuration information required to apply reconfiguration is obtained from the system at run-time. By doing this, we intend to separate the concerns of obtaining and maintaining configuration information for the reconfiguration design activities, and obtaining and maintaining configuration information for the application of reconfiguration.

Figure 4-15 shows a refinement of the model presented in Section 4.1, obtained by decomposing configuration information into configuration information obtained at design-time and obtained at run-time. Our mechanism covers the grayed boxes, i.e., the instrumentation to obtain configuration information at run-time, and the Reconfiguration Management functionality. The rest of the model is not covered by our mechanism, and is the responsibility of the reconfiguration designer. We could however extend our mechanism by using design-time ADLs of CLs to give more information about the design and architecture of the application to the Reconfiguration Design Activities. This would especially facilitate maintaining of application invariants. The usage of ADLs or CLs however would add to the responsibilities of the application designer, and thus reduce the transparency.

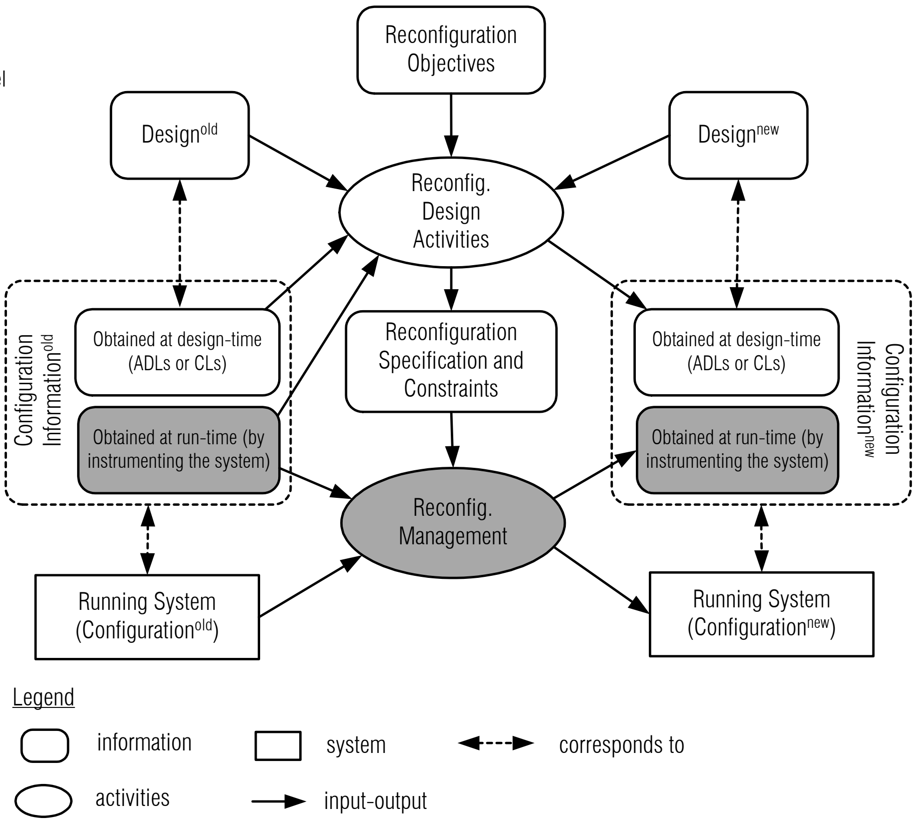

Reconfiguration Objectives
Design^old
Design^new
Reconfig. Design Activities
Configuration Information^old
Obtained at design-time (ADLs or CLs)
Obtained at run-time (by instrumenting the system)
Reconfiguration Specification and Constraints
Configuration Information^new
Obtained at design-time (ADLs or CLs)
Obtained at run-time (by instrumenting the system)
Reconfig. Management
Running System (Configuration^old)
Running System (Configuration^new)
Legend
information
system
corresponds to
activities
input-output

Figure 4-15 Dynamic Reconfiguration Model with refined configuration information

Reconfiguration Supported and Computation Model

Bidan et al. consider in [Bidan98] an approach to dynamic reconfiguration of CORBA-based applications and a reconfigurable entity is a CORBA object. This is similar to our mechanism, since a CORBA object is a specific example of our more generic component concept. In Bidan's approach, the reconfiguration infrastructure maintains a representation of the configuration of the system, through a directed graph of components connected through links. Components A and B are said to be linked if A can invoke an operation on target component B. In the approach, all client applications and target components must implement a passivate operation to block the initiation of requests in a specific outgoing link. The algorithm guarantees the reachability of an idle state by sending passivate messages to all the clients of a component and then to the component itself.

Unlike our mechanism, Bidan et al.'s approach does not support composite reconfiguration steps. In sets of related components, it is common that a change to one component may require changes to other components that depend on the component's behavior or other characteristics. Since only simple reconfiguration steps are allowed, the application of this approach is limited.

Furthermore, the approach does not support applications with re-entrant invocations. Therefore, a component that has initiated an invocation cannot play the role of server for some consequent invocation. The approach does not support re-establishment of application invariants and state translation either.

A second approach that is based on CORBA is the replication-based approach by Tewksbury et al. [Tewksb01B]. As stated before, this work was done independently and parallel to our work, and was published after we finished our work. The main differences between our mechanism and the Tewksbury et al. are:

- *Threading* - contrary to our mechanism, in the Tewksbury et al. approach an object can only service one invocation at a time [Narasimh99B], which means re-entrant invocations are also ruled out. This violates our general suitability requirement.
- *Implementation language* - because the intermediate object mixes the old and new implementation code, access to the source code is required, and the old and new version of the code have to be in the same (version of) programming language.

There are some more differences in how the approaches were implemented, which we will discuss when we have discussed our CORBA based prototype in Chapter 6.

Impact on Execution

Our mechanism proposes a mutual consistency mechanism that only interferes with application activities that require interaction with affected components during reconfiguration. This is not the case in most approaches we have studied [Bidan98, Kramer85, Goudarzi99, Wermel99], which block all potential system activities that may prevent the system from reaching the safe state.

Transparency

Our mechanism is completely transparent for the clients of reconfigurable components, in contrast with [Bidan98, Kramer90, Goudarzi99, Wermel99] where client applications have to provide support for reconfiguration.

Our mechanism facilitates the development of reconfigurable components by incorporating change management functionality in the middleware infrastructure. Therefore, it requires minimal reconfiguration expertise and development effort from the component developer. The transparency is not complete for the reconfigurable component developer, specifically the reconfigurable component developer is responsible to provide state-access operations and operations to drive an active component from an active state to a reactive state and back.

4.4 Design Overview

This section gives a high level overview of the design of our Dynamic Reconfiguration QoS mechanism. Details on the design of the prototype of this mechanism in CORBA can be found in Chapter 6.

Dynamic Reconfiguration Service

As described in Chapter 2, common functionalities that go beyond the basic communication functionalities are put in a common service, or service for short. We therefore designed the Dynamic Reconfiguration QoS mechanisms as a common service, and will refer to it as the *Dynamic Reconfiguration Service*.

The Dynamic Reconfiguration Service consists of a Reconfiguration Manager, a Location Agent and Reconfiguration Agents, and is depicted in *Figure 4-16*.

Reconfiguration Manager

The *Reconfiguration Manager* is the central component of the Dynamic Reconfiguration Service in that it interacts with all the other components of the service. It coordinates reconfiguration with Reconfiguration Agents and the Location Agent. The Reconfiguration Manager delegates object creation and removal to Reconfigurable Component Factories, it registers, re-registers and de-registers components through interaction with the Location Agent and it co-ordinates the Reconfiguration Agents to drive the system to a reconfiguration-safe state.

Reconfiguration Agent

A *Reconfiguration Agent* is created for each middleware instance that mediates invocations for reconfigurable components. Typically there will be a middleware instance per capsule. A Reconfiguration Agent is responsible for restricting the behavior of an affected component during reconfiguration through filtering of requests.

Location Agent

The *Location Agent* provides a registry for the location of reconfigurable components. It produces location-independent component references, and is capable of translating a location-independent component reference to a component reference with the current location of a reconfigurable component.

A *Reconfigurable Component* is the unit of reconfiguration. It provides state-access operations and is able to exhibit reactive behavior upon demand.

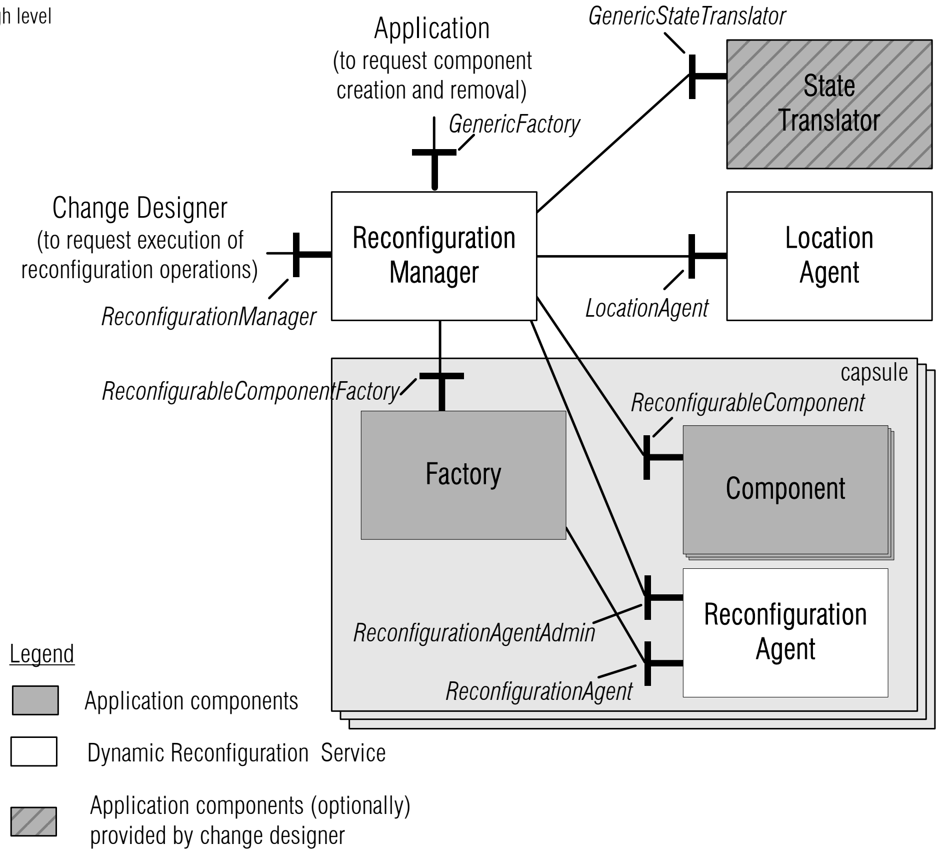
Reconfigurable-Component Factory

A *Reconfigurable-Component Factory* implements the Factory design pattern, creating and removing versions of Reconfigurable Components on behalf of the Reconfiguration Manager. Factories shield the Dynamic Reconfiguration Service from the specific support to component deployment offered by different languages, operating systems or virtual machines, such as, e.g., DLLs and the Java class loader.

The *State Translator* implements a state translation function, if needed for a reconfiguration. This state translation is application dependent.

As *Figure 4-16* also shows, the service concerns both the change designer and the component developer. The change designer can access the service of the Dynamic Reconfiguration Service to request the execution of reconfiguration steps, and has to supply the State Translator (if needed). The component developer has to supply the application-specific Reconfigurable-Component Factories and Reconfigurable Components that comply with the interfaces defined by the service. The Reconfiguration Manager, Location Agent and the Reconfiguration Agents are supplied by the developer of the Dynamic Reconfiguration Service.

Figure 4-16 High level design



4.5 Conclusions

We presented our model for dynamic reconfiguration, presented a new classification of dynamic reconfiguration approaches, described concepts and terminology in the area of dynamic reconfiguration and described, evaluated and compared related work in the area of dynamic reconfiguration. Based on this, we developed a new dynamic reconfiguration mechanism for component-middleware-based applications, and compared this to the existing approaches. We also presented a high level design of the Dynamic Reconfiguration Service that implements our mechanism. Chapter 6 describes a prototype of the Dynamic Reconfiguration Service, implemented using CORBA, which serves as a proof of concept.

Our dynamic reconfiguration mechanism:
- supports component creation and removal;
- supports replacement with a new version of the component with the same identity. This new version may run in another execution-environment type supported by the middleware platform;
- supports migration;
- supports composite reconfiguration steps, in which several components are reconfigured in an atomic action from the perspective of the application;
- prescribes how to obtain a correct incremental evolution of a system, preserving the component model;
- is applicable to a broad range of applications, including applications built from off-the-shelf components, multi-threaded components, re-entrant components, and stateful components;
- minimizes impact on execution during reconfiguration;
- scales with respect to the number of clients;
- provides full reconfiguration transparency to client developers, and requires minimal reconfiguration expertise from the reconfigurable component developer;
- does not require the use of a specific programming language for application development;
- does not require the use of additional formalisms for application development.

Component Middleware and Transparency
Contrary to most existing approaches we studied, our mechanism exploits the particular characteristics of component middleware. Embedding reconfiguration functionality in the middleware layer is a promising way to achieve transparency. Our mechanism can be realized with minimal

additional burden on the developer of the reconfigurable components and is fully transparent to the developer of client components. We compared our mechanism with middleware-based and non-middleware-based approaches in 4.3.6.

Our mechanism can be used in systems with a large and changing number of components, addressing the telematics requirement, see Chapter 3 (generic requirements for QoS Mechanisms). We can use message reflection, and in particular middleware interceptors, to instrument the middleware platform to obtain configuration information at runtime, as we will show in Chapter 6 when we discuss our implementation. This avoids burdening the component developer to provide extensive descriptions of the system and its components, and addresses the flexibility, time and expertise requirements (see Chapter 3). In addition, by using message reflection we are able to provide the functionality we need to freeze system interactions on demand when we drive the system to a reconfiguration-safe state. Our mechanism only interferes directly with those parts of the system that actually interact with the set of affected components during reconfiguration, allowing the rest of the system to execute normally and reducing the impact on execution.

Performance Impact

Reconfiguration of objects that are involved in long-running interactions may implicate high increase in response time experienced by the clients of the affected objects. Ultimately, the maximum acceptable increase in response time during reconfiguration is determined by the QoS requirements for the application.

Concluding Remarks

The main limitation of the mechanism is that it does not cope with preserving application invariants that cannot be discovered during run-time. The mechanism assumes that the reconfiguration design activities produce changes that have been validated a priori, and puts the responsibility for this with the reconfiguration designer.

The dynamic reconfiguration mechanism could be extended with the abortion of interactions that are possibly long running and do not affect the state of a component. This would lead to a hybrid abortion-avoidance and abortion approach that could decrease the impact on system execution, especially for systems with long-running interactions.

Chapter 5

Load Distribution

This chapter describes a QoS mechanism that improves the performance characteristics of a distributed application by distributing the load. Our focus is on making load distribution transparent for the component developer, facilitating a wide range of possible load distribution strategies and to support QoS differentiation.

This chapter is structured as follows: Section 5.1 presents our model of load distribution, contains definitions of relevant concepts in the area of load distribution, and discusses the suitability of different load distribution methods; Section 5.2 uses the model and concepts defined in Section 5.1 to present, compare and evaluate the state-of-the-art in load distribution, with a focus on middleware-based load distribution; Section 5.3 proposes our load distribution mechanism; Section 5.4 presents a high level design for a Load Distribution Service that implements our mechanism; Section 5.5 presents the major conclusions of this chapter.

5.1 A Model and Overview of Load Distribution

As described in Chapter 3, the end-to-end performance depends on both processing resources and network resources. Even though research in the area of QoS for distributed applications has focused mostly on network QoS, for the overall end-to-end QoS the processing resources can and will become a more important bottleneck, especially for the operational interactions that we consider in this thesis [Cardellini02]. A reason for this is that networking capacity is improving faster than processing capacity. This chapter proposes a mechanism for the distribution of processing load for component-middleware-based applications in order to improve the performance.

Load distribution

The goal of *load distribution* is to execute a certain amount of workload on a set of processors subject to some optimizing criteria [Chapin96, Santos01].

The nature of the workload depends on the type of application, and can, e.g., be of set of tasks, entities, components, objects, invocations or transactions. Literature also uses other terms to indicate load distribution such as *load management* [Santos01] and *global scheduling* [Chapin96]. Global scheduling refers to the scheduling *among* nodes, contrary to *local* scheduling, which happens inside a node by the nucleus.

Load distribution model

Figure 5-1 shows a model of load distribution that we adopt in this thesis. It combines the concept of a feedback model, as presented in, e.g., [Bergmans00], with the model of a scheduler as presented in [Santos01]. The three activities we distinguish in this model are:

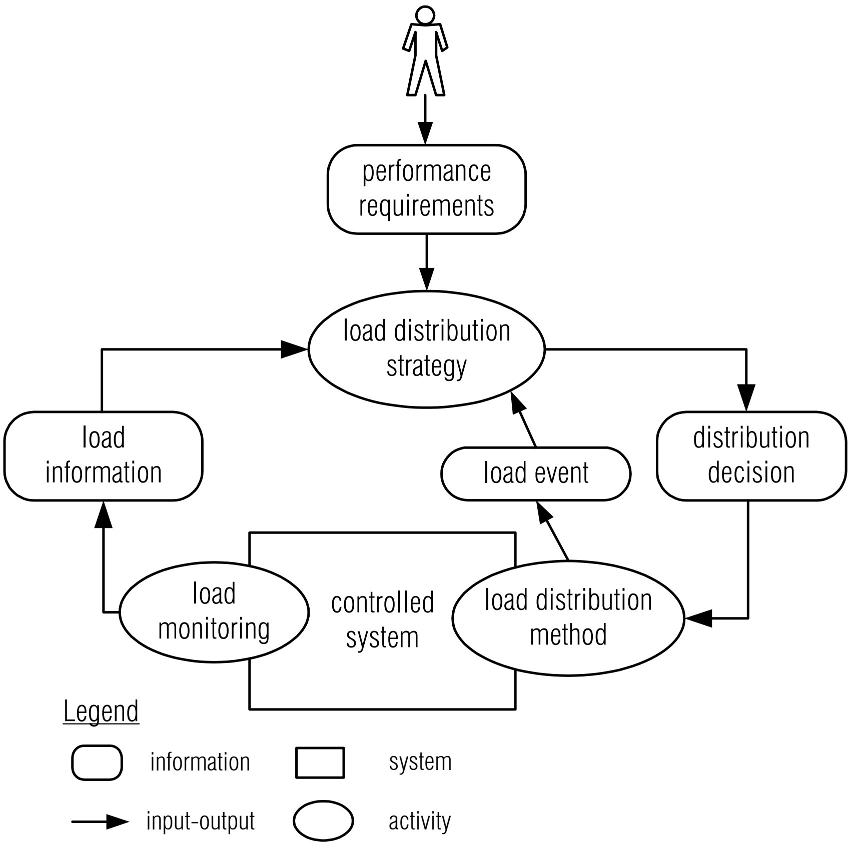
– *load monitoring functionality* - collects information on the current load of the controlled application. This load information is quantitative and dynamic, and indicates the amount of load at a certain moment in time. An example is CPU load.
– *load distribution strategy* - decides how to distribute the load, i.e., it contains some algorithm that tries to fulfill the performance requirements. An example is a least-loaded strategy that directs workload to the least loaded node. In literature this is sometimes also referred to as scheduler [Santos01], selection algorithm [Dahlin00, Casavant88], load evaluation [Linderm00], dispatching policy [Cardellini02] or controller [Bergmans00].
– *load distribution method* - actually distributes the load by executing the distribution decision made by the load distribution strategy. Examples of distribution methods are replication and migration. In literature this is sometimes also referred to as distribution mechanism [Schnekenb97], but we choose to name it distribution method since we use the term QoS mechanism to denote the complete load distribution functionality, including the load distribution strategy and load monitoring. Other terminology used in literature is actuator [Berman96], manipulation [Bergmans00] and effector [Santos01].

Load events

The load distribution strategy can receive *load events* if something happens to the controlled application that is relevant for the distribution of the load. Typical load events are the arrival of new workload and the completion of some workload. The events are created by and related to a distribution method. For example, a load distribution method may intercept every new invocation, and then notify the load distribution strategy through a load event so that the load strategy can subsequently decide which replica to use.

Contrary to load events, load monitoring is independent of any distribution method, and gives quantitative information on the load at a certain point in time.

Figure 5-1 A model of load distribution



Depending on the concrete load distribution solution, some parts of this model might not be used. For example, some load distribution solutions do not monitor the load, and some do not have load events.

In the following subsections we discuss the three activities more elaborately.

5.1.1 Load Monitoring Functionality

Load information is information used by the load distribution strategy to decide how to distribute the load. It is typically information on the current usage of the available resources. Load information should [Santos96]:

- Correlate well with workload response times. Since the load information is used by the load distribution strategy to make the distribution decisions, the load information has to correlate well with the actual performance of executing some workload (e.g., an invocation) at some particular resource.

- Be usable to predict the load in the near future, since the response time of a workload will be affected more by the future load than by the present load.
- Be relatively stable. Short fluctuations in the load should be discounted.

Application (in)dependent load information

Load information mentioned in literature [Shivaratri92, Santos96] includes CPU queue length, CPU utilisation, normalised response time, I/O queue length, memory utilisation, context-switch rate and application call rate. These are all *application independent load information*. It is less common to use *application dependent load information*, but for example [Santos01] and [Berman96] do advocate this. Examples of application dependent load information in a service provisioning platform with call control functionality are the number of active calls and the number of call setups in a certain time-interval.

5.1.2 Load Distribution Strategies

The load distribution strategy decides how to distribute the workload. It has as inputs (i) dynamic and quantitative load information on the status of the application, and (ii) events that occur in the controlled application that can be relevant for the load distribution, in particular the arrival of new workload. Examples of new workload are an arriving invocation or the creation of a new component. Depending on the type of load event and used distribution method, a load distribution strategy has to take a distribution decision when it receives a load event. For example, if case of an initial placement distribution method, the load distribution strategy decides where to create the component when it receives the corresponding load event.

Based on the load information, the events and the performance requirements, the load distribution strategy will decide how to distribute the load. *Figure 5-2* shows the part of the load distribution model (see *Figure 5-1*) that involves load distribution strategy.

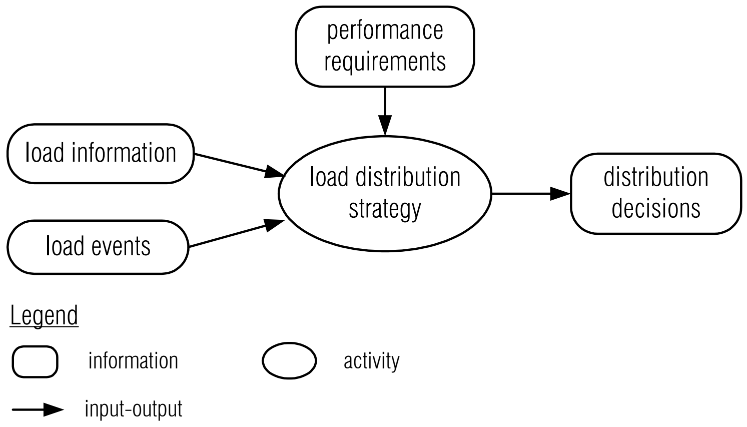


Figure 5-2 Load Distribution Strategy

Load Balancing, Load Sharing and Load Distribution

Load sharing

Load balancing

Two often-used load distribution strategies are load sharing and load balancing. *Load sharing* optimizes the total throughput of the application by keeping all resources busy. As long as there is work to be performed, all resources will be busy, independently of the amount of work actually assigned to each resource. Load sharing minimizing the time any resource is idle [Santos01, Shivaratri92]. *Load balancing* is a special case of load sharing that has as an additional goal to distribute the load evenly across all resources [Chapin96, Shivaratri92, Santos01]. This ensures a certain fairness, because all the workload will be given an even amount of resources which will result in less variations in response times.

Load distribution strategies in general however can have other optimization functions than those of load sharing and load balancing. For example minimizing costs, minimizing communications delay, prioritizing certain users [Chapin96] or QoS differentiations based on QoS requirements.

Static, Dynamic and Adaptive Strategies

Load distribution strategies can be broadly characterized as static, dynamic or adaptive [Shivaratri92, Casavant88]. *Figure 5-3* depicts this division.

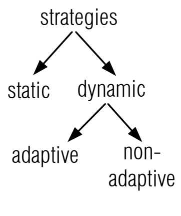


Figure 5-3 Division of load distribution strategies in static, dynamic and adaptive

Static strategies do not use information on the current state of the application when making distribution decisions. For example, workload can be assigned to a random node, or workload can be assigned to nodes in a cyclic way. Static strategies thus do not use load information.

Dynamic strategies, on the other hand, use load information that describes the state of the application. This means that the state of the application has to be collected in some way, which provides additional complexities and overhead. We will discuss load information below.

Adaptive strategies are a special case of dynamic strategies. An adaptive strategy can change its behavior to adapt to application state changes. For example, if a particular policy performs better for a specific application state, an adaptive strategy could change to that policy if it observes the required application state.

Scalability

Load distribution is a common means to increase the scalability of a application, e.g. [Cardellini02, Steen98, Othman01A]. We define *scalability* as the ability of a application to handle the addition of users and resources without users suffering a noticeable loss of performance. This definition is based on (numerical) scalability definition as found in [Neuman94]. Important for scalability is that a linear increase in available resources results in a linear increase in throughput, while maintaining a constant response time.

Scalability

Stale Load Information and Effective Scheduling

A problem with dynamic load distribution strategies is that the dynamic load information always describes the situation in the past since the collection and transportation of the load information takes time. The fact that load distribution strategies make their decision based on this *stale* load information can severely limit the effectiveness of the load distribution strategy, and can even cause instability. For example, in the very common least-loaded strategy, new workload is sent to the least loaded node. This can cause the so-called *thundering herd effect*, in which the least-loaded machine quickly becomes overloaded because of the large amount of new workload it receives until new load information is gathered [Dahlin00].

Thundering herd effect

The obvious approach to cope with stale load information is by frequently gathering the load information. However, a higher frequency of gathering load information causes more overhead. In addition, the load information will always be somewhat stale. A load distribution strategy therefore has to be able to perform effectively based on information available that is stale to some degree. [Dahlin00] proposes that load distribution strategies do not only base their load distribution decisions on the load information, but also on how stale this information is. The 'staleness' of load information depends on both the age of the load information, and an estimation of the rate at which new workload arrives that will change the load information. If the load information is fresh (as opposed to stale) because it is recent and/or the new workload arrival rate is small, workload is sent to a node with a low load. As the load information gets staler, the load distribution strategy behaves more as a random strategy.

Stability

Stability

We define *stability* as the ability of a load distribution strategy to detect when further actions will not improve the application state as defined by a user-defined objective [Casavant88].

Casavant identifies four sources of instability and their effects on the environment's behavior [Casavant88, Santos01]:

- *Intolerance instability* – if the distribution strategy reacts to small imbalances in load distribution, it may enter a state where tasks are continuously transferred among a given set of nodes to correct small load differences. This is called processor thrashing. To avoid this type of instability, the definition of optimal load distribution may be relaxed, allowing the application to tolerate small imbalances.
- *Overresponse instability* – this is caused by an attempt to respond too fast to local imbalance conditions. When an imbalance is detected, the application tries to transfer a large proportion of work to achieve an optimal load distribution as soon as possible. This may increase dramatically the number of transfers, causing instability and decreasing the distribution strategy's performance.
- *High static load instability* – if the load on the application increases, the opportunity for load imbalances and to overreactions becomes greater. This may lead to instability. Under conditions of heavy application load, the scheduler may do more harm than good with respect to application performance.
- *Invalid assumption instability* – this type of instability may occur when the load distribution algorithm violates certain assumptions made by the designer. It may happen, for example, if the application was designed to run on a certain environment and it is actually running on one with different characteristics. Or the designer assumed certain components to be co-located, and the load distribution strategy allocates them to different nodes.

[Casavant88] and [Santos01] argue that relatively simple policies can provide substantial performance gains, while more complex ones are not likely to offer much further improvements.

Division of a Strategy in Policies

A load distribution strategy can typically be decomposed in four logical components [Shivaratri92, Santos96], called policies:

- Information policy – to decide when, from where and what load information has to be collected.
- Transfer policy – to determine which resources to transfer workload to or from.
- Selection policy – to decide what workload can be transferred.

– Location policy – given that the sending or receiving resource is determined by transfer policy, to determine the respectively receiving or sending resource.

Information policy

The *information policy* decides when load information has to be collected, where that load information is collected, and what load information is collected. Information policies can be classified into three types, although hybrid policies are also possible:
– *Demand driven information policies* only collect information about the state of the application if an entity transfer has to be made. This means that the collection of load information is triggered by the transfer policy.
– *Periodic information policies* collect load information periodically. A fixed amount of overhead is introduced because information is collected whether it will be used or not. There is no extra delay when workload has to be transferred, because information has already been collected.
– In *state-change driven information policies*, the dissemination of load information is triggered by a change of the load to a certain degree.

Transfer policy

The *transfer policy* determines which nodes qualify to move load to, and which nodes qualify to move work from. Transfer policies may be based on thresholds, or may be relative transfer policies:
– *Threshold transfer policies* may decide that a source initiates a transfer of workload to a target (node) if the load of the source exceeds a certain threshold. It may also decide that a certain target initiates the transfer of workload from a source if its load falls below a certain threshold. Careful selection of these thresholds is necessary for a good performance of the load distribution strategy.
– A *relative transfer policy* considers the difference between the loads of targets (nodes) in a domain. If the loads differ more than a certain threshold, workload can be transferred.

A second and orthogonal division of transfer policies is into periodic and event-triggered policies. A *periodic transfer policy* periodically checks if the node's state qualifies for an entity transfer. Most transfer policies, however, are event-triggered. With *event-triggered transfer policies* entity transfers may be initiated because of a state change of other nodes, or because new workload originates at a source.

A third orthogonal characterization of transfer policies is the question who initiates the transfer of workload [Shivaratri92, Santos01]. If a heavily loaded node decides to transfer some workload, the strategy is said to be *sender-initiated*. If a lightly loaded node tries to get some workload from heavily loaded nodes, the strategy is said to be *receiver-initiated*. A *symmetrically* *initiated strategy* is also possible: both senders and receivers may initiate a workload transfer.

Selection policy

The *selection policy* is responsible for finding suitable workload to transfer. The suitability of workload can depend on certain rules or restrictions for this workload. The selection policy can take the following criteria into consideration:

- Workload transfer overheads must be minimized, i.e., the time and resources it takes to transfer the workload should be as small as possible;
- The transferred workload's execution time should be enough to justify the cost of the transfer;
- The dependency on local resources (e.g., file access, windowing application), should be minimal, since after transfer accessing these would have to be done remote. Dependency on local resources can also prohibit transfer if the resources cannot be accessed remotely.

A simple approach for a selection policy is to consider newly created tasks or entities to transfer, before they start executing. They are then much cheaper and easier to transfer.

Location policy

The *location policy* tries to find a suitable transfer partner when some workload has been selected for transfer, and the transfer policy determined the receiving or sending resource. This could be done using the state of the application (dynamic strategies), but it can, e.g., also be done in a random way (static strategies).

5.1.3 Load Distribution Methods

In this section, we discuss distribution methods, and in particular how we can classify them in the context of component middleware. Load distribution method is in literature sometimes also referred to as distribution mechanism [Schnekenb97], actuator [Berman96], manipulation [Bergmans00] and effector [Santos01].

Historically most work done in the area of distribution methods is on assigning processes (e.g., [Nasika00, Boyd02, Milojicic00]) or tasks (e.g. [Chapin96, Santos01, Foster00]) to nodes.

Task-based approaches

We cannot however apply this task-based research directly to component-middleware because of its very different computational model. E.g., in task-based approaches every task represents a typically large amount of computation that is done at some node, without any communication during the computation with other tasks. The tasks ends when the computation is finished. This differs from our domain of large-scale distributed applications that exchange a lot of invocations that typically require little computation per invocation.

Process-based approaches

The process-based research cannot be used because migrating a process does not preserve the validity of component references [Nasiki00, Boyd02]. In addition, for a process-based approach:

- The amount of state that has to be transferred is quite large, making it expensive [Milojicic00].
- This requires support from the operating application, which current commercial operating applications do not provide [Milojicic00].
- A process can contain a lot of components, making this a too coarse grained solution. In particular, we could not differentiate between QoS of the components that live within the process.

Replication based and Non-replication based Distribution Methods

Replication

Replication is a well-known load distribution method [Cardellini02, Neuman94]. In the context of component middleware, replication refers to the usage of replicas, which are multiple components instantiated from the same component template with a single identity. Replicas can logically be thought of as one single component [Neuman94, RMODPPart3], i.e., they exhibit the same behavior and have a consistent state. Each replica is assigned to a different node, and invocations are directed to one of the replicas. The decision which replica to use can be made with different granularities, in particular per invocation or per session. Replication is often used to improve availability, e.g., to create redundancy to cope with node or network failures, as is the case with Fault Tolerant CORBA [FTCORBA].

Non-replication

Non-replication means that we only use a single instance of a component. In contrast with replication, the invocations or sessions are always allocated to the same server component. The load is distributed by distributing the components over the set of available processing nodes. Different server components of the same type will have different identities and different states, but can be instantiated from the same template. In a *non-replicated distribution method,* components can be allocated to a specific node at creation time, and/or can be migrated to another node during run-time.

Pre-emptive and non-preemptive distribution methods

Distribution methods are commonly divided into preemptive and non-preemptive [Shivaratri92]:

Non-preemptive

- **Non-preemptive distribution method:** the workload is assigned to a target immediately after it has originated at a source. Once workload has been assigned to a target, it cannot be transferred to another target.

Preemptive

- **Preemptive distribution methods:** the workload can be migrated to another target after it has been assigned to its initial target.

Preemptive distribution methods introduce additional complexities because it must be possible to migrate the state of a workload to a new target. This makes a preemptive approach not only more complex to implement, but also potentially less transparent for the application developer. In addition, migration can cause a significant performance overhead. Studies show that the performance of preemptive and non-preemptive strategies are largely application dependent [Santos01].

Combining load distribution methods

Table 5-1 combines the above-mentioned dimensions for classifying load distribution methods. Non-preemptive and preemptive replication-based distribution can be done at different granularity levels: per invocation and per session. However, we dismiss preemptive per invocation replication since it is too intrusive to the application to access the application state with ongoing invocations (see Chapter 4). A preemptive replication method with session granularity migrates a session to another replica.

In the case of a non-replicated approach, a preemptive approach can change the location of a component during its lifetime, i.e., a component can be *migrated* to another location. Important in this case is preserving the correctness, which includes migrating the application state and updating the component references.

Table 5-1 Non-preemptive and preemptive distribution methods.

	replication		**non-replication**
	session	invocation	
non-preemptive	per-session	per-invocation	initial placement
preemptive	session-migration	-	migration

A specific load distribution solution can combine several of the above distribution methods. For example, a load distribution solution might initially create a set of replicas based on the load at that point in time. It then assigns invocations to the least-loaded replica, and migrates replicas that execute on a node that is consistently overloaded to another node.

5.1.4 Replication and State Consistency

In principle, replicas have to be consistent, i.e., every change to the state of one replica has to be applied simultaneously to all other replicas. This can be done by either processing every invocation in all replicas, or by processing it in one replica and sending the resulting state update to all other replicas. This however makes little sense if performance improvement is the goal of the replication. Replication is therefore only useful from a performance perspective if we can relax the consistency requirement. We will refer to 'real' consistency as *absolute consistency*, and to a relaxed consistency as *loose consistency* [Neuman94]. An example of loose consistency is when it is sufficient that replicas will eventually receive state updates. Whether this is acceptable depends on the specific application.

Absolute and loose consistency

There are many possible forms of loose consistency. Examples are update-propagation (every replica eventually receives every state update), and reads-your-writes (updates made by a specific client will be visible for that client the next time the client does a read) [Chockler00]. Most of them however have the disadvantage that they violate the transparency principle, i.e., they require the application developer to be exposed to the complexity of the consistency mechanisms that are used, and how they impact the application. We therefore limit ourselves here to only one form of loose consistency that does not violate transparency, which is replication for components that only have *session state*. In this case a component has state within a session with a specific client, but other sessions do not depend on or affect this state. By making sure that subsequent invocations that are part of the same session are handled by the same replica, this state can be kept by this replica and does not have to be propagated to the other replicas. This allows a straightforward and efficient implementation.

5.1.5 Suitability of Distribution Methods

We can divide components in three types based on the state that is maintained. In general, we must assume that a component has state, that this state is shared between concurrent sessions, and that the response to any incoming invocation may depend on any other previous or simultaneous invocation. We refer to this as *global state*. In more specific cases a component has only the before mentioned *session state*, i.e., it maintains state within one session and a response to an incoming invocation only depends on other invocations within the same session. Finally, a component can be *stateless*, i.e., no state is maintained between invocations and any response is independent of any other invocation. *Table 5-2* indicates for each of these three types of components the suitability of the different distribution methods.

Table 5-2 Suitability of distribution methods based on type of component

Distribution Methods / Type of Component	Initial Placement	Migration	Per-session Replication	Session-migration Replication	Per-invocation Replication
Stateless – no state between invocations	+	+	+	+	+
Session State – state between invocations in the same session	+	+	+	+	-
Global State – state between invocations	+	+	-	-	-

Replication Routing

Replication-based distribution methods, regardless of how they maintain consistency, can be divided in three types based on how they transport invocations from the client to the selected replica. This division is based on [Cardellini02], but since the original division is specific for web servers, we generalized it somewhat:

– *central invocation-switch* – all invocations for a certain replicated component pass through a centralized piece of functionality. This is depicted in *Figure 5-4*, in which client component C invokes server component S, and this invocation passes to an invocation switch that assigns it to replica R2. In the web server domain this is called a web-switch approach for cluster based web server replication [Cardellini02]. A limitation of a central invocation-switch is that all replicas have to be geographically co-located in order to prevent long network delays. In addition, the central switch is a potential bottleneck and single point of failure. It is possible to make a further subdivision into one-way architectures that send responses directly from replica to client, and two-way architectures in which the responses also pass through the central-invocation switch.

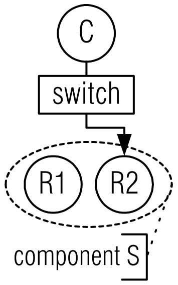


Figure 5-4 Central invocation-switch based replication.

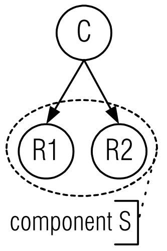


Figure 5-5 Replication and multi-casting

– *multi-casting* – all invocations are multi-casted to all the replicas. This is depicted in *Figure 5-5*, which shows client component C multicasting an invocation to all replicas. Benefit of this approach is that there is no central point of failure or potential bottleneck, as is the case for the central invocation-switch. This is referred to as a virtual web cluster in the webserver domain [Cardellini02]. A limitation is that the replicas typically have to be geographically co-located to accomplish multi-casting with minimal bandwidth consumption. The actual multi-casting can then be done close to the server using IP layer or below IP layer facilities. The approach must guarantee that one replica and only one replica processes the invocation, and responds to the client. This requires either a fixed/static algorithm, which reduces flexibility, or coordination between replicas, which causes processing overhead and delay.

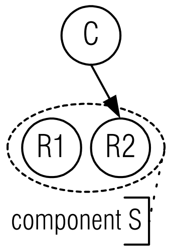


Figure 5-6 Replication and client middleware visible routing

– *client middleware visible routing*, invocations are directly routed to a specific replica. This is depicted in *Figure 5-6*, in which the invocation is directly routed to one of the replicas. In this approach the replica address is visible to the client It is essential not to violate the replication transparency, which can be done by making the (client-side) middleware responsible for the routing to a specific replica. This approach is similar to distributed web applications that use DNS, or standardized HTTP redirects. Benefits are that the replicas can be geographically distributed, and there is no central functionality that all invocations have to pass through. A disadvantage is that it can cause extra overhead to inform the client which replica to use, which is typically done as part of the first invocation of a session. A second disadvantage is that to be able to redirect a client to another replica during a session is established can again cause extra overhead. Due to the overhead involved with using client middleware visible routing for the first invocation, it does not make sense for per-invocation replication.

Content Blind versus Content Aware

For per invocation replication the selection of replicas can be content-blind or content aware [Cardellini02]. In case of content blind approaches, the distribution method does not inspect the content of the requests. For content-aware approaches the distribution method does inspect the content of a request, or the identity of the clients to decide which replica to use. Content aware approaches cause more overhead but allow more intelligent load distribution strategies.

Caching and Replication

Caching

A special type of replication is caching. *Caching* is a temporary form of replication, in which an entity or response is saved in a cache that is located at or close to a requesting node (this definition is based on [Neuman94]).

In some cases, e.g., in the case of web caching, only the response is cached and not execution logic that generated the response. *Caching of responses* does not fit very well with component-middleware-based application, in which a response is usually generated based on the parameters of the request, and the current state. We can therefore dismiss caching responses for load distribution in component-middleware-based applications. In the case of *caching of entities* however, we considered the cached entity a replica that is created close to the clients. The creation is typically done on-demand at the first invocation.

Like normal replication, caching can improve performance and scalability because the actual (replicated) server gets fewer invocations. And caching improves response time because the network latency is less. As is the case for normal replication, caches have to employ a validation technique to verify consistency between cached entity and the server, i.e., the cached entity is a replica and both the state and execution logic have to be consistent with the other replicas.

5.2 State-of-the-Art in Load Distribution

This section describes the state-of-the-art in load distribution for middleware-based applications. This includes approaches in commercial middleware products, for as far as we could find a publicly available technical description of the load distribution functionality.

Although there is also work on load distribution for non-middleware-based systems, we do not include this work in this state-or-the-art overview since we cannot directly apply it to middleware-based application, as explained in Section 5.1.3. We limit ourselves here to some pointers to approaches for non-middleware-based systems:

– Task-based load distribution – [Santos01] focuses on the usage of stochastic models to increase effectiveness of application level scheduling, in particular on how to deal with incompleteness and inaccuracy of load information.
– Process-based load distribution – [Nasika00] describes work in process migration that takes interactions between distributed processes into account. The follow-up paper [Boyd02] explicitly mentions that their work is not suitable for distributed object applications.
– Web server load distribution – [Cardellini02] provides a survey on the usage of replication to increase web server scalability. Web server replication is similar to replication for middleware-based applications because both consider client-server connections that consist of a series of interactions between the client and the server. Current research in web server scalability often assumes that web server interactions are

stateless, and with static content. Replication of web servers that have stateful sessions and require significant processing is considered future work [Cardellini02].

5.2.1 Middleware-based Load Distribution

This section describes the state-of-the-art in middleware-based load distribution.

Schnekenburger et al.

Schnekenburger's approach [Schnekenb96, Schnekenb97] is based on IONA's CORBA ORB Orbix, and relies on the CORBA Trading Service. It implements per-session replication and migration.

In the case of replication, the client uses the trader to select an object (replica) to invoke. Changing to another object is possible, but not without explicit involvement of the client (it requires a new import to the trader). State synchronization or state migration issues are not dealt with, effectively making it only possible to migrate sessions in case of stateless components.

Migration of objects to a node with a lesser load can be done periodically. This is not transparent for the client since it invalidates the object reference. When the client receives an exception after the server object has been migrated, it has to contact the trader for an updated object reference. Issues concerning reconfiguration safe state or consistency guarantees in general are not mentioned, and are probably not supported.

The load information consists of the average percentage of time a CPU is busy.

Barth et al.

[Barth99] describes a load distribution service in CORBA that is based on using the Naming Service. An existing UNIX based resource management application called WINNER is used to collect load information. Several objects are bound to one name in the Naming Service, and a resolve on that name will return the object on the least loaded node. Since from the perspective of the application logic, it will randomly get one of the objects bound to this name, we classify this as replication. Since there is no concept of changing to another replica during a session, and no state synchronization between replicas, this is a per-session replication for components with session state (or stateless components).

The load distribution strategy uses a combination of CPU queue length and percentage of CPU idle time. The paper argues that CPU queue length can be misleading in case of many shortly running processes, but that if the percentage idle time is approaching zero the queue length has to be taken into account.

A benefit of this approach, and all Naming Service based approaches, is that it is transparent for the client and the server. It is however limited to applications that use the Naming Service for every new session, and it can only change the load distribution for new sessions.

Friends

[Man00A] describes an approach that uses initial placement to balance load in a CORBA-based service platform called Friends. The load distribution strategy uses CPU load information, and combines this with application specific information that is hard-coded in the strategy. The initial placement method is implemented through a centralized factory.

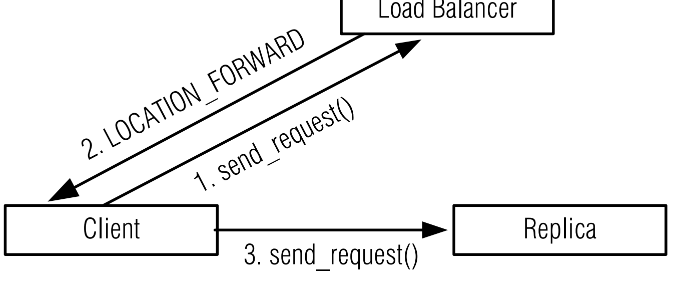
Othman et al.

[Othman01A] and [Othman01B] describe a CORBA-based replication approach. It provides per-request, per-session and session-migration. Client-side transparency is provided by using a location agent type of solution, combined with the standard CORBA LOCATION_FORWARD mechanism.

Per-request approach uses a central switch to distribute requests over the different replicas. Othman et al. claim that per-request incurs too much overhead to be of practical use.

Figure 5-7 shows how per-session load balancing works. A client initially receives a reference to a load balancer. When a client makes a request, the load balancer redirects the request to the appropriate target server replica using a LOCATION_FORWARD reply. The client will continue to use the object reference from the LOCATION_FORWARD message to interact with the server.

Figure 5-7 Per-session distribution by Othman et al.



This approach advocates session-migration (called on-demand by Othman et al.) as the preferred approach for load distribution. Reason for this is that session-migration does not introduce the overhead in the per-request approach, but still allows for distribution during a session.

Measurements for a test setup in which objects are stateless and provide a service that runs a relatively long time show that latency and throughput change only little when using on-demand and per-session assignment. Using per-request assignment, latency doubles, while throughput decreases fifty percent, thus the overhead for per-request is too high.

[Othman01A] compares a static per-session strategy to a dynamic session-migration strategy. In the used test scenario, the static strategy is not able to balance the load over the available replicas. In the case of the dynamic strategy the load across the replicas fluctuates for a short period of time, after which it stabilizes, distributing the load evenly across the available replicas.

As [Othman01C] mentions, this approach does not provide any facilities for state synchronization, thus limiting its use to stateless objects. [Othman01C] does mention the possibility to use the state access methods to synchronize state, or to use the CORBA Persistent State service for this [CORBAPSS]. It however does not properly address the consistency issues, and does not address the performance penalties associated with state synchronization. Summarizing, this approach can only provide load balancing for stateless objects.

Badidi et al.

Badidi et al.'s approach [Badidi99] implements a CORBA load sharing service. It does this by using the standardized CORBA Trading Service to implement a straightforward replication approach comparable to Barth et al. Per-session replication is implemented by the Trading Service, which gives a client the object reference of an existing server (replica) that is lightly loaded. There is no state synchronization, limiting per-session replication to stateless components, and components with session state.

This approach differs from Barth et al. with the possibility to do per-request replication. This is implemented through smart proxies, a proprietary way to add functionality at the client side in the invocation path for every invocation. These smart proxies are used for every subsequent invocation, to make it possible to redirect subsequent invocations to another server. The paper does not contain details on how the smart proxies are aware of load of each replica, and which replicas exist. Since state transfer issues are not dealt with, this approach allows per-request replication for stateless objects only.

Three strategies are possible, round-robin, random and least loaded. The load of a server is defined by the server utilization, i.e., the amount of time the server is busy handling requests during 1 second of time. This is measured by manually instrumenting server code to measure response time.

Thissen et al.

Thissen et al. [Thissen00] use the CORBA Trading Service [CORBATS] to balance the load for a service that is replicated over several nodes. Each Trading service is instrumented with load measuring functionality to collect load information for the node it runs in. When a client imports a service, the Trader searches its service directory for suitable services. The Trader returns a list of services that fulfill the client's import request, together with a quality score for every service that characterizes the degree of fulfillment to the client's request. This list is ordered by the load balancer based on the load of the system (or is random), and allows the client to import the service that is, e.g., least-loaded.

The approach was evaluated by performing measurements in several different setups, measuring the mean response time of the servers from the client's perspective. A random strategy proved to be inadequate in most situations, especially when loads are relatively high. When the servers are homogeneous, i.e., the server hosts have equal hardware characteristics, different load balance strategies that use the load of the system have roughly the same performance. However, when the servers are heterogeneous the strategies do vary in performance.

Lindermeier

Lindermeier's approach [Linderm00] combines initial placement, migration and per-session replication in CORBA. A central load evaluation component is responsible for selecting a host during object creation, redirecting requests to migrated objects, and assigning requests to replicas in case of replication. Replication is supported for *replication safe* objects only, which means that there is no need for consistency protocols between the replicas.

The load evaluation component, which resides at the implementation repository, is responsible for selecting a node for initial placement, and for deciding when replication or migration of an object should take place. The load evaluation component gets load information from nodes and middleware by using the Simple Network Management Protocol (SNMP). Load information to represent the load for a certain host is the processor utilization, amount and used memory and network capacity and utilization. The load evaluation component gets load information on objects by querying the ORB and POAs. Load information gathered this way is request rate, waiting time of requests to be processed, processing time of requests, waiting time for sending requests, data volume of requests and what are common communication partners.

State transfer can be achieved by using the common state access methods, and the standard CORBA request forwarding method is used to (re)direct clients to the appropriate location. The issue of save state is mentioned, but no solution is described on how to obtain this. Also the issue of state synchronization between replicas is mentioned but not dealt with.

This approach implements initial placement, migration and per-session replication, and discusses the state issues related with these load distribution methods. It however does not solve all these issues, and the way this approach is implemented violates some of our requirements. Specifically:
- This approach requires changes to the POA interface, which is a standardized interface and thus reduced transparency for the server object developer.
- This approach requires changes to the ORB implementation.
- The consistency in case of migration is not dealt with.
- The consistency between replicas is not dealt with, limiting this approach to stateless objects.

Also details on how the load information is obtained and used are not mentioned in the paper.

Orbix

Orbix [Orbix] provides per-session replication for stateless objects using the Naming Service. Object groups consisting of multiple objects can be bound to a single path in the naming service. When a client resolves a path that an object group is bound to, the naming service returns one member object of the group based on the selection policy. This selection policy can be static (round-robin or random) or active. The active policy selects the object with the least load. The load for each replica is not collected by Orbix, but has to be set manually after which it remains valid for a certain period. Since all new clients will be directed to the least-loaded replica, this seems to be a risk for thundering herd effects.

COM+

Load distribution in COM+ (called Component Load Balancing [Carter99]) is done by the activation of object implementations at a specific target location. This is the same as assignment of an object implementation to a server at instantiation time. After the object implementation has been assigned to a server, it cannot be migrated. Assignment of requests is not used for CLB.

When a client activates an object, the request is passed to a (centralized) load balancing server. This server keeps a list of machines that can create an instance of the object implementation, and selects one that will receive the activation request. The target machine then returns an interface pointer directly to the client. The client then continues to make invocations directly on the object implementation, until the client is finished with the object.

The machines that participate in the load balancing are regularly polled by the load balancing server for their response time (information policy). The load balancing server sorts the list of target machines according to load, and then selects targets from the list in a round robin fashion until the next response time poll (selection policy).

One problem with this approach is the use of a centralized load balancing server. If many object implementation activations take place, the load balancing server can be overloaded. It also introduces a single point of failure. Another problem is that once a client is bound to an object on a particular target, it cannot be rebound.

WebLogic

EJB itself provides no explicit support for load distribution. However, several commercial EJB container implementations do provide load distribution support. We describe here how BEA WebLogic 6.1 [Weblogic] supports load distribution by clustering of server objects.

WebLogic provides load distribution through replica-aware stubs. These replica-aware stubs can exist on two level: for a component home, and for a EJB object.

A replica-aware stub for a component home supports initial placement since the creation request for a specific object is routed to a component home that is selected by the load distribution strategy.

For stateless session beans, the replica-aware object stub provides per-request assignment for client requests. For stateful session beans, the replica-aware object stub routes always to the primary object, and only to another replica (secondary object) in case of a failure. So replication is supported for stateful components, but not for the purpose of load distribution.

For entity beans that are read-only, the replica-aware object stub does per-invocation replication. For entity-beans that also support writes, all invocations are always routed to the same object.

WebLogic provides only static load distribution strategies, viz., round-robin, weight-based and random. It is also possible to provide custom strategies via the so-called call router that decide which replica to use. These call routers become part of the replica-aware stub, and thus execute at the client side.

Summarizing, WebLogic provides initial placement for all types of beans, and per-invocation replication for stateless beans.

Globe

Globe [Steen98] is a research middleware that aims at building scalable applications in a wide-area network. Globe is not based on common middleware technologies.

In Globe an object is a *distributed object* in that an object can be distributed over several nodes in any way the developer sees fit. This contrary to the more common computational model that mandates that an object is located on one node, and generated stubs are co-located with every client (in the same capsule). Common middleware technologies do provide interceptors (as described in Chapter 3), which also allows functionality co-located with every client beyond the generated stubs, but this is more limited than what is possible in Globe.

The *distributed object* concept in Globe can be used to implement different types of replication. Where in most replication approaches the whole object implementation and state are replicated, Globe makes it easier to replicate only part of the state and implementation. The different parts that combined are the distributed object can use pre-defined or custom consistency policies, which can be more efficient than absolute consistency.

Absolute consistency is considered to cause too much overhead to be useful for scalability purposes. [Steen98] therefore advocates more efficient and 'looser' consistency, which can depend on specific application requirements. Instead of a one-size-fits-all approach, an application developer can determine an application specific replication strategy.

Recent papers on Globe focus on Globule (e.g., [Pierre01, Pierre02], which applies Globe to on-demand replication of web documents. However, [Jansen01] describes how to use Globe for remote object middleware. Globe is integrated with a remote object middleware application named CORE that, according to the paper, resembles object middleware technologies such as CORBA. One of the main differences is that with CORE the proxies are downloaded by clients, contrary to being generated at development time, as is typically done with CORBA. [Jansen01] describes how CORE is extended to allow an object to be distributed over several machines using Globe. In the extended CORE an application developer can implement customized proxies that do more than the simple remote-access type of proxies that the normal CORE offers.

Concluding, Globe incorporates very interesting ideas on replication and scalability, but assumes a different computational model than the common middleware technologies we consider in this thesis. Compared to common middleware such as CORBA, Globe sacrifices transparency in favor of flexibility. This violates our common middleware requirement (see Chapter 3, generic requirements for QoS mechanisms). In addition, the extra flexibility that Globe offers has as a disadvantage a decrease in transparency. This violates our time and expertise requirements.

Similar ideas on application dependent consistency as advocated in [Steen98] are applied in Cascade, a CORBA-based caching service that we will discuss next.

CASCADE

CASCADE [Chockler00] is a caching service for CORBA objects. CASCADE provides several levels of consistency guarantees for keeping replicas of objects consistent. Replicas are cached close to the client to improve the network latency when accessing an object.

The approach is based on a hierarchy of logical caching domains, consisting of several servers (called Domain Caching Servers). In practice, a domain represents a geographical location. It is assumed that network latency in a domain is low, that the network latencies between domains are high and that servers are available for caching in every domain. The root of the hierarchy is the original location of the object. When a client makes a request for a service, the object is copied from the closest domain in the hierarchy. Different policies can be set for object consistency requirements at a per-request granularity, and the caching service makes sure these requirements are met.

The following six levels of consistency conditions are defined:

- *Update Propagation* – Each update on an object is eventually received by each DCS.
- *Read Your Writes* – The effects of an update an application makes on an object are visible to all subsequent queries of that application.
- *Monotonic Reads* – The effects of every update seen by an application query are visible to all subsequent queries of this application (unless overwritten by later updates).
- *Monotonic Writes* – Two updates from the same application are applied in the same order as issued.
- *Writes Follow Reads* – Updates whose effects are seen by an application query are applied before all subsequent updates issued by this application. Along with Monotonic Writes, this ensures causal ordering of updates.
- *Total Ordering* – All updates are applied in the same order (by all Domain Caching Servers). In other words, there is a global sequence of updates. Total Ordering includes Writes Follow Reads.

The total ordering is the most strict consistency condition, and can be used for the widest variety of application. However, total ordering also imposes the most overhead. It is implemented by ascending all updates to the root, which orders them in a sequence and applies them to itself. After having done so, the root propagates an update downwards by either propagating the request itself, or by propagating the resulting version of the object. The latter is the only option in CASCADE if external objects are affected (nested invocations) according to [Chockler00]. Although not clearly stated, this is probably because if a nested invocation would be processed by each replica, the nested invocation would be duplicated. This is a general issue for replicated systems in which requests are processed by several or all replicas, and probably could be suppressed in a similar way to how this is done for active replication for Fault Tolerance [FTCORBA]. There is however a performance penalty for this which might prove to be unacceptable for most applications.

The paper mentions that sometimes a client has to provide consistency information, e.g., the last update seen by client. This is then done through either a special interface, violating client transparency, or it can be done through an interceptor. Details on this are not provided.

CASCADE is geared towards reducing response time in case of slow network connections, not on increasing throughput. Applications with lots of read-only operations can benefit most from CASCADE. But depending on the consistency required, throughput and even response time can actually degrade.

The application developer is responsible for setting the policies in such a way that the consistency is as 'loose' as possible, while not actually violating application consistency. This requires expertise from the application developer, violating our expertise requirement (see Chapter 3). A lot of applications require total ordering, in which case CASCADE could just increase response time instead of decreasing it (depending probably on the ratio read/writes). The application would have to be designed to prevent this, which violates the transparency.

However, in cases where there are servers available in the same domain as where clients are located, and the application is suitable for caching, CASCADE can increase performance while hiding the complexity of the consistency protocols for the application developer.

5.2.2 Observations

Historically most work done in the area of load distribution is done on scheduling of processes and tasks to different nodes, and more recently on load distribution for web servers. We cannot however apply this research directly to component-middleware-based application because of (i) the very different computational model we assume, and (ii) we want to be able to differentiate QoS based on the client or session, which requires a fine granularity which is not available at the network or operating application level.

Supported Load Distribution Strategies and Load Information

Most middleware-based load distribution approaches focus on load distribution methods, and apply simple least-loaded load balancing strategies that use CPU measurements as load information. To our knowledge, there are no approaches that thoroughly compare and evaluate alternative strategies and/or usage of different load information. All approaches have one or only a small number of fixed strategies, based on one or limited set of load information solutions. None of them allow addition of custom strategies, or custom load meters.

A problem with comparing the different approaches is that each approach evaluates its solution using very different test scenarios. The test scenarios differ in aspects such as:

- the amount of servers (few, many);
- the amount of client (few, many);
- the amount of clients per server;
- whether they have nested invocations or not;
- whether the servers have state or not;
- the frequency of state changes;
- whether the objects are long or short lived;
- whether processing an invocation takes a lot of processing power, or very little processing power, or varying processing power;
- whether they assume a homogenous environment with no other processes that consume resources, or a heterogeneous environment with other processes consuming varying amounts of resources;
- whether client and servers are located in the same LAN (high bandwidth and low latency), or are connected via a slow network connection.

All these aspects can be important for determining the most appropriate strategy, load information and distribution method for a specific scenario.

Initial Placement

Initial placement of components is implemented by several approaches [Man00A, Carter99, WebLogic, Linderm00]. [Man00A] and [Linderm00] describe approaches that are CORBA based, and use the factory pattern in which a component is instantiated via a central factory. Here, the central factory delegates the instantiation to a node that is selected by the load strategy, similar to our approach. [Carter99] describes Component Load Balancing for COM+, which provides an initial placement distribution method that activates object implementations via a (centralized) load-balancing server. WebLogic [WebLogic] is an Enterprise Java Beans (EJB) server that supports initial placement through its so-called replica-aware home stubs. These (client-side) stubs are downloaded by the client during run-time, and contain a list of EJB Home objects. The client stub will select one Home object when the clients want to create or find an EJB.

Migration

Only a few approaches offer migration of components [Schnekenb97, Linderm00]. [Linderm00] describes transparent redirecting of a client to a new location of a migrated object by exploiting the request forwarding mechanism that is part of the CORBA specification [CORBA, Almeida01C]. [Schnekenb97] describes a CORBA-based approach that uses the Trader Service. The transparency of this approach is limited, since after each migration the client has to get the new object reference from the Trader Service. None of the approaches consider all the correctness aspects related to migrating components.

Replication

Replication is offered by quite some approaches [Othman01A, Badidi99, Schnekenb97, WebLogic, Orbix, Thissen00], but none of these approaches is suitable for stateful components. Stateless components and sometimes components with session state are supported.

Quite some approaches use session-based granularity by using a Directory Service type of solution: [Schnekenb97], [Badidi99] and [Thissen00] describe the use of the CORBA Trader Service, and [Barth99] and [Orbix] describe the use of the CORBA Naming Service. Only [Barth99] and [Badidi99] mention the possibility to switch to another replica during the session, without considering state issues.

[Othman01A, Othman01B, Othman01C] and [Linderm00] also describe approaches for replication of CORBA-based applications, but they use the more advanced CORBA request forwarding mechanisms to implement (re)direction. The approach described in [Othman01A] supports session-migration replication for stateless components. [Othman01B] does mention the possibility to use the state access methods to synchronize state, or to use the CORBA Persistent State service [CORBAPSS] for this. It however does not actually propose a solution, or sufficiently addresses the consistency issues and performance penalties. [Linderm00] describes how to offer replication for stateless components for both per-invocation and per-session granularity. Since the per-invocation replication is also based on the request forwarding mechanism, this causes considerable overhead. A criticism towards [Linderm00] is that it not only requires changes to the ORB implementation, but also to the CORBA standard.

WebLogic [WebLogic] supports per-invocation and per-session replication for stateless session beans and read-only entity beans only through so-called replication-aware EJB Object stubs, in which the locations of the replicated (stateless) beans are embedded.

None of the above approaches allows session migration for component with session state.

5.3 A New Load Distribution Mechanism

This section describes our load distribution mechanism for component-middleware-based applications. Our mechanism addresses the generic requirements for QoS mechanisms (as identified in Chapter 3) and specific requirements for load distribution, which we will identify in this section.

This section is further structured as follows: Section 5.3.1 motivates the need for a new mechanism for load distribution of component-middleware-based applications, Section 5.3.2 states the specific requirements for such a mechanism, Section 5.3.3 gives an overview of our mechanism; Sections 5.3.4, 5.3.5 and 5.3.4 respectively elaborate on the supported distribution methods, the load strategies and the load monitoring functionality. Section 5.3.7 describes how we differentiate QoS in our mechanism. Finally, Section 5.3.8 discusses the limitations of our mechanism and compared to the approaches found in the literature.

5.3.1 Motivation

Load distribution for component-middleware-based applications can be done at different layers:
– application layer
– middleware layer
– resource layer

Solutions at the resource layer are process-based solutions that migrate or replicate processes, and network-layer-based solutions that distribute invocations based on IP address or DNS name. Although they offer a generic solution to load distribution, these solutions are too coarse grained and do not allow QoS differentiation. The reason for this is that the granularity of such a solution is a process or node, and these contain a lot of components, and these solutions cannot distinguish between invocations for the different components, or migrate a specific component.

Solutions at the application layer are specific for a certain application, and burden therefore the application developer. They violate the flexibility, time and expertise requirements (see Chapter 3, generic requirements for QoS mechanisms).

We propose a middleware-based solution here. At the middleware layer there is more information on the application available than on the resource layer, such as information on the client, the target component, the method name etc. This allows more intelligent load distribution strategies, and allows QoS differentiation. At the middleware layer the load distribution can still be solved in a generic and transparent manner, i.e., without burdening the developer of client or server components.

Existing middleware-based approaches offer point solutions that target specific types of applications, often without making this explicit (see Section 5.2), do not sufficiently consider state synchronization issues, do not sufficiently consider correctness issues for migrations and/or violate transparency.

5.3.2 Requirements

We consider the following requirements for our load distribution QoS mechanism, in addition to the generic requirements for QoS mechanisms as identified in Chapter 3:

1. *Not application type specific* – the load distribution QoS mechanism should not be specific for certain application type, e.g., be specific for applications with long running invocations, or for short lived components, or for processor bound applications, or for non-nested invocations. This extends generality requirement (see Chapter 3), which states that a QoS mechanism should not be specific for a certain application.
2. *Node heterogeneity* – the load distribution QoS mechanism should allow different types of nodes, e.g., with different operating applications, or with different levels of processing power. Part of this requirement is the ability of the mechanism to function effectively in an environment with nodes with different performance characteristics, e.g., the load distribution strategy should function properly with one slow and one fast node. This requirement extends the heterogeneity requirement (see Chapter 3).
3. *Multiple geographic locations* – the load distribution QoS mechanisms should be able to function properly with nodes at different geographic locations. The load distribution QoS mechanism has to be able to function properly with a slow network connection, not only between the clients and the servers, but also with a slow network connection between the servers.
4. *Quality of Service* – the load distribution QoS mechanism should provide functionality that allows enforcement of QoS requirements, i.e., be

suitable for QoS differentiation contrary to be limited to load sharing or load balancing.

5. *Minimize overhead* – the load distribution QoS mechanisms should cause minimal overhead. Some overhead is unavoidable, for example for load monitoring, transportation of load information, generation of load events and state synchronization.

5.3.3 A Framework-based Mechanism

An optimal load distribution strategy, and the load information that is needed for this, depends on characteristics of the application and the environment in which it operates. This means there is an inherent trade off for load distribution between being generic and optimized for certain applications and environments. We therefore propose a framework-based mechanism that offers a wide range of often used distribution methods, load information and load distribution strategies, and that can easily be extended with new strategies or load information types.

As an example, an application that creates new components that are very short lived and uses a lot of CPU would typically benefit most from a combination of initial placement as the distribution method, and CPU load information. The frequency of gathering load information has to be balanced with the relative stability of this information (as explained in Section 5.1.2). On the other hand, for an application with a large number of long running components that cause little CPU load per component, migration could be the most suitable distribution method, but the strategy should migrate multiple components simultaneously because the impact of moving one component at a time is too small.

The framework is not extendible with respect to the distribution methods because, contrary to what load information to use or what strategy to use, there is consensus in literature on a limited set of distribution methods. In addition, implementing distribution methods can be very intrusive to the middleware, application and the framework itself, making this less suitable for extension. Our mechanism includes the following distribution methods: initial placement, migration, per-session replication and session-migration replication.

In the following sections we will elaborate on the different parts of our framework-based mechanism: the load monitoring functionality, the load distribution strategies and the load distribution methods.

5.3.4 Load Monitoring Functionality

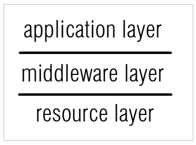


Figure 5-8 Monitoring at all three layers

In our mechanism, load monitoring can be done at all three different layers we distinguish in this thesis: application layer, middleware layer and resource layer (see *Figure 5-8*).

A *load meter* instruments the controlled application at one of the three different layers, and produce some specific type of load information. Our framework is extendible in that it is possible to add new load meters very easily. We define interfaces for this that do not expose internals of the load monitoring, including how the load information is transported from the load meter to the load distribution strategy.

We provide default load meters at the resource and middleware layer. Our *default resource-layer load meters* provide CPU and memory related information. Our *default middleware-layer load meters* provide load information on the number of active requests, response times and throughput. Since at the middleware layer it is possible to distinguish between requests for different components, it is possible to derive load information that is actually application specific, and at a fine granularity. Examples are response times for a certain component, throughput for a certain component, or derived values from this such as average response time. This makes it possible to use the load monitoring functionality to monitor the achieved QoS.

We do not provide default application-layer load meters since these are by definition application dependent. An example of application-layer load information would be the size of some application internal buffer. We provide interfaces for this that the application developer can use. What the load is, is opaque for our framework, the load is simply passed to the strategy without any form of interpretation.

Push, pull and periodic models

We support three models for the transportation of load information to the load distribution strategy:

- A *push model*, a state-change driven information policy (see Section 5.1.2) in which load meters actively report load information to the strategy, e.g., when some threshold is reached.
- A *pull model*, a demand-driven information policy in which the strategy requests data from the load meters.
- A *periodic model*, a periodic information policy in which load information is pulled at a configurable frequency.

The transportation of the load information causes overhead, i.e., it consumes network and processing resources. We try to minimize this overhead by using a hierarchical model in which the information is collected per node, and derived values such as averages can be calculated locally.

5.3.5 Load Distribution Strategies

Load distribution strategies decide how to distribute the load in order to fulfill the QoS requirements. Our mechanism does not limit the way in which the QoS requirements are expressed, since this can differ per load distribution strategy. Typically, QoS requirements are quantitative (for example response time and throughput), class-based (for example "best-effort", "silver" and "gold") or do not differentiate QoS (for example load balancing).

Strategies can use (i) the initial placement distribution method to control the node where components are created, (ii) the migration distribution method to migrate an existing component to another node, and (iii) per-session replication and (iv) session-migration replication to distribute the load of one component over several nodes.

Besides offering some default load distribution strategies, our framework-based mechanism allows easy addition of new strategies. Strategies can interact with the framework to:

– *Get access to load information*, using the before mentioned pull, push or periodic models.
– *Be notified of relevant load events*. The initial placement distribution methods will produce a load event for the creation of a component, and the per-session replication distribution methods will produce a load event for the start of a new session.
– *Convey distribution decisions*, which can be: the location for a new component, the migration of an existing component, the placement of replicas, which replica to use for a certain session and the migration of a session.

5.3.6 Supported Distribution Methods

We discuss the support for the different distribution methods one by one. We give special attention to the transparency and overhead issues, and indicate typical usage of the distribution method.

Initial Placement

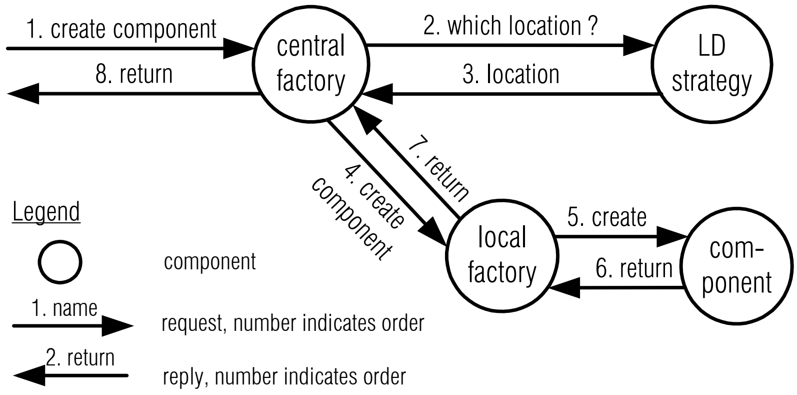
Initial placement makes it possible for a load distribution strategy to control the location where a component is instantiated. The component is created with its normal initial state, and is not moved or replicated. Initial placement can therefore be implemented without burdening the application developer with state access or state synchronization issues.

Since this distribution method operates only at instantiation time, there is no overhead during the rest of the lifecycle. Only at instantiation time there is some performance overhead due to forwarding the instantiation request to the proper location.

This distribution method can be used for all types of applications, has only a small overhead and is transparent for both client and server developer. The disadvantage is that it is not possible to change the distribution of the load for existing components. Without the creation of new components there is no possibility to control the load distribution. For example, a application with long-lived components is less suitable than a application with short-lived components.

We use the Factory pattern [Gamma94] to implement initial placement in a transparent manner. A client requests the creation of a component of a specific type to a logically centralized factory. This factory interacts with the load distribution strategy, which selects a target location for the creation of the component. A request for the creation of a component triggers a load event that is sent to the load distribution strategy, see *Figure 5-1*. The creation of the component is then delegated to a local factory at the selected location. *Figure 5-9* depicts the interactions between the involved components to implement initial placement using the factory patterns. The numbers indicate the order of the requests and replies.

Figure 5-9 Initial placement



Migration

Migration of components makes it possible to migrate a (non-replicated) component to another node. For migration we can re-use the migration operation of our dynamic reconfiguration mechanism (see Chapter 4) to ensure correctness (preserving structural integrity, mutual consistent states, and application invariants). Before actually migrating a component, we drive it to a safe state in which the component is not involved in any invocations. Ongoing invocations are completed. New incoming invocations are intercepted, and queued if they can be processed after the migration.

Migration is not as transparent as initial placement, since for typical components the application developer has to implement state access methods to allow the transfer of the state to the migrated component.

Migration can be used to migrate components in case there are insufficient resources available at the current location to provide the required QoS. A load distribution strategy might also do the opposite, and move components from a certain location to increase the amount of resources that is available for the remaining components. A third possibility to use migration is to migrate components that interact a lot to the same node, or to nodes that have a high bandwidth network connection between them (typically on the same LAN).

Figure 5-10 gives an example of a migration of a component S that has three clients, and is migrated from node X to node Y. After the migration the clients interact with S at its new location.

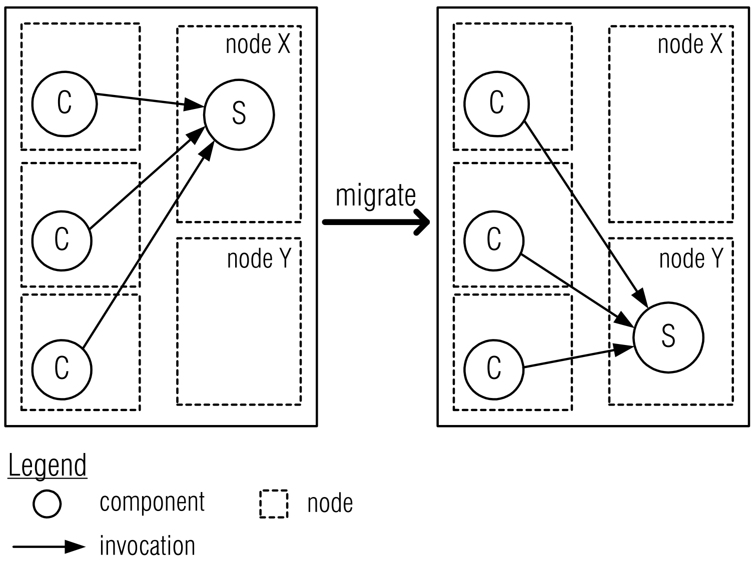


Figure 5-10 Example of migration

The overhead of the migration distribution methods is minimal during instantiation and during normal operation. Only when an actual migration is executed there is a temporary performance penalty because the components are frozen for a certain amount of time. The amount of time this takes depends on the longest running invocation in the set of re-located components.

Using migration is only beneficial from a performance perspective if the performance gains outweigh the temporary freeze of part of the application caused by the migration. Typically, migration is suitable for long-lived components that have short lasting invocations. In case of short-lived components the potential performance increase after migration might be too small to compensate the overhead caused by the migration. In case of long lasting invocations the migration can freeze the component and clients that use the component too long.

Replication

We offer replication for components that are stateless, or have only session state. For other types of components, the application developer has to implement his own mechanisms to guarantee (loose) consistency.

Per-invocation replication can be implemented using a central switch (see Section 5.1.5), which inspects every request and based on the current load condition forwards it to an appropriate replica. Other solutions would require coordination between the replicas, or with the client middleware, which would decrease transparency and cause extra overhead. A central switch however causes an extra delay for every invocation, and creates a potential bottleneck because all requests for a certain component have to go through this central switch. For this reason, we do not offer per-invocation granularity because we consider this performance overhead is too big (linear with the amount of requests), and because of the related scalability issues.

We do support per-session replication and session migration replication. For this we adopt a *client middleware visible routing* approach (see Section 5.1.3, or [Othman01C]) to route request to an appropriate replica. The client directly sends its request to the appropriate replica, and in case of session migration the client middleware changes the replica to which the requests are directed during the session. The overhead is limited to initially directing the client to the selected replica, and in case session-migration the re-direction to another replica. The alternative approach would multi-cast the requests, which would require support for this from the network and would require coordination between the replicas to communicate which replica would process the request, which causes extra overhead.

Figure 5-11 gives an example of a session migration for a replication based distribution method. It shows three clients C1 and C2 that use one replica (R1) on node X, and client C3 that uses another replica (R2) on node Y. By migrating the session that client C2 has with R1 to R2, part of the workload is shifted from node X to node Y.

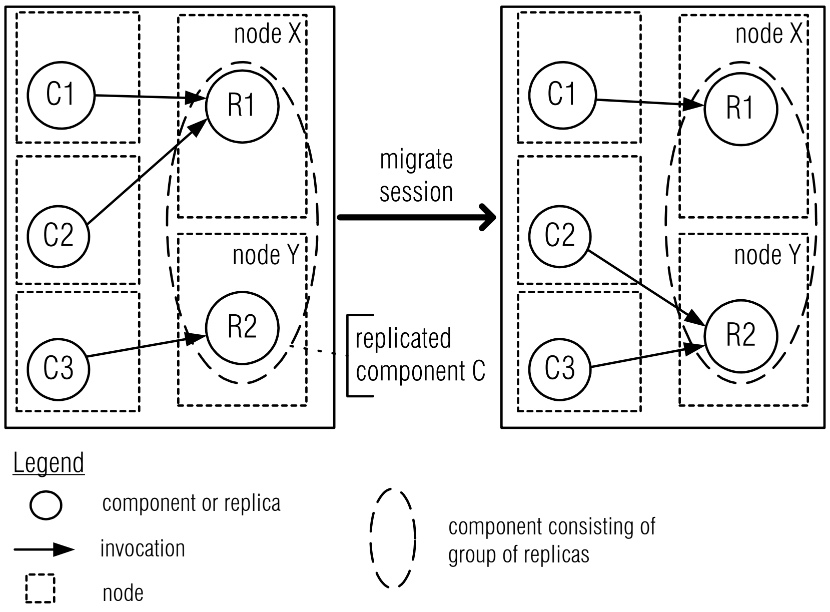


Figure 5-11 Example of a session migration.

In case of stateless components, no state transfer is needed between the involved replicas. However, in case of component with session state, the session migration includes the transfer of the relevant session state between the replicas. This is done with similar state access methods as in the case of component migration.

5.3.7 Quality of Service

Load distribution strategies distribute the load based on the QoS requirements. We do not limit the way these QoS requirements are expressed since this may differ per load distribution strategy. We however can categorize load distribution strategies based on the type of QoS requirements:

- *Quantitative performance requirements* – we consider two performance characteristics to quantify performance: throughput and response time (see Chapter 2). It is possible to have derived values of these two characteristics, e.g., 90% of the time the response time has to be less than 150ms.
- *Class-based performance requirements* – instead of quantitative requirements, a certain component or client-server connection is put in a class, e.g., gold, silver, bronze class. The different classes will have different amounts of resources available to components in that class. This division in classes in similar to Differentiated Services for IP networks [DiffServ98].

– *Load sharing or load balancing requirements* – no QoS differentiation is required, and there are no specific QoS requirements. The load distribution strategy should only maximize total throughput of the application (in case of load sharing) or balance the load (in case of load balancing).

The load distribution strategy bases its distribution decisions on the QoS requirements for the specific application. What the most effective way is to enforce these QoS requirements depends on the application, the environment and the nature of the QoS requirements. Besides accounting for the different factors that determine what the optimal way is to enforce the QoS, we also have to consider the stability, performance and scalability, i.e., the strategy should not use load information or distribution methods that incur too much overhead, and the strategy should scale up to high volumes of load events, load information, nodes, invocations and components.

To reduce overhead, increase scalability and account for the inherent unpredictability of large-scale applications, we propose a dynamic and heuristic strategy that is based on creating classes of nodes.

Class-based QoS Differentiation LD Strategy

We want to use load distribution to differentiate QoS, i.e., to allocate more resources to certain components in case this is needed to fulfill the QoS requirements. The assumption underlying the class-based QoS differentiation load distribution strategy is that we do not know the amount of resources a component will need, and that we also do not know the amount of resources that is available since resources are shared with other components, capsules etc. (see also our motivation for a dynamic approach in Chapter 3). We can however estimate this to some extent by considering current resource usage, which is provided to the strategy by the load information. The assumption we make is that the needed and available resources will not change very erratic, i.e., that past resource need and availability is a useful measure to predict future resource needs and availability.

We use this by creating classes of nodes. Within a class of nodes every node will have a load below a certain threshold, and the load is balanced between nodes. We can use any combination of load information to classify the nodes, e.g., we can classify using some CPU idle time, but also using a combination of CPU idle time, CPU queue length, memory usage and middleware-layer response time. Nodes can be added and removed from a class, when needed.

Figure 5-12 gives an example of four nodes that are divided over three classes, based on their CPU load. The lowest class “Bronze” does not have a maximum CPU load, and is a best effort class. The “Silver” and “Gold” have an increasingly lower maximum load, and thus there are increasingly more processing resources available for the components that run in these classes.

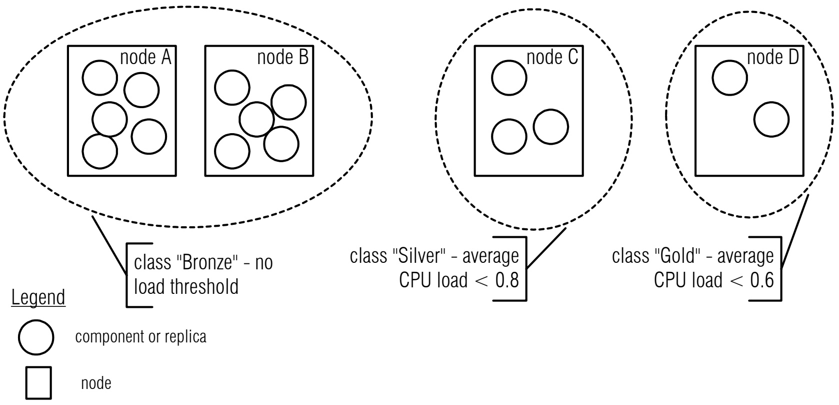

node A
node B
node C
node D
class "Bronze" - no load threshold
class "Silver" - average CPU load < 0.8
class "Gold" - average CPU load < 0.6
Legend
component or replica
node

Figure 5-12 Creation classes of nodes based on the available resources

Should some class reach a load that is too high, a node can be added to that class. The load can then be distributed over more nodes, resulting in a lower load per node. If the load is significantly lower than the threshold load for that class, one of the node in this class can be reclaimed, and possibly used for another class.

The granularity for QoS differentiation in case of initial placement and/or component migration distribution methods is per component, i.e., it is not possible to differentiate between clients or invocations for the same component. If a replication distribution method is used, the replicas can be distributed over the different classes, and thus it is possible to differentiate per session. Because of state synchronization issues between the replicas, this can have undesirable side effects that a replica in a higher class cannot meet certain QoS requirements because one of the replicas of a lower class slows down the synchronization process.

We now first describe the strategy in case of class-based performance requirements, and then for quantitative performance requirements.

Class-based QoS Requirements

In the case of class-based QoS requirements, we have both class-based QoS requirements and classes of nodes, and we let the classes of nodes coincide with the classes in the QoS requirements. The algorithm is then straightforward: create new components on a node of the appropriate class and assign new sessions to a replica in the appropriate class.

Figure 5-13 depicts an example in which client component C1 has “Bronze” class QoS requirements, and client component C2 has “Silver” class QoS requirements. Since there is no load threshold for the “Bronze” class, this is in fact best-effort.

The load is balanced within a class using the available distribution methods, and nodes are removed or added to classes if the needed. In case of overload situations, new load can be rejected (access control).

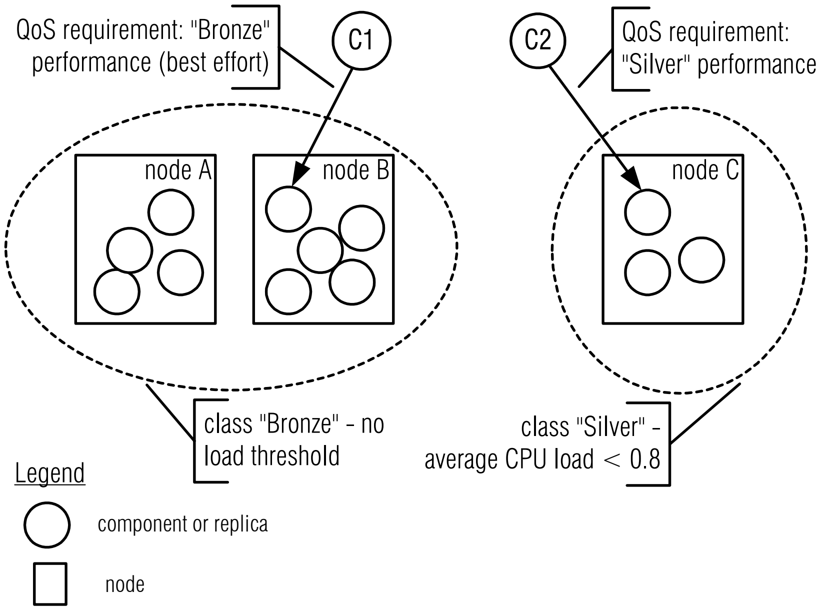

QoS requirement: "Bronze" performance (best effort)
C1
C2
QoS requirement: "Silver" performance
node A
node B
node C
class "Bronze" - no load threshold
class "Silver" - average CPU load < 0.8
Legend
component or replica
node

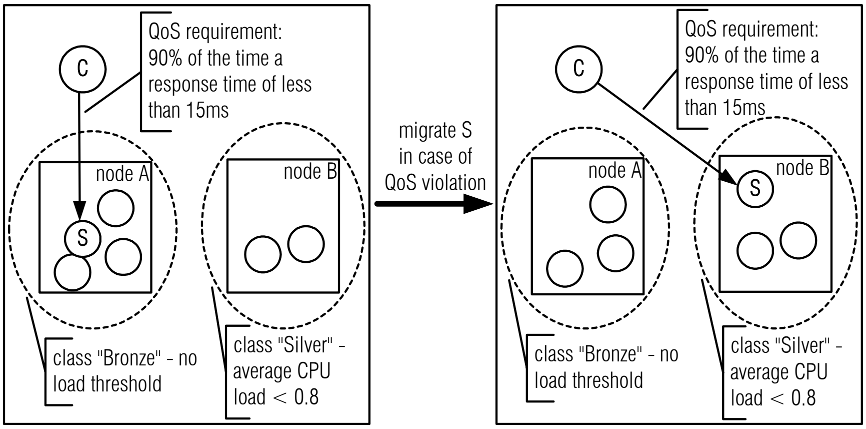
Figure 5-13 Example with class-based QoS requirements

Quantitative QoS Requirements

In case of quantitative QoS requirements we extend the above algorithm in two ways. The first extension is that we monitor the achieved QoS by using middleware-layer response time and the throughput load meters. This enables the strategy to determine if the required QoS is met. The granularity of this monitoring deserves extra attention, for example reporting the response time for every single invocation is not very scalable. A threshold-based load meter that pushes QoS violations to the strategy will reduce the overhead. The second extension is that if QoS requirements are not met, the strategy upgrades components or sessions to a higher class of nodes that has more resources available. The possibilities for this upgrade depend on the used distribution method:

– In case of the initial placement distribution method, this strategy is limited in flexibility. The only possible action for existing components is to move a whole node up to a higher class, and with some delay decrease the load on that node. For new components of the same type, which

can be expected to have similar resource needs, the strategy can choose to instantiate them on a node in a higher class.
- In case of the migration distribution method the strategy can migrate the involved components to a higher class. *Figure 5-14* gives an example in which server component S is migrated because the QoS requirements of client component C are not met.
- In case of replication, a session can be migrated to a replica that executes on a node of a higher class.


QoS requirement:
90% of the time a response time of less than 15ms
C
S
node A
node B
class "Bronze" - no load threshold
class "Silver" - average CPU load < 0.8
migrate S in case of QoS violation
Figure 5-14 Example of a migration to enforce quantitative QoS requirement

Overload

If a component or replica already runs in the highest class, and thus there does not exist a higher class to migrate a session or component to, a new class can be created that has a lower load threshold than the current highest class.

The above-described strategies assume that sufficient resources are available. In an overload situation where there are insufficient resources a combination of access control to refuse new workload and removing current workload can be used.

Other QoS Enforcing Strategies

Our framework-based mechanism is suitable to implement other load distribution strategies that enforce QoS requirements. Examples of such strategies are:
- Strategies that co-locate components to reduce communication overhead or to minimize network resource usage. Interaction patterns can be discovered using message reflection techniques, or be based on deployment descriptors, or a developer or application administrator can configure the strategy with knowledge on the interaction patterns.

– We can apply same class-based approach as above, but use capsule or even container as unit of granularity, e.g., migrate component to a capsule or container that has more execution threads available.

5.3.8 Comparison with Other Middleware Approaches

Main differences between our mechanism and other approaches are that, to our best knowledge, we are the only one to offer migration that preserves correctness, and we are the only one to use load distribution for QoS differentiation. None of the other approaches offers extendibility for custom load strategies, instead they only offer one or a few strategies without much justification why these strategies are sufficient. With respect to the load information, we offer several middleware-layer load meters, have a concept of application-layer load meters, and offer extendibility with respect to load meters.

We support several distribution methods, in particular stateless replication, session state replication, initial placement and component migration. Contrary to other approaches we do not neglect state synchronization and consistency issues, which is especially relevant for replication and migration-based approaches. We have a better transparency than most other approaches, most of which rely on a Naming or Trader Service.

5.4 High Level Design

Figure 5-15 gives a high level overview of all the different components of the Load Distribution Service (LDS). The shades indicate who supplies the components. For clarity we do not show interactions between components that are both part of the LDS, i.e., we only show interactions between the LDS and:

– the application, including the factory and components
– the load distribution strategies, since the LDS can be extended with extra strategies
– the load meters, since the LDS can be extended with extra load meters

The different parts of the LDS are:

– *Load Distribution Strategy* – either provided as a default strategy by the LDS, or 'plugged into' the LDS. It interacts with the Central Factory in case of initial placement method, with the Migration Manager in case of migration method and with the Replication Manager in case of replication method. It receives load information from the load monitoring components.

- *Strategy Manager* – handles configuration issues such as registering strategies, which strategy should be used for which components, configuration of strategies, configuration of load monitoring etc.
- *Central Factory* – used by the application when creating a new component that has to be 'load distributed', it sends creation and deletion events to the LD strategy to determine which local factory to use, and thus implements the initial placement distribution method.
- *Load Meter* – monitors some type of load information, default resource and middleware-layer Load Meters are supplied as part of the LDS, but application layer and additional resource and middleware-layer Load Meters can be added.
- *Other Load Monitoring Components* – monitoring functionality consists of the reporting of load data, the exchange of load data, and the collection of load data. This actually consists of several components. We will further explain this in Section 5.4.2.
- *Component Migration Manager* – implements in collaboration with other components such as the location agent and the LD agent the migration distribution method.
- *Replica Manager* – implements in collaboration with other components such as the Location Agent and the Load Distribution agent the replication of components, and the distribution of invocation over the replicas.
- *Location Agent* – used by client to get the component reference for a component that is subject to the replication or migration distribution method.
- *Component and Factory* – provided by the application developer.
- *Load Distribution Agent* – the instrumentation required for the migration and replication distribution methods.

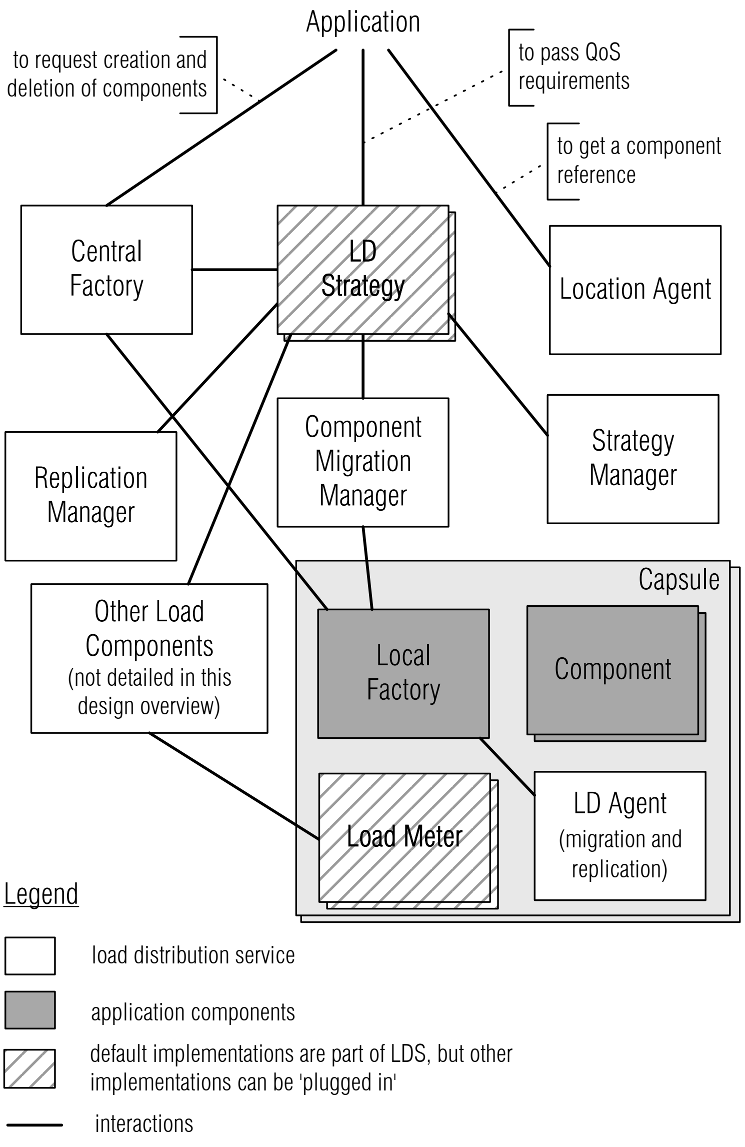

Application
to request creation and deletion of components
to pass QoS requirements
to get a component reference
Central Factory
LD Strategy
Location Agent
Replication Manager
Component Migration Manager
Strategy Manager
Other Load Components (not detailed in this design overview)
Capsule
Local Factory
Component
Load Meter
LD Agent (migration and replication)
Legend
load distribution service
application components
default implementations are part of LDS, but other implementations can be 'plugged in'
interactions

Figure 5-15 High Level Design of the Load Distribution Framework

5.4.1 Load Distribution Strategies

The LDS provides some default load distribution strategies, but an essential property of our LDS is the possibility to extend the LDS with new strategies. New strategies can be 'plugged in' easily, without requiring access to the source code of the LDS or requiring recompilation of the LDS. The LDS implements the distribution methods and load monitoring functionality, which can be used by the strategy through offered interfaces. The responsibility of the designer of a new strategy is limited to making distribution decisions. A strategy can be application or environment dependent, i.e., use specific characteristics of an application of the environment the application runs in to be most effective.

As explained in Section 5.1.1, the designer of the load distribution strategy does have to make sure the strategy is stable, i.e., he should avoid processor thrashing and thundering herd effects.

Strategies have to be registered with the Strategy Manager, who instantiates the strategies and also determines which strategy received which load events. For example, in case of the application wants to instantiate a new component only one strategy can process the create load event. Several strategies can be active at the same time, and it is the responsibility of the Strategy Manager that the load event is directed to the correct strategy.

5.4.2 Load Monitoring

Figure 5-16 depicts the basic architecture of the monitoring part. The arrows indicate how the relationship should be read. This architecture is based on [Rackl01]. The structure contains four logical components that have to be mapped to physical implementation components. The four different components are:

– *Load Meter* – performs load metering for a certain load type.
– *Meter Agent* – collects load data from one or more Load Meters, and may also manipulate that data (for example to provide averages).
– *Load Notifier* – propagates load data to the Load Collectors that have registered for such notifications.
– *Load Collector* – collects the load data.

We discuss each of these load monitoring components in some more detail below.

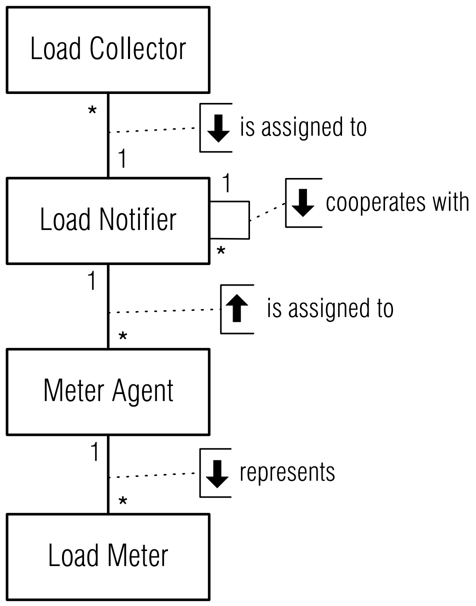


Figure 5-16 Load monitoring components

Load Meters are suppliers of load information. This information can relate to load on the resource layer, on the middleware layer or on the application layer. Load meters can support either of two data exchange models: push or pull. Load meters that support the pull model provide an operation to retrieve the load data measured by the meter. Periodic load information can be gathered by periodically pulling the load information for the Load Meters.

The primary purpose of a *Meter Agent* is to act as a façade for multiple Load Meters that may exist in a location. The Meter Agent can do some additional manipulation of the data received from Load Meters, and should be local to the Load Meters to avoid communication overhead.

The *Load Notifier* component receives load data from Load Meters, filters these reports and propagates them to Load Collectors that have registered as consumers. The Load Notifier is a logically centralized component, but may be physically distributed.

The *Load Collector* is the component that receives the load data sent by Meter Agents. The strategy component uses the Load Collector for retrieving load information.

5.5 Conclusions

We gave an overview of the area of load distribution, and how load distributions concepts and terminology can be applied to middleware-based applications. We discussed and compared related work in the area of middleware-based load distribution. We proposed a new mechanism for load distribution of component-middleware-based applications that integrates parts of existing approaches, and extends them with QoS differentiation, a higher degree of transparency and more consideration for consistency. Central to our mechanism is the extendibility, i.e., we make it possible to easily add new strategies and to monitor other types of load information. We presented a high level design of a Load Distribution Service that implements our mechanism.

Our load distribution mechanism:
– supports initial placement of a component;
– supports migration of a component, while maintaining correctness;
– supports replication of stateless components;
– supports replication of components with session-state;
– supports migration of a session to another replica;
– supports resource, middleware and application-layer load monitoring;
– has default load meters for common load information such as CPU load, throughput and response time;
– has a hierarchical monitoring design that minimizes overhead and increases scalability;
– enables QoS differentiation;
– has some default load distribution strategies such as least-loaded distribution based on CPU load;
– can easily be extended with new load distribution strategies;
– can easily be extended with new load information.

Main Contributions
The main contributions of this chapter are:
– we show that a load distribution QoS mechanism can be implemented in middleware;
– we discuss the trade-offs with respect to transparency for the different load distribution methods;
– our mechanism integrates different distribution methods, and allows for extendibility with respect to the load distribution strategies and load information;
– we show it is possible to use message reflection, i.e., portable interceptors, to make our mechanism more transparent than other existing approaches;
– we achieve separation of concerns of the designer of strategy and load information, and the concern of the application component designer (no mixed code);
– we propose how to use load distribution for QoS differentiation.

Other Contributions

Other contributions of this chapter are:

- we propose a new generic model for load distribution;
- we propose a new categorization of distribution methods;
- we indicate important limitations in current middleware-based approaches with respect to applicability and the consistency;
- we discuss the trade off between being optimal and being generic, and argue that effective load distribution solutions will have to incorporate application dependent knowledge.

Our Load Distribution Mechanism

Major issues to solve for any load distribution approach are (i) how to direct invocations to the appropriate component or replica (ii) how to handle the inherent trade-off between offering an optimal solution and offering a generic solution (iii) how to keep application consistency in case of migration or replication distribution methods.

Our mechanism solves the first issue, i.e., the direction of invocations to the appropriate component or replica in a transparent manner using the middleware.

We addressed the second issue by having a framework-based mechanism that allows easy extension of the Load Distribution Service with additional load distribution strategies and load information. Application or even environment specific solution can then easily be implemented, while re-using the distribution method, load monitoring and other functionality of the Load Distribution Service.

The third issue cannot be completely solved in a generic, transparent and efficient manner. Maintaining consistency is dependent on the distribution method. For initial placement this is straightforward. Maintaining consistency in case of migration is more difficult, but is solvable in a generic and transparent manner, although it depends on the application whether the overhead is acceptable. For replication however this is not solvable, especially if we consider that this has to be done with minimal overhead and has to be transparent to the component developer. Our mechanism therefore does not provide replication in the true sense, i.e., our mechanism does not keep all the states of all the replicas consistent. We however do provide stateless replication, and session based replication. The latter means that state changes are local to a specific replica, which from a consistency perspective is similar to stateless replication. It could be possible to implement some state consistency protocols, similar to how CASCADE does this for caching, or to how Globe does this. This would however violate transparency, and we chose not do pursue this.

Comparison with Other Approaches

A main difference between our mechanism and other approaches is that our mechanism offers a wider range of distribution methods, in particular our mechanism offers migration of components, (while preserving correctness) and session-migration replication for components with session state. None of the other approaches offers extendibility for custom load strategies, and only offer one or a few strategies without justifying why these strategies are sufficient. With respect to load information, we offer several middleware-layer load meters, have a concept of application-layer load meters, and offer extendibility with respect to load meters.

Future Work

Strategy per application category

We have argued in this chapter that the optimal load distribution strategy, and the distribution methods and optimal load information required for this, is application dependent. Without contradicting this, we think there might very well be a certain limited set of application categories for which a certain load distribution strategy works best. Future research could be on determining these categories, and the corresponding strategies.

Loose replication

While full replication is not desirable for load distribution purposes, we could research further what the possibilities and trade-offs are for loose replication. Current work (CASCADE, Globe) is applied to 'close to client replicas/caches', but the proposed solutions might be suitable for the more general case also. It would violate transparency, but it could be beneficial to get insight in the trade-off between transparency and performance, and to have support for this in our Load Distribution Service.

OMG standard

The OMG issued a request for proposal for a Load Balancing and Monitoring specification [CORBALB]. The submissions to this request for proposal indicate that the adopted solution will have many similarities with [Otham01C]. It is based on using to the standard CORBA request forwarding mechanism to direct client to the appropriate object. It has fixed load metrics and fixed (but configurable) strategies, but mentions possible subsequent additions to the specification to allow replacements of these. The submissions are based on replication, i.e., they assume that the load is balanced over a group of objects that share one identity. The submissions however do not propose any facilities for state synchronization. Our load distribution mechanism is a superset of the submissions, and should OMG actually adopt a Load Balancing and Monitoring specification, we could align our mechanism with it.

Concluding Remarks

As became clear in this chapter, initial placement, replication for stateless components and per-session replication for components with session-state are the most transparent distribution methods. Avoiding migration and especially state synchronization between replicas benefits both the transparency, and minimizes the overhead. One can do this by partitioning the application such that a component is dedicated to a single client, thereby avoiding concentrating too much load on one component. [Neuman94] identified three ways to build a scalable application, replication, caching and partitioning. Basically, what we are advocating here is that replication, or caching, is only part of the solution, and application design should still take the partitioning principle into account. This can be considered a violation of transparency in that it restricts application design, but it will increase transparency with respect to state access and synchronization, and allows optimal use of load distribution.

Chapter 6

Proof of Concept

This chapter describes a prototype of the QoS mechanisms that are proposed in this thesis: the Dynamic Reconfiguration Service and the Load Distribution Service. Our prototype integrates both QoS mechanisms, where the Load Distribution Service makes use of the Dynamic Reconfiguration Service. We do however describe and evaluate the QoS mechanisms separately.

For the implementation of our prototype we had to choose a specific technology, namely CORBA. Similar prototypes could however be implemented in other component middleware technologies.

This chapter is structured as follows: Section 6.1 describes the Dynamic Reconfiguration Service prototype; Section 6.2 describes the Load Distribution Service prototype.

6.1 Dynamic Reconfiguration Service

This section[4] describes the Dynamic Reconfiguration Service (DRS) prototype. The dynamic reconfiguration mechanism and high level design of the DRS were already described in Chapter 4.

There are three different types of users of the DRS: the change designers, the reconfigurable component designers and the designers of clients of a reconfigurable component. We first describe the view that each of these types of users have on the DRS. We then describe the design choices we made for the DRS components themselves. We end this section with an evaluation of the prototype, which includes a discussion of the performance, the transparency and future work.

[4] Parts of this section have been published in the papers [Almeida01B] and [Wegdam03A], which are co-authored by the author of this PhD thesis, and in a master thesis that was supervised by the author of this PhD thesis [Almeida01C].

The prototype is developed in Java, and with the ORBacus 4.0.4 CORBA ORB [ORBacus]. For an introduction to CORBA, see Chapter 2. Some parts of this chapter however require more advanced CORBA knowledge to understand them.

6.1.1 Change Designer View

The change designer interacts with the Dynamic Reconfiguration Service through the `ReconfigurationManager` interface. The `ReconfigurationManager` interface provides operations for creating and removing objects, managing factories and specifying reconfiguration steps.

Normal Creation and Removal

Creation and removal of objects is part of the normal lifecycle of any object. In itself these operations are not specific to the DRS. It is mandatory however that the application logic creates and deletes objects through the `ReconfigurationManager`, because the DRS has to assign a unique identifier to each reconfigurable object.

Operations for object creation and removal are inherited from the `GenericFactory` interface defined in the Fault Tolerant CORBA specification [CORBA].

The `create_object()` operation allows the application to request the creation of an object by specifying the identifier of the object's type and the criteria to be used in the creation. The `delete_object()` removes an object.

The IDL fragment is shown below. We simplified the IDL by leaving the exceptions and type definitions out, see [Almeida01C] for the complete IDL.

```
interface GenericFactory {
   Object create_object(
      in TypeId type_id,
      in Criteria the_criteria,
      out FactoryCreationId factory_creation_id);

   void delete_object(
      in FactoryCreationId factory_creation_id);
};
```

The `type_id` is the same identifier as used in the interface repository to denote the most derived type of an interface. The type identifier is used in conjunction with the criteria to determine the local factory that creates the application object.

The `the_criteria` parameter allows application to define initialization parameters, and restrictions on how to create the object. Examples of criteria are initialization values, the required version of an object and the preferred location of an object.

The `factory_creation_id` parameter allows the entity that invokes the factory and the factory itself to identify the object for subsequent manipulation. The `factory_creation_id` is an `Any` value that contains a `ReconfigurableObjectId`. This `ReconfigurableObjectId` is used to denote a reconfigurable object.

The object reference returned by the `create_object()` operation is a reference to the reconfigurable object, which is valid during the complete reconfigurable object lifetime. This object reference continues to be valid after subsequent replacements and migrations.

Reconfiguration Step

A system evolves incrementally from its current configuration to a new configuration in a reconfiguration step, which is perceived as an atomic action from the perspective of the application.

A reconfiguration step is modelled by a `ReconfigurationStep` object, which can be created through the `create_reconfiguration_step()` operation of the `ReconfigurationManager` interface.

Composing a reconfiguration step

The `ReconfigurationStep` interface provides means to compose a reconfiguration step from reconfiguration operations, namely,

- object creation;
- object removal;
- object replacement;
- object migration.

The change designer composes a reconfiguration step and commits it, i.e., requests its execution. The actual reconfiguration does not start till after the commit.

Creation and removal

The operations for object creation and removal have the same syntax as defined in the `GenericFactory` interface. They differ from the ones defined in the `GenericFactory` interface in that they are executed when the reconfiguration step is committed. The object reference returned by `create_object()` should only be used after the reconfiguration step has been executed. An additional operation `remove_type()` can remove all objects of a certain type.

Replacement

Object replacement can be done both on an individual basis, i.e., by specifying `factory_creation_ids`, or on a type basis, i.e., by specifying the type of the objects. While replacement on an individual basis provides a fine-grained control over the version of each object in the system, its use should be avoided when all objects of a type can be replaced simultaneously. Reconfiguration on a type basis simplifies version management, by preventing objects of the same type from having different versions.

The operation `replace_object()` requires the user to specify the object being replaced and the criteria to be used in the creation of the new version of the object. The criteria are used to determine which factory to use.

```
interface ReconfigurationStep {
   void replace_object(
      in FactoryCreationId factory_creation_id,
      in Criteria the_criteria);
```

The operation `replace_type()` requires the user to specify the type being replaced, the new type and the criteria to be used in the creation of the new version of objects. The new type must be identical or derived from the original type. If the Reconfiguration Manager receives requests for the creation of objects of a type that is being replaced, then those request are deferred until the end of the reconfiguration. After reconfiguration, the identifier of the original type can still be used when requesting object creation, so that type replacements with sub-typing can be transparent for the client application. Nevertheless, the new derived type is used for the actual creation. The `replace_type()` operation returns the list of objects replaced.

```
FactoryCreationIds replace_type(
        in TypeId current_type_id,
        in TypeId new_type_id,
        in Criteria the_criteria);
```

Migration

Object migration can be done both on an individual basis, i.e., by specifying `factory_creation_ids`, or on a type-location basis, i.e., by specifying the type of the objects to be migrated and their current location. Migration on an individual basis provides a fine-grained control over the location of each object in the system. The local factory that is used to create the relocated version of an object is determined by the criteria.

```
void migrate_object(
        in FactoryCreationId factory_creation_id,
        in Criteria the_criteria);

FactoryCreationIds migrate_objects(
        in TypeId type_id,
        in Location origin,
        in Criteria the_criteria);
```

Default criteria

It is possible to set default criteria for the creation of a type by invoking `set_default_criteria()`. It influences the behaviour of object creations after reconfiguration, e.g., by specifying the default location of new objects of the type.

```
void set_default_criteria(
        in TypeId type_id,
        in Criteria the_criteria);
```

The optional state translator for the reconfiguration step can be provided by invoking `set_state_translator()`. We describe the state translation below.

```
void set_state_translator(
        in GenericStateTranslator translator);
```

Requesting the Execution of a Reconfiguration Step

A reconfiguration step can be executed in blocking mode by invoking `commit()`, in which case the operation returns when the reconfiguration is complete. A reconfiguration step can also be executed in non-blocking mode by invoking `deferred_commit()`, in which case the operation returns immediately. In the non-blocking mode, `is_completed()` should be invoked to determine whether the reconfiguration step has already been executed, or whether errors have occurred.

The non-blocking mode is necessary for self-replacement, i.e., when the object that initiates the replacement is expected to be replaced. In this case, the blocking mode would lead to deadlock, since the object being replaced would have a pending request (`commit()`) and would never reach the idle state.

```
void commit();

void deferred_commit();

boolean is_completed();
```

State Translation

In the replacement operations, the change designer can optionally specify a state translator. The state translator is used by the DRS when the system has reached the safe state. In the safe state, all the states of the affected objects are consistent and stable. These states are used as input to the state translator, which translates them to the state of the objects being introduced to replace the affected objects.

A state translator has to be implemented by the change designer. A state translator implements the `GenericStateTranslator` interface. This interface defines the structure `Instance`, which comprises the type identifier, the reconfigurable object identifier, the state of a reconfigurable object instance and the reconfiguration operation being applied to it. The operation `types_supported()` returns the types supported by the state translator. In the absence of a supplied state mapping for a particular type, the identity function is used, i.e., the state is not modified. The operation `translate()` translates the states of a set of instances into derived states.

```
enum ReconfigurationOperationType {
       CREATION,
       REPLACEMENT,
       MIGRATION,
       DELETION};

struct Instance {
       TypeId type_id;
       ReconfigurableObjectId id;
       State the_state;
       ReconfigurationOperationType op_type;};

TypeIds types_supported();

void translate(
       in Instances original,
       out Instances derived);
```

The state of an object may include object references that are narrowed by a state translator. If a reference to be narrowed points to an object being replaced, e.g. as part of a `replace_type()` with sub-typing, an unchecked narrow must be performed. A checked narrow would invoke an operation on the object to check if the type is correct, which we want to avoid since this object is part of the reconfiguration. Also a check narrow would delay the state translation.

Unchecked narrows have been incorporated in the CORBA standards with the introduction of CORBA Messaging. For ORB implementations that do not support unchecked narrows, an object reference should be externalized as `CORBA::Object`. These references should only be narrowed when first used by the new version of an object.

6.1.2 Component Designer View

We discuss here how a reconfigurable component designer interacts with the DRS, and the design restrictions that the DRS poses on the reconfigurable component designer. This section is structured in four parts:

- *state access* – discusses the methods a component designer has to implement to provide access to the internal state.
- *factory* – the usage of the DRS makes the usage of a factory pattern mandatory. We discuss the exact interfaces that have to be used.
- *active object* – for active reconfigurable objects there are some additional methods that the component designer has to implement for the DRS to be able to reach a reconfiguration safe state.

- *threading* – in some non-typical cases the component designer has to pass some implicit context information to outgoing invocations to enable the DRS to determine the invocation path.

State Access

Reconfigurable Objects must implement the `ReconfigurableObject` interface, providing the state-access operations `get_state()` and `set_state()`, which are identical to the state-access operations in the Fault Tolerant CORBA specification. The state is encoded as a sequence of octets. The encoding of the state may be application-specific. Nevertheless, the application developer is strongly recommended to specify the state as a structure in IDL. This guarantees interoperability and allows re-use of available CORBA functionality to encode data structures as sequences of octets (Common Data Representation [CORBA]).

```
interface ReconfigurableObject {

   State get_state() raises(NoStateAvailable);

   void set_state(in State s) raises(InvalidState);
};
```

Factory

Reconfigurable Object Factories implement the `ReconfigurableObjectFactory` interface, which inherits the `GenericFactory` interface. These factories must provide `create_object()`, `delete_object()` and `get_reconfiguration_agent()` operations. The `get_reconfiguration_agent()` operation returns the `ReconfigurationAgent` associated to a given reconfigurable object.

```
interface ReconfigurableObjectFactory :
 GenericFactory {
    ReconfigurationAgent get_reconfiguration_agent(
        in ReconfigurableObjectId id);
};
```

A Reconfigurable Object Factory creates and deletes instances of objects on behalf of the Reconfiguration Manager, and registers and de-registers these instances with the Reconfiguration Agent.

Figure 6-1 depicts the participation of an object factory in the creation, replacement and migration of an object.

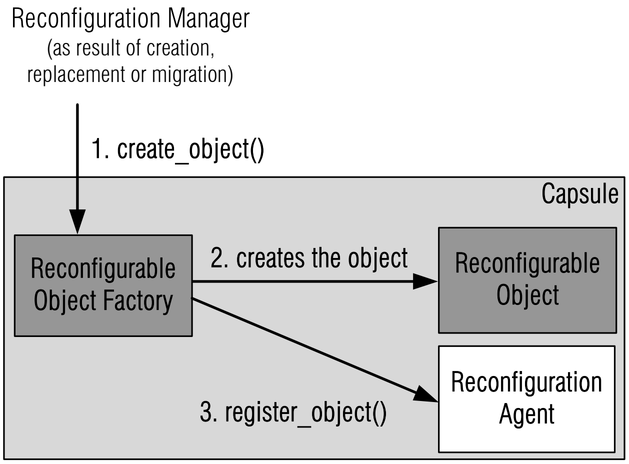


Figure 6-1 Participation of factory in creation, replacement and migration

The `create_object()` operation is invoked by the Reconfiguration Manager (1) to create an instance of an object. `create_object()` may be invoked in the course of object creation, replacement or migration.

In the case of replacement or migration, the Reconfiguration Manager delegates the creation, by repeating the parameters supplied by the user and adding extra properties (name-value pairs) in the criteria parameter. These properties are the location-independent object reference to be used by the instance of the object and its reconfigurable object identifier. This allows the object to maintain its identity across subsequent reconfigurations, and publish the location-independent object reference as its object reference. We explain how we implemented the location-independent IOR in Section 6.1.4 (page 174).

A reconfigurable object may retrieve its location-independent object reference and its reconfigurable object identifier from the Reconfiguration Agent, by invoking the `get_reference()` and `get_reconfigurable_object_id()` operations. A reference to the Reconfiguration Agent can be obtained by invoking `ORB::resolve_initial_references("ReconfigurationAgent")`. If the reconfigurable object invokes POA methods to retrieve its object reference, the POA supplies the conventional location-dependent object reference.

In case of an actual reconfigurable object creation, the Reconfiguration Manager includes in the criteria the IOR and the Id properties, and an extra `ApplicationObjectCreation` property. This allows the factory to distinguish, if necessary, between an actual object creation and a creation that results from replacement.

`create_object()` creates the instance of the object (2), registers it with the Reconfiguration Agent (3) and returns the location-dependent object reference to the Reconfiguration Manager.

```
interface ReconfigurationAgent{
   void register_object(
         in ReconfigurableObjectId id,
         in Object rec_obj_reference,
         in octets adapter_id,
         in octets object_id);

   void deregister_object(
         in ReconfigurableObjectId id);

   Object get_reference(
         in octets adapter_id,
         in octets object_id);

   ReconfigurableObjectId
     get_reconfigurable_object_id(
         in octets adapter_id,
         in octets object_id);

   boolean is_affected(
         in ReconfigurableObjectId id);
};
```

`register_object()` receives as parameters the identifier of the reconfigurable object and the location-independent object reference as sent by the reconfiguration manager, the identifier of the object adapter in which the object is located, and the object identifier used by this object adapter.

The `delete_object()` operation is invoked by the Reconfiguration Manager to delete an instance of an object. The execution of `delete_object()` may be invoked in the course of reconfigurable object removal, replacement or migration. If a factory finds it necessary to distinguish between object removal on the one hand, and replacement and migration on the other hand, it may invoke the operation `is_affected()` of the `ReconfigurationAgent` with the identifier of the object as a parameter. `is_affected()` returns true if the object is currently being replaced or migrated. The use of `is_affected()` allows us to maintain the syntax for `delete_object()` as defined in the Fault Tolerant CORBA specification.

Figure 6-2 depicts the participation of an object factory in the removal, replacement and migration of an object.

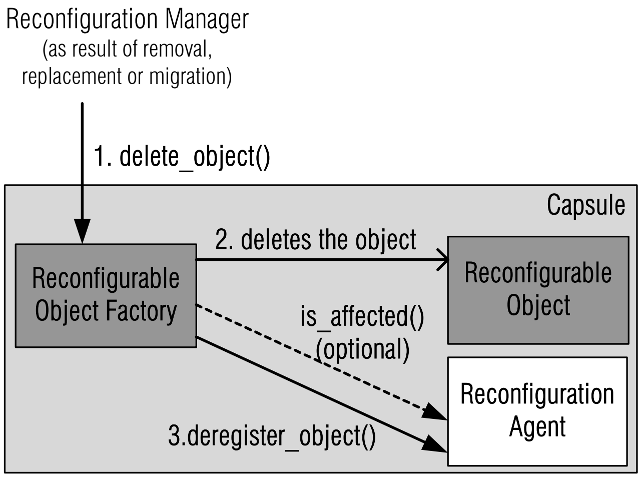


Figure 6-2 Participation of factory in removal, replacement and migration

Active Object

Active reconfigurable objects must also implement the `ActiveObject` interface, in order to provide `passivate()` and `activate()` operations in addition to state-access operations. An active object is an object that can initiate non-nested requests, i.e., requests that are not causally related to an incoming request. An active object should react to the `passivate()` operation by refraining from initiating non-nested requests, i.e., by exhibiting reactive behaviour. The `activate()` operation is the inverse of `passivate()`, i.e., it informs an object that it is allowed to exhibit active behaviour.

```
interface ActiveObject {

   void passivate();

   void activate();
};
```

Threading

For reconfigurable objects, the DRS maintains some context information for the thread in which a request is being processed. This context information, called the DRS context, contains the invocation path of the request being treated and the identifier of the object treating the request. The DRS context is used in order to determine the invocation path that is sent implicitly with a request. As discussed in Chapter 4, we need the invocation path when we drive the system to a reconfiguration safe state.

Figure 6-3 shows the DRS context of a thread treating a request req_1. The DRS context contains the invocation path of req_1 ($\{O_1, \ldots O_N\}$) and the identifier of the object O_{N+1} treating $\texttt{req}_1$. The request $\texttt{req}_2$, which is a nested request of $\texttt{req}_1$, contains the invocation path of $\texttt{req}_1$ appended with the identifier O_{N+1} ($\{O_1, \dots O_N, O_{N+1}\}$).

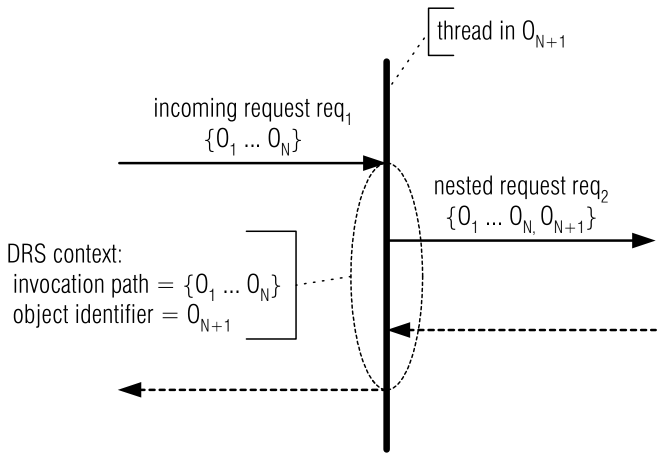

thread in 0_{N+1}
incoming request req_1
$\{0_1 \dots 0_N\}$
nested request req_2
$\{0_1 \dots 0_{N,} 0_{N+1}\}$
DRS context:
invocation path = $\{0_1 \dots 0_N\}$
object identifier = 0_{N+1}

Figure 6-3 The DRS context and the propagation of the invocation path

The DRS context is accessible through the `ReconfigurationCurrent` local object. An instance of the `ReconfigurationCurrent` object can be obtained by invoking
`ORB::resolve_initial_references("ReconfigurationCurrent")`.

An active reconfigurable object must register each thread that issues non-nested requests with the DRS. This is done by using the operation `register_thread()` of the `ReconfigurationCurrent` object. The parameters for `register_thread()` are the identifier of the object adapter in which the object is located, and the object identifier used by this object adapter. For the Portable Object Adapter (POA) these parameters are obtained through `POA::id` and `POA::reference_to_id()` respectively.

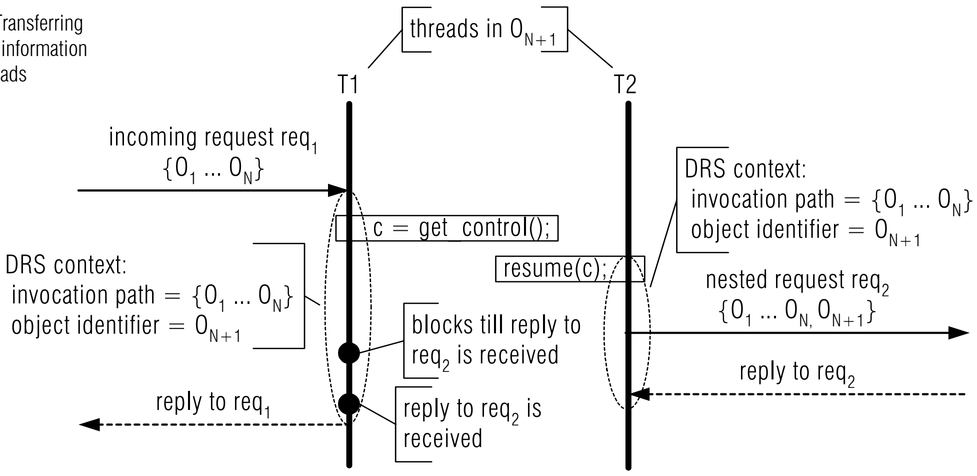
```
module ReconfigurationService {
    interface Current : CORBA::Current {
         void register_thread(
           in octets adapter_id,
           in octets object_id);
...
```

In the typical case an incoming request is treated in only one thread, as depicted in *Figure 6-3*, and the propagation of the invocation path is completely transparent for the reconfigurable object developer. The propagation of the invocation path can be done without involvement of the component developer because of the usage of message reflection. CORBA Portable Interceptors (see Chapter 3) inspect and alter the implicit parameters of an invocation, and copy them to and from the `ReconfigurationCurrent`. It also does not require any changes to the CORBA ORB. We thus fulfil the time, expertise and common middleware requirements, as identified in Chapter 3 (generic requirements for QoS Mechanisms).

In less conventional threading strategies, however, more support from the reconfigurable object developer is required, as explained in the remainder of this section.

In case the completion of an incoming request req_1 served in a thread `T1` depends on the completion of a nested request issued in another thread `T2`, the DRS context information in thread `T1` must be transferred to thread `T2`. This situation is depicted in *Figure 6-4*, where `T1` blocks waiting for nested request `req`$_2$ in `T2` to be processed.

Figure 6-4 Transferring DRS context information between threads


threads in 0_{N+1}
T1
T2
incoming request req_1
$\{0_1 ... 0_N\}$
c = get_control();
DRS context:
invocation path = $\{0_1 ... 0_N\}$
object identifier = 0_{N+1}
resume(c);
DRS context:
invocation path = $\{0_1 ... 0_N\}$
object identifier = 0_{N+1}
nested request req_2
$\{0_1 ... 0_{N,} 0_{N+1}\}$
blocks till reply to req_2 is received
reply to req_2
reply to req_1
reply to req_2 is received

In this case, the `get_control()` operation of the `ReconfigurationCurrent` must be invoked in thread `T1` to obtain the `Control` structure that must be passed to the `resume()` operation of the `ReconfigurationCurrent` in thread `T2`. The names `get_control()`, `resume()` and `Control` are adopted in order to resemble the Indirect Context Management with Explicit Propagation in the CORBA Transaction Service [CORBATS].

```
interface Current : CORBA::Current {
   void register_thread(
      in octets adapter_id,
      in octets object_id);

   struct CurrentSlotInfo {
```

```
            ReconfigurableObjectId id;
            ReconfigurableObjectIds invocation_path;};

        typedef CurrentSlotInfo Control;

        Control get_control();

        void resume(in Control which);
    };
```

The invocation path must also be propagated through non-reconfigurable objects that are in the invocation path between reconfigurable objects. If these non-reconfigurable objects are implemented with the unconventional threading strategies identified previously in this section, the object developer is responsible for transferring DRS context information between threads in the same way as required for reconfigurable objects with unconventional threading strategies.

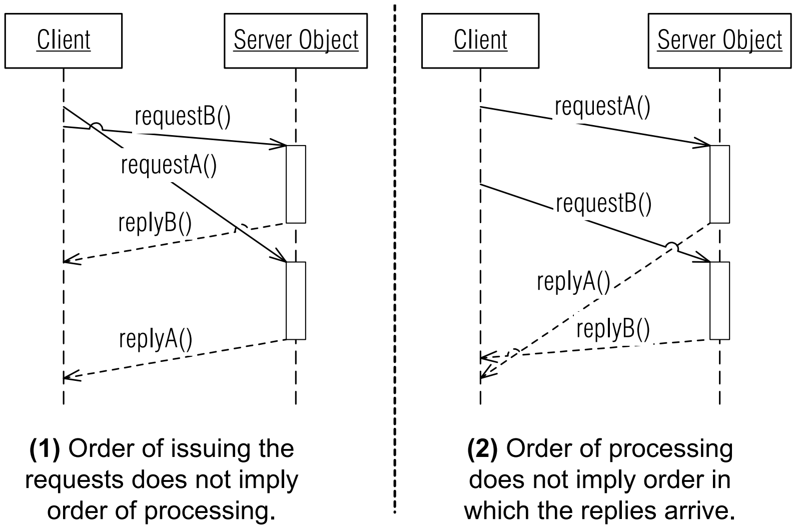
6.1.3 Client Designer View

The Dynamic Reconfiguration Service is transparent for client applications, which manipulate object references and issue requests to reconfigurable objects in the way prescribed in the CORBA object model. During reconfiguration, requests may be queued by the ORB and re-directed to the target object, transparently for the client application.

One may think that the selective queuing of requests interferes with ordering guarantees provided by the middleware infrastructure. Nevertheless, in the CORBA object model, the order in which a client issues requests does not imply the order in which a target object processes the requests. This can, for example, depend on the server-side queue in the ORB, which is not part of the CORBA specification.

This can be seen in example (1) of *Figure 6-5*. In addition, the order in which replies reach a client does not imply the order in which the server processed the requests. This can be seen in example (2) of *Figure 6-5*.

Figure 6-5 CORBA ordering guarantees


Client
Server Object
requestB()
requestA()
replyB()
replyA()
(1) Order of issuing the requests does not imply order of processing.
Client
Server Object
requestA()
requestB()
replyA()
replyB()
(2) Order of processing does not imply order in which the replies arrive.

Nevertheless, CORBA does guarantee that (i) the issuing of a request is eventually followed by the processing of the request, and that (ii) the processing of a request is eventually followed by the arrival of the reply at the client-side. A client can assume that requests are processed sequentially if it issues a request after the arrival of the reply of a previous request. Our DRS does not jeopardize these guarantees.

6.1.4 Design

The two main design choices for the DRS are concerned with the location independent object reference, and the implementation of the selective queuing. We describe both choices, and discuss alternative solutions and motivate the selected one. We also describe how a simple reconfiguration step is implemented, and how a composite reconfiguration step is implemented.

Location Independent Object References

There are three alternatives to implement location independent object references: forwarding proxies, client ORB notifications and location agents. We discuss each of them, and motivate why we selected the location agent alternative.

Forwarding proxies

A location independent object reference can be implemented by keeping *forwarding proxies* in the location of the old targets. These forwarding proxies forward requests to the new location (or version) of an object.

The problem with this solution is that the forwarding chain grows each time an object is replaced or migrated. Therefore, when compared to the

Location Agent solution, this approach introduces more overhead, is harder to manage and uses more system resources than necessary. Furthermore, locations where forwarding proxies exist cannot be actually taken off-line, e.g., in the case of a hardware upgrade.

Sending a LOCATION_FORWARD reply to the client ORB when the client makes the first request after reconfiguration can solve the first disadvantage. The LOCATION_FORWARD message is a standard CORBA message that notifies the client ORB redirect the client to use an updated IOR. The client ORB then resends the request to the new IOR, and from that point onwards keeps using the updated IOR. This however does not solve the second disadvantage: the forwarding proxy has to remain active until there is certainty that no clients might still have the old IOR. In addition, this takes still would require a way to know which clients might still have the old IOR. Keeping track of all clients would prevent the scalable implementation of the DRS. Object reference distribution in CORBA, or in any of the common middleware technologies for that matter, is not controlled by the ORB, i.e., object references can be exchanged between objects by many different means.

Notify client side ORB

A second alternative is to *notify the client-side ORB* of the reconfiguration, substituting the current object reference with the new modified object reference. Although compared to the first alternative this alternative would be faster, this alternative would also have to keep track of all possible clients of a reconfigurable object.

Location agent

A third alternative to keep an object reference valid after reconfiguration is to use of a *location agent* [Henning98]. In case a request on a modified object reference is performed, an exception at the client ORB makes it contact the Location Agent, which uses the above described LOCATION_FORWARD mechanism to inform a client ORB of the new location of the target object. We select this alternative because it does not require the DRS to keep track of all clients, and it makes it possible to take the node on which the old target ran offline. We describe the details on how we implemented this below.

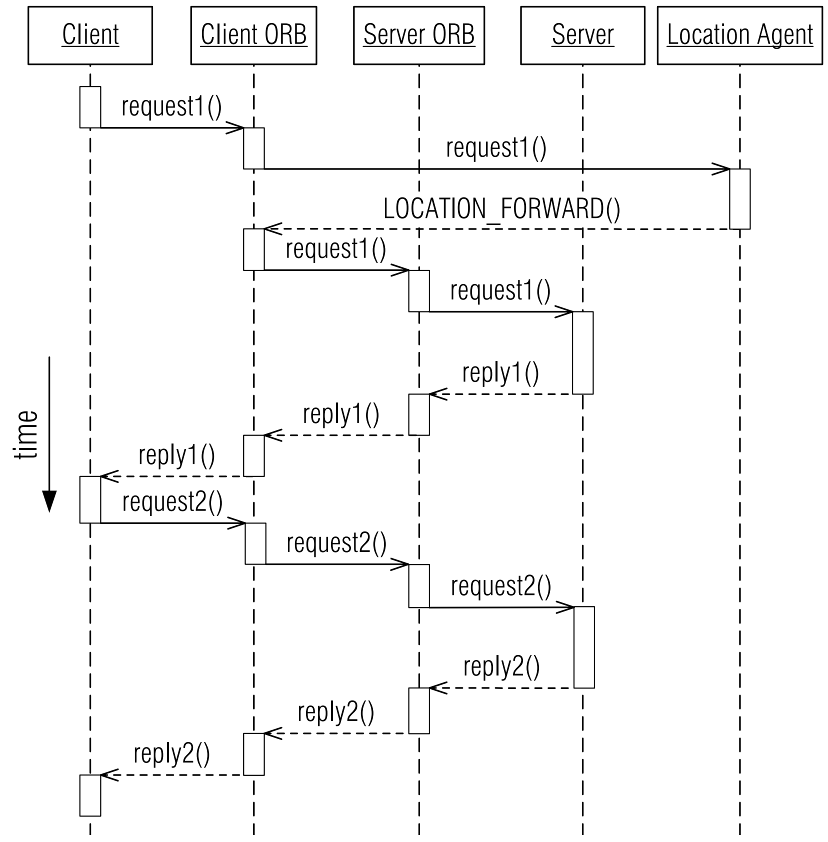
Location agent implementation

The location agent must generate object references that point to itself instead of pointing to the actual location of object. These object references are called location-independent object references. The location agent does this by creating an object reference that contains the location's agent address and the reconfigurable object identifier as the object-key.

When a request is issued by a client for the first time, the location agent is invoked. The location agent is implemented with a servant locator, which keeps a registry mapping a reconfigurable object identifier to the conventional location-dependent object reference. The servant location throws a `LocationForward` exception with the current location-dependent object reference that points to the current version of the object. This exception reaches the client ORB as a `LOCATION_FORWARD` GIOP (General Inter-ORB protocol) message. As prescribed in the rules of GIOP, the client ORB reissues the request with the new object reference, until an error occurs when using this reference.

Figure 6-6 depicts the basic functioning of the mechanism in the establishment of the binding.

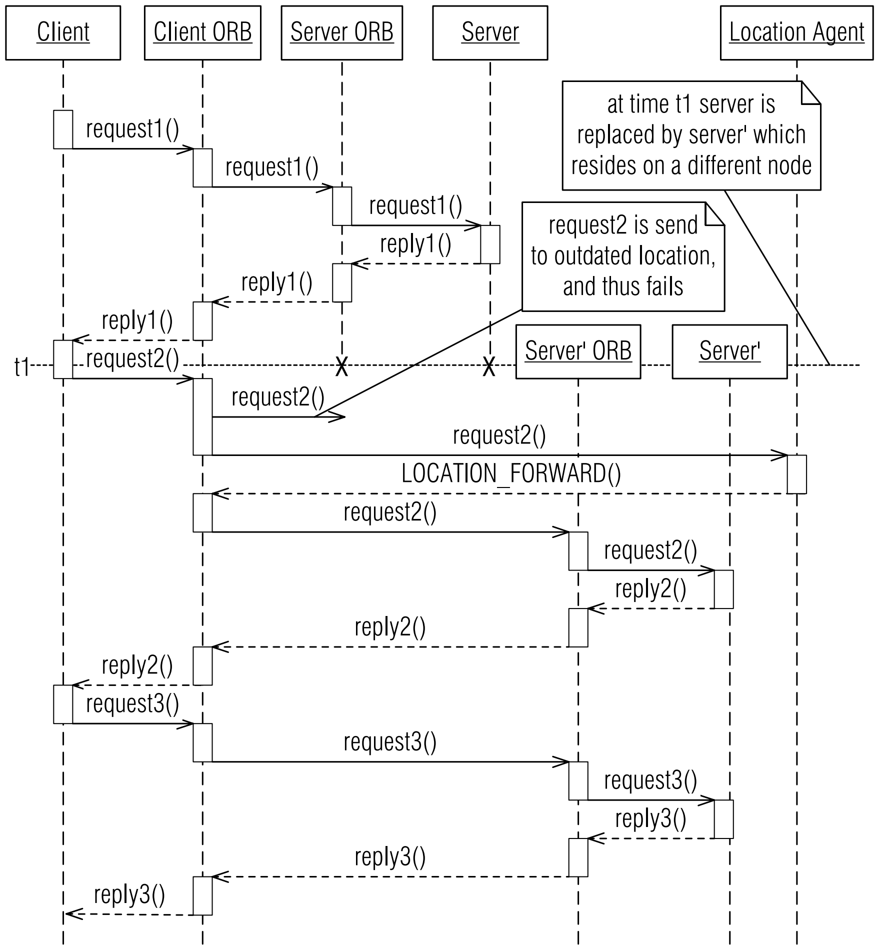
Figure 6-6 Transparent binding establishment



When reconfiguration occurs, the reference being used by a client ORB is no longer valid. At this point in time, GIOP mandates that the client ORB switches back to the original object reference, which in this case is the location-independent reference. The re-establishment of the binding follows the same procedure as in the first establishment, transparently for the client application.

Figure 6-7 depicts the basic functioning of the mechanism in the re-establishment of a binding broken by reconfiguration.

Figure 6-7 Transparent binding re-establishment



This mechanism is fully transparent to the client application and the overhead for this solution is limited to the first invocation of a client on the reconfigured target object. The forward mechanism is implemented in the implementation repositories of some CORBA ORB implementations, although the interface between the implementation repository and the server ORB has not been standardized.

In our implementation, the Location Agent implements the `LocationAgentAdmin` interface, which allows the Reconfiguration Manager to register an object while retrieving its location-independent object reference (`register_object()`), get the location-independent object reference to an object (`get_reference()`), get the location-dependent object reference to an object (`get_target_object()`) and remove the current reconfigurable object identifier and location-dependent object reference association (`deregister_object()`).

```
interface LocationAgentAdmin {
   Object register_object (
     in Object target,
     in ReconfigurationService::
             ReconfigurableObjectId id);

   Object get_reference (
     in ReconfigurationService::
             ReconfigurableObjectId id);

   Object get_target_object (
     in ReconfigurationService::
             ReconfigurableObjectId id);

   void deregister_object (
     in ReconfigurationService::
             ReconfigurableObjectId id);
};
```

Selective Request Queuing

In order to bring the system to the reconfiguration-safe state we have to implement a selective queuing of requests. Requests that do not belong to the 'laissez-passer' set should be queued transparently for clients and target objects.

Selector

We identify two functions that are needed to realize selective queuing: a selector and a queue. For each request directed to an affected object, the *selector* determines if the request belongs to the 'laissez-passer' set. If it does, the request is forwarded to the target object as in normal operation. Otherwise, the request is sent to the queue.

Queue

The queue is responsible for storing requests until reconfiguration is complete. Stored requests are redirected to the new version of the target object after the reconfiguration.

We want the implementation of selector and queue to be transparent to both client and server object. This means we have to implement them as part of the middleware layer, opposed to implementing them as CORBA objects on top of the middleware.

We have different alternatives on how to implement the selector and queue in the middleware layer. We distinguish them by considering the allocation of the selector and the queue to different parts of the middleware infrastructure, namely the client-side ORB (client ORB) and the server-side ORB (server ORB). The benefits and drawbacks of each alternative are the following:

Selector and queue at client side

– *Pure client-side solution* – Selector and queue are implemented as extensions of the client ORB. Requests are selected and blocked at the client side, imposing no overhead to the communication infrastructure. Nevertheless, there is a serious scalability problem since all potential clients of an affected object must be known a priori, and all these clients must be notified of the set of affected objects. This drawback applies to all solutions that place the selector in the client ORB. Moreover, this solution complicates management, since the client ORB extensions have to be deployed in every potential client of the reconfigurable objects;

Selector and queue at server side

– *Pure server-side solution* – Selector and queue are implemented as extensions of the server ORB. This solution offers better scalability than the pure client-side solution, as clients do not need to be known a priori and do not need to be informed of the reconfiguration. Since the client ORB does not have to be extended, management and deployment can be simplified;

Hybrid solution for selector and queue

– *Hybrid solution* – The selector and the queue are implemented as extensions of the server ORB and the client ORB, respectively. Clients that attempt to issue a request to an affected object are informed to block the request and re-issue it when they get a notification that the reconfiguration has been completed. In effect, the queue becomes distributed among all clients that attempt to issue a request to an affected object during reconfiguration. This solution requires more communication overhead than in the case of the pure server-side solution.

The solutions discussed above imply that the ORB has to be extended somehow, i.e., the ORB has to be instrumented. This can be realized by either making proprietary modifications to the ORB code, but this violates the common middleware requirement (see Chapter 3). We therefore will use message reflection to implement the instrumentation. Since we use CORBA, we can use Portable Interceptors for this (see also the discussion on message reflection in Chapter 3)

The pure client-side solution can be directly implemented using portable interceptors in the client ORB, but has the above mentioned scalability issue.

The implementation of the pure server-side solution with portable interceptors has a problem because the server side `receive_request()` interceptors executes in the same thread as the target invocation [CORBA]. We would have to block this thread to implement the queuing function. The effect of the blocking depends on the threading model of the server object. Suppose this is a fixed pool of threads or a single thread, this can cause deadlock. And if it would be a thread-per-request threading model, we have a scalability issue. Because of these problems we have to reject the server-side solution.

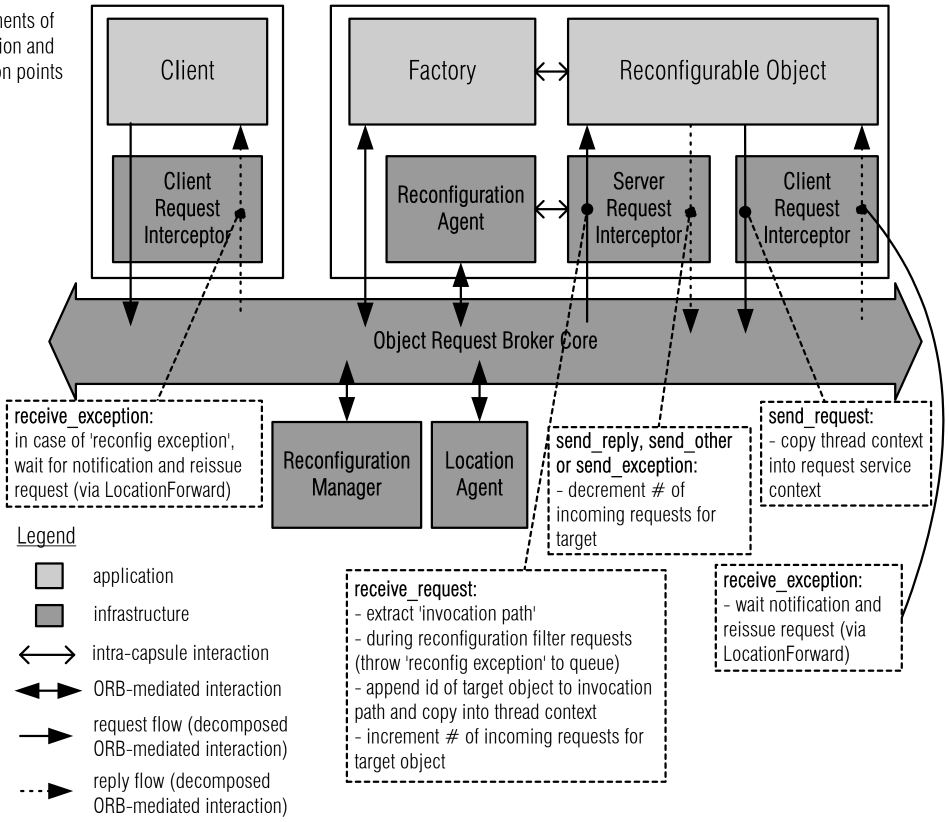
The hybrid solution can be directly implemented using portable interceptors in the client ORB and in the server ORB, without risk of deadlocks or scalability issues. We therefore select the hybrid solution.

Implementation of hybrid solution

The selector can use the service context of a request to determine a request belongs to the 'laissez-passer' set. Service contexts allow implicit arguments to be passed in a method invocation. When a reconfigurable object issues a request, it adds its identifier to the service context. During the first stage of the reconfiguration process, when a request arrives at the selector the request's service context is inspected. If the identifier of an affected object is included in the service context, the request belongs to the 'laissez-passer' set and should not be queued.

Figure 6-8 gives an overview of the implementation with a brief description of the actions undertaken at the client- and server-side request interception points.

Figure 6-8 Elements of the implementation and request reification points



Client
Client Request Interceptor
Factory
Reconfigurable Object
Reconfiguration Agent
Server Request Interceptor
Client Request Interceptor
Object Request Broker Core
Reconfiguration Manager
Location Agent
receive_exception: in case of 'reconfig exception', wait for notification and reissue request (via LocationForward)
send_reply, send_other or send_exception: - decrement # of incoming requests for target
send_request: - copy thread context into request service context
receive_request: - extract 'invocation path' - during reconfiguration filter requests (throw 'reconfig exception' to queue) - append id of target object to invocation path and copy into thread context - increment # of incoming requests for target object
receive_exception: - wait notification and reissue request (via LocationForward)
Legend
application
infrastructure
intra-capsule interaction
ORB-mediated interaction
request flow (decomposed ORB-mediated interaction)
reply flow (decomposed ORB-mediated interaction)

Before a reconfigurable object receives a request, the request is reified in the `receive_request` interception point, and the service context propagated with the request is extracted. A service context is an implicit parameter used by CORBA services to propagate information along with a request. For the DRS, it contains the list of reconfigurable objects that depend on the execution of the request to become idle. The list of reconfigurable objects is appended with the identification of the request's target object and the appended list is copied into the `ReconfigurationCurrent` local object. The `ReconfigurationCurrent` object provides access to an implicit per-thread context, and in this way the thread is associated with the reconfigurable object.

During the first stage of the reconfiguration process, server request interceptors inspect the propagated service context. If any of the affected objects is listed in the service context, the request should be allowed to complete, so that all affected objects can progress to the idle state. If no affected objects are listed, an exception is raised. This exception is intercepted in the client-side request interceptors, which block the thread of execution and reissue the blocked requests later by raising a `LocationForward` exception.

There is one disadvantage to the hybrid queuing solution, which we cannot avoid. Suppose an affected object needs to issue a request to an unaffected object to be able to finish some ongoing invocation. In the normal case the client will process this invocation, and return the reply to the affected object that can then reach the quiescent state. The issue here is that the client ORB might not have the resources to process this request, because they resources are blocked by our client-side queue. For example, it might have a fixed thread pool of 4 threads. If the client already has four blocked requests that it tried to send to affected objects, there are no threads available to process the incoming request from the affected object. Although we think this is very unlikely to actually happen, this is an inherent limitation of the way we implemented the queuing. A solution would be to adapt the algorithm to extend the set of affected object with this specific object, and unblock the blocked requests. This is however not trivial since we have to be able to detect this situation, and because it requires extra communication between the Reconfiguration Manager and the affected objects. Another solution that would at least prevent the reconfiguration as a whole to wait forever is to detect that there has been a deadlock (e.g. with help of a timer), and abort the reconfiguration. We however did not implement this.

Because of this disadvantage, and also to prevent the installation of client-side interceptors, a solution that does not involve the client side could be considered as an alternative implementation. This could be done by providing a separate queue. The implementation of such a queue, however, cannot be done in a portable manner since it would either involve ORB changes or using ORB proprietary functionality.

Re-issuing requests after reconfiguration

We consider two different policies for determining the moment at which queued requests are re-issued: the wait-and-retry policy and the wait-for-notification policy. The policy is determined by the Reconfiguration Manager prior to reconfiguration.

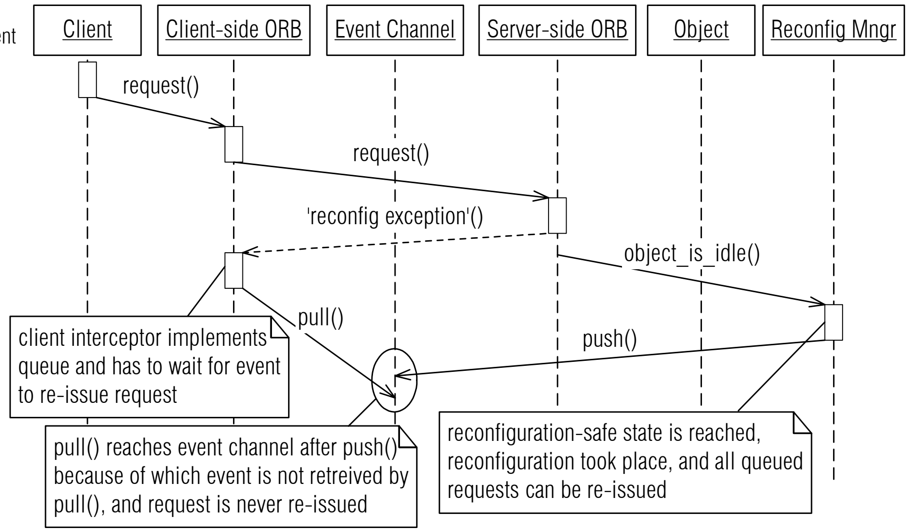
Wait-and-retry policy

In the *wait-and-retry policy*, the Reconfiguration Manager sends a time interval to the Reconfiguration Agents of the affected objects. This time interval is passed in the reply service context of the exception that is sent to clients during reconfiguration. The client-side request interceptor waits for the time interval specified and reissues the request. If the reconfiguration is not yet completed, the server-side request interceptors will raise the exception again, and the client-side request interceptor will block for the time internal again.

Wait-for-notification policy

In the *wait-for-notification policy*, the requests are re-issued when the Reconfiguration Manager gives a signal to do so. The most straightforward implementation of this would be to use the CORBA Event Service, and have the Reconfiguration sent an event to all the blocked clients that the reconfiguration is over. There is however a problem with this, in some cases a client might remain blocked. An example of this is shown in *Figure 6-9*. In this execution, the client's `pull()` request reaches the event channel after the event that indicates the end of reconfiguration. According to the specification of the event service, the event is not delivered, and the client is left waiting for a response to `pull()` indefinitely.

Figure 6-9 Problem using the CORBA Event Service for notifying clients


Client
Client-side ORB
Event Channel
Server-side ORB
Object
Reconfig Mngr
request()
request()
'reconfig exception'()
object_is_idle()
pull()
push()
client interceptor implements queue and has to wait for event to re-issue request
pull() reaches event channel after push() because of which event is not retreived by pull(), and request is never re-issued
reconfiguration-safe state is reached, reconfiguration took place, and all queued requests can be re-issued

Although the above-described problem with the CORBA Event Service, excludes its usage, we could implement a similar event service that would pass the event to the client even if the push already occurred. We however decided it was easier to block the client by introducing a (logically) centralized point which all clients use to block.

We implemented this by introducing the `ReconfigurationManagerCallback` object that the Reconfiguration Manager creates before the start of a reconfiguration. The Reconfiguration Manager sends the reference of the callback object to the Reconfiguration Agents of the affected objects. This reference is passed in the reply service context of the exception that is sent to clients during reconfiguration. The client-side request interceptor invokes the `block_until_ready()` method of the `ReconfigurationManagerCallback` object, which blocks until the end of the reconfiguration. Serializing the access to the object, e.g., by using a single threaded POA, can prevent possible scalability issues with this callback object. The client application is not at any moment aware of the reconfiguration, potentially observing an increase in the response time of invocations that are queued waiting for reconfiguration.

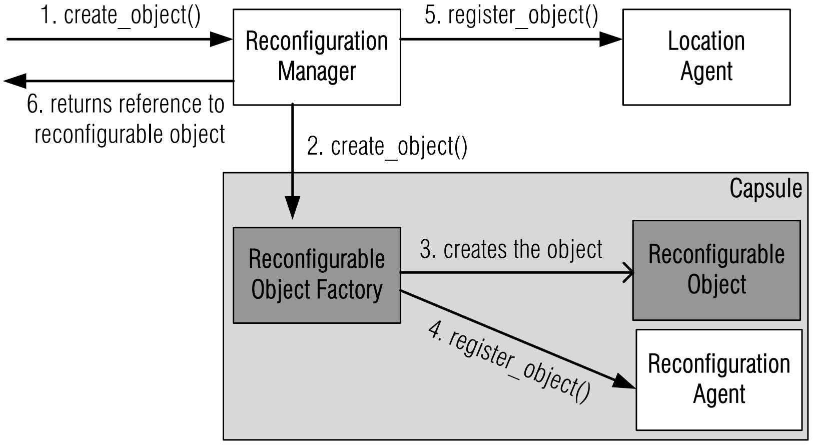
Performing a Reconfiguration Step

The DRS components, namely the Reconfiguration Manager, the Location Agent and the Reconfiguration Agents, cooperate to perform a reconfiguration step. Below we detail the activities executed to perform a simple reconfiguration step.

Object Creation

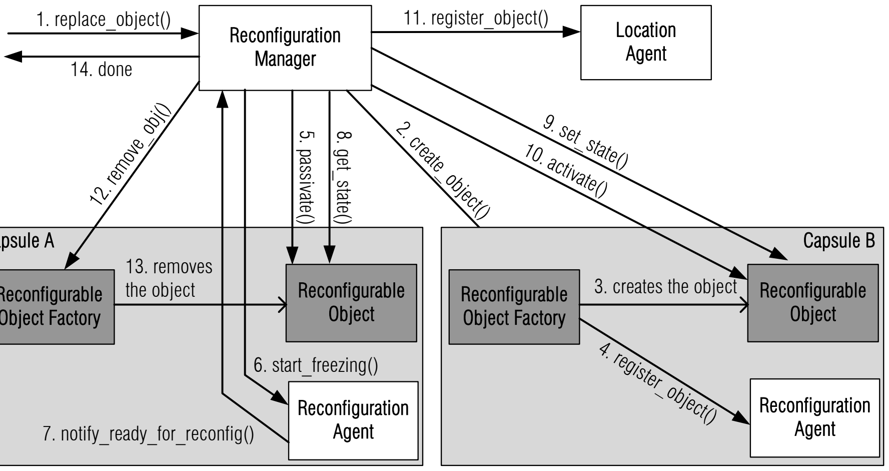
Figure 6-10 shows the creation of an object. The Reconfiguration Manager delegates the creation to a local Reconfigurable Object Factory (2), which creates the object (3) and registers it with the Reconfiguration Agent responsible for the capsule where the object lives (4). After that, the Reconfiguration Manager registers the recently created object with the Location Agent (5), and returns the object reference to the client that requested the object creation (6).

Figure 6-10 Object creation



Object Replacement
Figure 6-11 shows the replacement of an active object. Initially, the Reconfiguration Manager delegates the creation of the new version of the object to a local Reconfigurable Object Factory (2). After that, the Reconfiguration Manager notifies the affected reconfigurable object and its Reconfiguration Agent of the start of the reconfiguration (5, 6). The Reconfiguration Agent restricts the behavior of the affected object, and notifies the Reconfiguration Manager when the object is ready for reconfiguration (7). The state-transfer is conducted (8, 9), the object is allowed to exhibit active behavior (10), the new location of the object is registered with the Location Agent (11), and the local factory is requested to remove the previous version of the object (12). In *Figure 6-11* we do not show state translation that may take place.

Figure 6-11 Object replacement



Object Migration

Object migration is treated as an object replacement where the factory of the new version of the object is located in the destination capsule.

Object Removal

The Reconfiguration Manager delegates object removal to the Reconfigurable Object Factory responsible for the object being removed, and de-registers the object with the Location Agent.

Performing a Composite Reconfiguration Steps

The procedures for the execution of simple reconfiguration steps described above are special cases of the procedure to execute a composite reconfiguration step. To minimize the time in which the affected objects are blocked, we do all the creation's before driving to a safe state, and do all the removal's after the unblocked all the affected objects. This results in this following procedure:

1. for each object creation, migration and replacement, the Reconfiguration Manager invokes the `create_object()` operation of the appropriate local object factory (determined by type and criteria);
2. the Reconfiguration Manager invokes the `passivate()` operation of all active objects in the affected set;
3. the Reconfiguration Manager invokes the `start_freezing()` operation of all active objects in the affected set. The parameters of `start_freezing()` include the set of affected objects and the information for the queuing policy adopted;
4. all the affected objects eventually reach the idle state and the Reconfiguration Agents invoke the `notify_ready_for_reconfig`() to let the Reconfiguration Manager know this. The safe-state is reached;
5. the Reconfiguration Manager reads the states of the affected objects, translates them with the `translate()` operation of the state translator, when this is supplied, and sets the states of the new or relocated versions;
6. the Reconfiguration Manager (re-) registers the location-dependent object reference with the location agent;
7. the Reconfiguration Manager invokes the `activate()` operation of the new or relocated versions;
8. the client-side ORBs are notified that the reconfiguration is over, in case reissuing policy is wait-for-notification; and
9. for each object removal, migration and replacement, the Reconfiguration Manager invokes the `delete_object()` operation of the local object factory that holds the version to be discarded.

6.1.5 Evaluation

The DRS prototype has been successfully tested for applications with multiple multithreaded objects, including nested and re-entrant invocations. See [Almeida01C] for a description of the tests. The prototype validates our dynamic reconfiguration QoS mechanism for component-middleware-based applications. It is a prototype nevertheless, and should be tested further to assure practical usability and correctness.

We evaluate the prototype by discussing the performance and transparency, and we mention some future work on the DRS.

Performance

We separate two aspects of the performance of the DRS: (i) the overhead during normal operation and (2) the impact on execution during reconfiguration.

Overhead during normal operation

The usage of the portable interceptors causes some extra overhead for every invocation during normal operation. Our tests show that this causes an increase in response time of about 0,13ms [Almeida01B]. This increase is more or less independent of the type of invocation. In the worst case this causes a relative increase in response time of about 12.4%. These measurements however are quite dependent on the used ORB (ORBacus), implementation language (Java) and the environment in general.

Impact on execution during reconfiguration

Clients of an affected object observe an increase in the response time of operations that are invoked during a replacement. This increase only applies to invocations that reach the target object after the beginning of the reconfiguration and before the end of the reconfiguration.

The increase in response time during reconfiguration is highly dependent on the application. It is upper-bounded by the duration of the longest pending invocation in the set of affected objects at the moment the reconfiguration starts plus a fixed delay introduced by the reconfiguration service for co-ordination overhead. For active objects, the amount of time taken for the object to exhibit a reactive behavior should also be considered in the calculation of the upper bound of the increase in response time.

According to an experiment conducted with the replacement of one single object, this delay is approximately 530 ms of which 320 ms are related to marshalling and de-marshalling of service contexts. We see opportunities for optimizations that should reduce these values.

The experiments should be repeated for different ORB implementations to reach more conclusive results. In addition, further tests should consider the effects of reconfiguration on the performance of the new object right after the reconfiguration, as all the queued requests are directed to it.

Transparency

We discussed the transparency aspects of our mechanism already in Chapter 4. Here we will summarize how this is effectuated in the prototype.

For the typical case of a reactive object that does not spawn additional threads, there are two obligations for the developer of a reconfigurable object: to use the factory pattern, and to provide state-access methods. These are the same requirements as for the CORBA Fault-Tolerance Service. Also portable interceptors have to be installed for reconfigurable objects and clients of reconfigurable object, and more general for any object that could be in the invocation path for re-entrant invocations. The installation of the portable interceptors however does not require changes to the application code, it is only a deployment issue.

For active objects there is an additional obligation to implement methods for passivating and re-activating the object. For active objects, and for objects that spawn additional threads to process an incoming request, the component developer has the additional responsibility to update the invocation path in the ReconfigurationCurrent. This is in our opinion not time consuming, but does violate especially expertise requirement (see Chapter 3) since the developer has to be knowledgeable when to do this.

The obligations for passivation and invocation path updates depends on the specific middleware technology. For example, in EJB active components or components that spawn new threads are not allowed, and this would thus not be an issue.

Future Work

We could extend the prototype with a mechanism to abort on ongoing reconfiguration if we detect a problem, for example some run-time exception. An abort would mean re-activating all passivated affected objects, accepting invocations again and unblocking all clients.

We can also use this abortion mechanism if the time to reconfigure exceeds some value specified by the reconfiguration designer. This can serve two purposes. The first is if the QoS requirements for the application put an upper bound on the time a reconfiguration can last, in which case aborting the reconfiguration and resuming operation with the old configuration might be preferred to violating the QoS requirements. The second is that the reconfiguration might have reached a deadlock, e.g. because one of the affected components does not comply with its restrictions. An abort of the complete reconfiguration is then the only solution.

Our prototype of the dynamic reconfiguration service could be made available to a large number of developers and it could be applied in complex realistic applications. This would further validate the service and provide feedback that would lead to possible improvements.

We expect our dynamic reconfiguration mechanism to be applicable directly on a component-based middleware infrastructure such as Enterprise JavaBeans [EJB] and CORBA Component Model [CCM]. The support to dynamic reconfiguration in this case may be located in the container of a reconfigurable component. A component could be declared to be reconfigurable in its deployment descriptor, thus providing a strict separation between application and reconfiguration concerns. With component-based middleware, it would be easier for the component developer to define the state access functions, because the relationships between components are not encapsulated in the implementation of a component, and these relationships can be reified and manipulated at run-time by a third-party, which, in our case, is the dynamic reconfiguration service. Since a component is a deployment unit, it would also be possible to re-use or adapt the deployment facilities of a middleware infrastructure in order to include dynamic reconfiguration. This includes re-use of the factory patterns that newer generation component middleware mandates.

6.2 Load Distribution Service

This section[5] describes the prototype that validates our load distribution QoS mechanism. The validation focuses on the load distribution methods, the load monitoring capabilities and the pluggeability of the load distribution strategies and load meters. The provided default strategies are focus at testing the functional and performance overhead aspects of the Load Distribution Service (LDS). We spent only limited effort in developing advanced strategies. As a consequence, the strategies we developed are not suitable for a wide range of applications and environments, and cannot enforce complicated QoS requirements.

This section is structured as follows: Section 6.2.1 describes the design of the distribution methods; Section 6.2.2 describes the design of the load monitoring functionality; Section 6.2.3 describes the load distribution strategies, and the Strategy Manager; Section 6.2.4 gives the view on the LDS from the perspective of the different users: the component designer,

[5] Parts of this section also appeared in a master thesis [Post02] that was supervised by the author of this PhD thesis

the client designer, the strategy designer, the load meter designer and the system administrator; Section 6.2.5 describes the tests we did, and the resulting measurements on the overhead and performance of the LDS prototype; Section 6.2.6 evaluates the LDS prototype.

6.2.1 Distribution Methods

This section describes the design of the distribution methods: initial placement of objects, migration of objects and replication.

Initial Placement Distribution Method

Support for initial placement is provided by the Central Factory. The Central Factory implements the `GenericFactory` interface from Fault Tolerant CORBA [FTCORBA]. This interface provides operations for the creation and destruction of objects, implementing the abstract factory design pattern [Gamma94]. Although in the prototype the Central Factory is a singleton object, i.e., only a single instance exists, the Central factory can easily be distributed should this be required for scalability reasons.

Clients use the `create_object()` operation to create an object of a specific type, specified by the CORBA repository identifier of the target's most derived type. Clients also specify the criteria that should be used when creating the object. Criteria are name-value pairs that allow clients to specify how an object should be created. An example of a criterion is an initialisation value for the object to create.

When a client that created an object wants to delete it, it can use the `delete_object()` operation. The IDL is shown below. We left out the exceptions and some of the type definition to make it more readable. The full IDL can be found in [Post02].

```
interface GenericFactory {
   typedef any FactoryCreationId;

   Object create_object(
         in TypeId type_id,
         in Criteria the_criteria,
         out FactoryCreationId factory_creation_id);

   void delete_object(
         in FactoryCreationId factory_creation_id);
};
```

The `create_object()` operation returns a reference that points to the newly created object and a creation identifier. This identifier should be retained by the creating entity so that it can delete the object using the `delete_object()` operation. The identifier is opaque, and only valid within the factory that produced the identifier.

A call to the `create_object()` operation on the Central Factory is delegated to a factory on the node where the object will actually be created. The selection of this local factory is based on the initial placement strategy for the type of object being created. The corresponding `delete_object()` operation at the end of the lifecycle is delegated to the local factory that created the object. Every local factory also implements the `GenericFactory` interface.

The Central Factory object also inherits the `FactoryManager` interface for the administration of local factories. Information that has to be provided when a factory is registered consists of the object reference of the factory, the location of the factory, the default criteria for object creation and the type identifiers that are supported by the factory. The `FactoryManager` interface also specifies operation for removing factories. Other operations that are supported and are also useful for a load distribution strategy are operations to retrieve information on a specific factory, to get a list of all object types supported by the Central Factory, and to get a list of local factories that support a specific type of factory.

```
struct FactoryInfo {
   GenericFactory the_factory;
   common::Location the_location;
   Criteria the_criteria;
};

interface FactoryManager {
   typedef any FactoryId;

   FactoryId add_factory(
      in FactoryInfo factory_info,
      in TypeIds type_ids);

void remove_factory(
      in FactoryId factory_id);

FactoryInfo get_factory_info(
      in FactoryId factory_id,
      out TypeIds type_ids);

common::TypeIds get_type_ids();

FactoryInfos get_factories_creating_type(
      in common::TypeId type_id);
};
```

Migration Distribution Method

The Component Migration Manager implements the migration distribution method. The Component Migration Manager uses the `migrate()` operation of the `ReconfigurationManager` interface of the Dynamic Reconfiguration Service (DRS), see Section 6.1.1. We can directly use this functionality and the other involved DRS components such as the Reconfiguration Agent and the Location Agent. We will not repeat the description of the implementation of this migration functionality.

Based on load information and QoS requirements, the load distribution strategy can instruct the Component Migration Manager to initiate the migration of one or more components to another node.

Replication Distribution Methods

The replication distribution methods are currently not implemented in the LDS prototype. Since our replication method, at least for stateless components, is similar to the one presented in [Othman01C] (see also the section on related work in Chapter 5), we are confident that the implementation can be done. Our load distribution mechanism extends [Othman01C] with the migration components that have session state. Below we briefly describe a possible implementation.

Replica and replica group management are provided by a Replication Manager, which is based on replication in the Fault Tolerant CORBA specification [FTCORBA]. The load distribution strategy interacts with the replication manager to decide on issues as number of replicas, placement of replicas, assignment of sessions to a specific replica, and the migration of sessions to another replica.

For the actual placement of replicas, for assignment of a session to a replica and for migrating a session, existing functionality of the initial placement and migration distribution methods can be re-used. This means that creation of a replica is done via a Central Factory, that delegates it to a local factory. Assignment of a session to a certain replica is controlled via a Location Agent. Redirecting a client to another replica (session migration) is done by instructing the load distribution agent to send a LocateForward exception to the client ORB. We implement the load distribution agent using a portable interceptor, contrary to [Othman01C] which proposes the use of a Servant Manager.

Although we could also implement per-request replication is a similar way as per-session and session-migration replication, we consider the associated overhead to be unacceptable. This overhead is per invocation, contrary to per session or per migration. Every invocation is send twice, once to the Location Agent and once to the replica.

6.2.2 Load Monitoring

Section 5.4 already gave an overview of the monitoring components. In this section, we zoom in on how we implemented this in the prototype.

Load Meters

As described in Chapter 5, the LDS supports both push and pull models for the load monitoring. As a result, we also have two different types of Load Meters, namely the Push Load Meter and the Pull Load Meter. We support the periodic model for the load monitoring by periodically pulling the load. There is common functionality for all load meters, especially functionality to identify them. We therefore let the interfaces for the push and pull load meters inherit from one base interface. The resulting UML class diagram is shown in *Figure 6-12*.

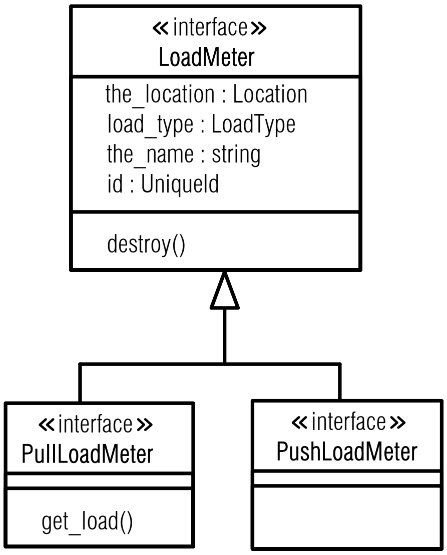


Figure 6-12 Load Meter class diagram

A LoadMeter provides operations for retrieving the location of the load meter (`the_location` attribute), the type of load that is measured by the meter (`load_type` attribute), the name of the load meter (`the_name` attribute), and a unique id for identification purposes (`id` attribute). The `destroy()` operation is called to destroy the load meter, and can include clean-up related tasks.

A location is described by using a name as defined by the CORBA Naming Service [CORNANS]. A name consists of one or more components, such that hierarchical names can be formed for identifying different locations. To make up a hierarchy, the following information is used: the IP address of the node, the ORB identifier, the name of the Portable Object Adapter, the object identifier, and the interface operation method name.

The `PullLoadMeter` interface provides an additional operation called `get_load()` to query for the current load value. The `PushLoadMeter` does not provide any additional operations, and pushes the load value to the local Meter Agent. Our current prototype does not include any push load meters.

Default Load Meters
The LDS provides four resource-layer load meters and two middleware-layer load meters, all based on the pull exchange model:

– *Standard CPU load meter* – This resource-layer load meter calculates the load by determining the percentage of time the CPU is executing non-idle threads. This information is collected using operating system specific system calls.
– *Average CPU load meter* – This resource-layer load meter calculates the load by measuring the CPU load over a configurable time interval (default is 1 minute). The CPU load is measured in the same way as the standard CPU load meter.
– *Memory load meter* – This resource-layer load meter calculates the load by determining the ratio of used and total available amount of memory. This information is also collected using operating system specific system calls.
– *Intrusive CPU load meter* – This resource-layer load meter calculates the load by counting the number of integer increments that can be done in a specific time interval. Calculating load in this way causes more overhead than using operating system specific system calls, but it has two benefits. The first is that it does not rely on operating system specific functionality. The second is that it is an absolute measurement of available processing resources. For example, suppose we have a heterogeneous environment with one very powerful node and one much less powerful node. Even in cases were the powerful node is loaded somewhat more than the less powerful node, it is typically better for performance to direct load to the more powerful node. Using a CPU measurement based on idle time might cause the load distribution strategy to make the wrong choice.
– *Average response time load meter* – This middleware-layer load meter calculates the load by measuring on the server side the time it takes to execute a certain object operation, and calculating the average over a configurable time interval (default is 10 seconds).
– *Invocation throughput load meter* – This middleware-layer load meter calculates the load by counting on the server side the number of invocations per configurable time interval for a certain object operation.

Load Information Exchange

The monitored data exchanged between Load Meters and a Meter Agent is captured in a Load Value. The following IDL code shows the type definitions of load values.

```
// TimeT is from the CORBA Time Service
typedef TimeBase::TimeT TimeT;

struct LoadValue {
   LoadType type;
   float value;
};
```

A Load Value consists of the type of load it represents, and the value of that load. Because load data is usually expressed in some numerical form (e.g., the percentage of the time a CPU is busy), a floating-point number is used to represent the load value. In some cases when the nodes have different capacities, the load values are not directly comparable, and have to be normalized.

Data exchanged between Meter Agents and Load Collectors is carried in Load Events. These Load Events describe the load values that have been measured, the location where that load was measured, and a timestamp that indicates when the load was measured.

Depending on the load meter, the location consists of one or more name components. For example, for a CPU load meter, a location consists only of the IP address of the node, because CPU load is shared between all objects on a node.

The timestamp re-uses the basic types from the CORBA Time Service [CORBATime]. Comparing timestamps makes sense if the timestamps refer to measurements performed on the same node, because no logical clock is used. We expect that using a logical clock [Raynal96] would cause an unacceptable overhead. Besides, we could not find a use-case in which a load distribution strategy would really need to compare timestamps between two different nodes.

The following IDL code defines Load Events:

```
struct LoadEvent {
   TimeT time;
   drs::Location loc;
   LoadValue value;
};
```

For load data exchange, the CORBA Notification service [CORBANot] is used. The Notification service supports both the push and the pull model. Because the Load Notifier basically acts as a channel for distributing events, it can be implemented using the Notification service. An additional benefit is that the Notification service provides the ability to filter events allowing consumers to only receive the events they are interested in.

Load Meter Registration

A Load Meter registers with the Meter Agent that exists at the same node as the load meter. Meter Agents support the `MeterAgentAdmin` interface, as defined in the following IDL:

```
interface MeterAgentAdmin {
   readonly attribute common::Location the_location;
   readonly attribute common::UniqueId id;

   LoadMeterId register_pull_load_meter(
      in PullLoadMeter load_meter);

   LoadMeterIds get_all_load_meters();
   LoadMeter get_load_meter(
      in LoadMeterId load_meter_id);

   void unregister_load_meter(
      in LoadMeterId load_meter_id);
};
```

The operations provided by this interface allow for the registration of load meters with the Meter Agent (`register_pull_load_meter()`), the removal of load meters from the Meter Agent (`unregister_load_meter()`), and the retrieval of registered load meters (`get_all_load_meters()` and `get_load_meter()`). The `MeterAgentAdmin` interface also provides a location attribute (`the_location`) and an identifier (`id`), both for configuration and identification purposes.

Load Collector

The Load Collector provides an interface for strategies to retrieve load data. The `LoadCollector` interface is defined as follows:

```
interface LoadCollector {
   monitoring::LoadEvents get_load_for_type(
      in monitoring::LoadType load_type);

   monitoring::LoadEvents get_load_for_locations(
      in monitoring::LoadType load_type,
      in common::Locations loc_seq);
};
```

The `get_load_for_type()` operation retrieves a list of load events for the specified load type. The returned list includes the last received load event of the specified type for each location that has load monitors for the specified type. The `get_load_for_locations()` operation retrieves a list of load events for the specified load type and list of locations.

The Load Collector collects load events by pulling them from the monitor channels, and storing the retrieved events.

Configuration of Load Meters
Load meters can be created by using a load meter factory. The interface of the load meter factory is as follows:

```
interface LoadMeterFactory {
   PullLoadMeter create_pull_load_meter(
      in LoadType load_type);

   PushLoadMeter create_push_load_meter(
      in LoadType load_type);

   PullLoadMeter create_named_pull_load_meter(
      in LoadType load_type,
      in string name);

   PushLoadMeter create_named_push_load_meter(
      in LoadType load_type,
      in string name);
};
```

Invoking the `create_pull_load_meter()` or `create_push_load_meter()` operation results in the factory creating a push or pull load meter of the specified load type. The factory assigns the name to the created load meter.

The `create_named_pull_load_meter()` and `create_named_push_load_meter()` operations allow the creation of load meters with a specific name. These methods are provided to support distinction between load meters that measure the same type of load by application provided names.

The `LoadMeterFactory` does not register the created load meters with the Meter Agent, nor is it used to destroy load meters. The registration is the responsibility of the creator of the load meter, as is the destruction.

In order to let the monitoring objects find each other, a simple object discovery system has been provided. It is event based, using a channel in the Notification Service to exchange the events. It is designed in such a way that it does not expect a specific order in which the different object are started, and new objects can easily be added during run-time. The events are `CREATE`, `DESTROY`, `QUERY` and `QUERY_REPLY`. Details can be found in [Post02].

6.2.3 Load Distribution Strategies

We define separate interfaces for the distribution methods that are supported by the LDS. All strategy interfaces derive from an abstract Strategy interface. *Figure 6-13* shows this.

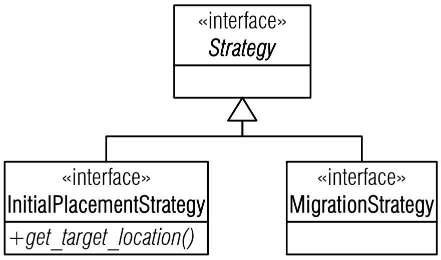


Figure 6-13 Interface inheritance for the different strategies

Strategies that use the initial placement method implement the `InitialPlacementStrategy` interface. This interface provides a single operation that asks the strategy for the target location for an object creation.

```
interface InitialPlacementStrategy : Strategy {
   common::Location get_target_location();
};
```

Strategies that use the migration distribution method have to implement the `MigrationStrategy` interface. This interface does not provide any operations because a migration is initiated by the strategy itself. A typical migration strategy uses the load data by querying the load collector to determine if a migration is needed. A strategy can use both the initial placement and the migration distribution methods by implementing both corresponding interfaces.

The LDS includes some default strategies for initial placement and migration. These are discussed in Section 6.2.5, where we also discuss their performance.

Strategy Manager

The Strategy Manager implements the `StrategyAdmin` interface, which provides operations to administrate the strategies. The `add_initial_placement_strategy()` operation is used to add a strategy for initial placement. An initial placement strategy is coupled to a specific object type. When a request is made on the Central Factory to create an object using the initial placement mechanism, the Central Factory queries the `StrategyAdmin` for the initial placement strategy for the type of the object being created by invoking the `get_initial_placement_strategy()` operation. After retrieving the initial placement strategy object, the `ObjectManager` queries the strategy for the target location for the object creation by invoking the `get_target_location()` operation on the `InitialPlacementStrategy` object.

```
interface StrategyAdmin {
   StrategyId add_initial_placement_strategy(
      in common::TypeId type_id,
      in InitialPlacementStrategy strategy);

      StrategyId add_migration_strategy(
         in MigrationStrategy strategy);

      void remove_strategy(
         in StrategyId id);

      Strategy get_strategy(
         in StrategyId id);

      InitialPlacementStrategy
         get_initial_placement_strategy(
            in common::TypeId type_id);
};
```

To add a migration strategy, the `add_migration_strategy()` operation is provided. Operations that add a strategy return a strategy identifier that can be used to remove a strategy (`remove_strategy()` operation).

6.2.4 Views on the Load Distribution Service

In this section, we summarize views of the different involved users of the LDS. We distinguish the following users:

- component designer – a designer of an object that is subject to load distribution
- client designer – a designer of a client of an object that is subject to load distribution
- strategy designer – a designer of a new load distribution strategy
- load meter designer – a designer of a new load meter
- system administrator – the person responsible for deploying an application and the LDS

Component Designer View

The transparency from the perspective of the component designer depends mainly on distribution methods that can be used with the object. For every distribution methods that are supported in the LDS, the component designer has to comply with the factory pattern. In addition, object should only interact with other objects via the ORB. We basically expect a CORBA object to be 'componentized', which goes beyond what CORBA 2.x requires (e.g,, stricter encapsulation and use of factory pattern). For initial placement these are the only transparency issues. The main additional transparency issue for migration is that the component designer has to provide state access methods. See Section 6.1.3 for an elaborate discussion on this.

Client Designer View

The LDS is completely transparent for the client designer. In case of the migration distribution method, the client-side portable interceptors do have to be installed and activated, but this is more a system administrator issue.

Strategy Designer View

A designer of a new strategy has to:

- inherit from the `InitialPlacementStrategy` and/or `MigrationStrategy` interface (see Section 6.2.3);
- if the initial placement distribution method is used, the strategy designer has to implement the `get_target_location()` method, through which the strategy receives the create load events, and which controls where new components are created (see Section 6.2.3);
- if the migration distribution method is used, the strategy designer has to invoke the `migrate()` operation of the Migration Manager to initiate migrations (see Section 6.2.1);
- use the `LoadCollector` methods to get access to load information (see Section 6.2.2).

Load Meter Designer View

A designer of a new load meter has to:

- implement the `PushLoadMeter` or the `PullLoadMeter` interfaces;
- use the `MeterAgentAdmin` interface to register the load meter (see Section 6.2.2);
- in case of a push load meter, invoke the `push()` method to send load values to the Meter Agent (see Section 6.2.2).

System Administrator View

When deploying an application that uses the load distribution framework, the system administrator has the following responsibilities:

- The Strategy Manager needs to be configured. A Java property file is used to specify a list of strategies that should be installed, and what parameters these strategies have. When the Strategy Manager starts, the strategies will automatically be installed.
- The application has to be started with the LD agent present. The LD agent is implemented using portable interceptors, which can be activated by a command-line parameter, or by small code modifications.

The same applies for middleware-layer load meters, which are also typically implemented as portable interceptors.

- Clients of the application have to have portable interceptors installed for the migration distribution method.
- Load meters that operate on the resource layer are created using a resource load meter factory. This resource load meter factory is a separate executable that should run on nodes where these load meters will be created.
- The exchange of load information also requires the meter agent to be run on each node that may have load meters installed.

6.2.5 Applications and Measurements

This section describes the types of applications and measurements used to compare the effects of different load distribution strategies and mechanisms on several application types. Since we did not implement the replication method, we do not discuss it in this section.

Application Types

As we motivated in Chapter 5, it depends on the application and the environment it runs in which distribution method and which strategy will be most effective in distributing the load to fulfil the specific QoS requirements. It is beyond the scope of this thesis to determine what are the exact parameters, which determine what distribution method and strategy are the most effective. We instead take a framework-based approach that allows easy extension of the LDS to fit a specific application and/or environment. We will however make a common distinction between session and service type of component to describe the tests we did.

According to [Henning99], applications typically fall into one of two general categories:

1. *Service-oriented applications*. Service-oriented applications tend to support persistent objects that are long-lived and stable. Different clients may access such an object, possibly interleaving requests. An example of a service-oriented application is the CORBA Naming Service.
   A further distinction can be made for service-oriented applications:
   - Long-lived objects that have state. These objects are from this point on referred to as ‘stateful’ objects.
   - Long-lived objects that do not have state. These objects are referred to as ‘stateless’ objects.
2. *Session-oriented applications*. Other server applications may be designed in such a way that clients first create the objects they tend to use, use those objects and then destroy them. These applications are session-oriented because most objects only live as long as the clients need them. Clients

create objects programmatically by using an object factory. The objects are referred to as 'session' objects. The concept of session objects can also be found in the Enterprise JavaBeans architecture [EJB].

Migration mechanism are likely to work better with service-oriented applications, because objects in such applications are long-lived and as such could benefit in the long term from moving between different nodes. Session-oriented applications would intuitively benefit less from migration since they are generally short-lived, and the overhead would be too big compared to the potential gain that could be achieved with the migration. Both would benefit from initial placement, especially since the overhead is relatively small.

Test Environment

The load distribution framework provides a set of classes and applications that can be used to create different application scenarios that are easily reproducible.

Application scenarios are specified by scenario files. Scenario files are XML [XML] files that contain definitions of client and server applications, and how these client and server applications interact. See [Post02] for the XML Schema.

The scenario file allows the specification of zero or more server applications. Server applications have several attributes: a name, the address of the node the server runs on, the location of the configuration file and a flag for verbose logging output.

When a scenario is executed by running the server application of the test environment, the scenario file is searched for a definition of a server application for the node where the scenario is executed. This allows different servers on different nodes to be specified in a single scenario file.

Server Side

A server can contain object factories, or it can contain a hierarchy of Portable Object Adapters (POAs). Every POA may have one or more objects created on the POA.

For factories, the scenario file allows the specification of the type of objects created by a factory (by specifying a CORBA IDL repository identifier), and the default criteria that are used when an object is created by the factory. The factory is added to the Factory Manager when the scenario is executed.

The most important part of the specification of an object is the specification of the properties of the object's service time. The service time determines the type and length of the work executed when a particular operation (called `do_some_work()`) of the object is invoked.

The test environment allows two types of work: CPU time, and waiting-time. With CPU-time work, the `do_some_work()` operation performs mathematical operations that stress the CPU of the node where the server application is running. With waiting-time work, the operation waits for a specified amount of time, doing nothing.

With these two types of service time, it is possible to create objects that perform CPU intensive tasks, or perform tasks that do a lot of waiting (for example waiting for I/O events).

The scenario file also allows the specification of the length of the service time. The length can be some constant value, or a value that is distributed according an uniform or exponential distribution.

Client Side

The test environment also allows the specification of the client side. A client consists of one or more threads of execution that are running concurrently. Each thread consists of three stages: creation, usage, and removal of objects. In the creation stage the thread can create objects it will use. In the removal stage it has to destroy them.

In the usage stage a thread invokes operations on one or more target objects. The objects whose operations are invoked can be objects that have been created in the creation stage, but can also have been created by other means. The scenario file specifies the number of requests that the thread makes, the statistical distribution that specifies the time between each request, and the targets of the requests.

The specification of a distribution of the requests made by the thread can be throughput (do not wait between invocations), exponential (wait with an exponential distribution) and burst (alternate fast and slow invocation rate).

A target specifies the object that receives the invocation, and the name of the method that will be invoked. It is possible to specify a set of target objects, and assign weights to each target. The client will then use choose between the targets, based on their weight.

A limitation of the test environment is that a thread can only execute one sequence of creation, usage and removal stages. It is not possible for a thread to create an object, make invocations on the object, delete the object, create a new object, make invocations, etc. The consequence of this limitation is that we are limited in our ability to do measurements in scenarios that simulate session-oriented applications with clients that create, use and destroy more than one object.

All threads can start simultaneously, but each thread can also be started at different times by having each thread wait for a different event from a CORBA notification channel.

Test Setup

The test setup consists of five machines in total. All machines are connected via a 100Mpbs switched Ethernet LAN. One machine is used for hosting the notification service and channels. Two machines are used for running server applications, one machine is used for running the load distribution framework components, and one machine is used for running the client application.

The hardware configuration and role of each machine is listed in *Table 6-1*.

Table 6-1 Machine configuration

Name	CPU type	CPU speed (MHz)	Memory (MB)	Operating System	Role
Machine A	2x Pentium III	2x 1000	512	Linux with kernel 2.2.18	Notification server
Machine B	Pentium II	400	192	Windows 2000 Professional	Server location 1
Machine C	Pentium II	400	256	Windows 2000 Professional	Server location 2
Machine D	Pentium III	866	256	Windows 2000 Server	Framework components
Machine E	2x Pentium III	2x 1000	512	Windows 2000 Server	Client

The ORB used is IONA's ORBacus version 4.1.0 for Java, together with Sun's Java2 Platform, Standard Edition JDK version 1.3.1_01. The Notification Service implementation used is IONA's OBNotify version 2.0.0, together with ORBacus version 4.1.0 for C++.

The timer used for measuring response times has microsecond precision. The normal Java timers are not very accurate, so we use high-precision timer capabilities provided by the operating system. These operating system timers are accessed using the Java Native Interface.

To measure the time the execution of the timer itself takes, we did a test in which we executed a loop that starts and stops the timer 1000 times. This test indicated that the overhead every time we run the timer code is 6.5μs. This overhead is negligible compared to the response time of an invocation on a remote object.

LDS Overhead

The overhead for using the LDS consists of several categories:

– *Overhead for creation of components* – Because object creation happens via the Central Factory which delegates to local factories, an additional delay is introduced for every object creation. This delay however is quite minimal, and only occurs once in the lifetime of a component.
– *Overhead caused by the request forwarding mechanism* – At the start of a session, i.e., for the first invocation of a client with some server object, the client-side middleware will first contact the Location Agent that redirects it to the actual location of the server object using the standard CORBA request forwarding mechanism. After the client middleware receives the (new) location of the object, subsequent invocations will directly go to the object. The client middleware only returns to the location agent in case of a migration. The extra delay introduced by this depends on the network latency, but it is typically small and is only for the first invocation of a session, and for the first invocation after the migration of an object (or session).
– *Overhead caused by the instrumentation for the migration* – The migration distribution method requires instrumentation, for which we used CORBA Portable Interceptors. This overhead is fixed (independent of the parameter size, method etc.), see also Section 6.1.5.
– *Overhead caused by the load monitoring* – The load is monitored by Load Meters, and transported via the Meter Agent and Load Notifier to the Load Collector. All this takes processing and network resources. The actual overhead depends on the type of load meter, the number of load meters, the type of derived information that has to be calculated (such as averages) and the frequency of the exchange of load information.
– *Performing a component migration* – The migrating of a component causes a temporary disruption of execution, which can, depending on the application, be considerable. Invocations initiated by the clients of the migrated component after the beginning of the migration and before the end of the migration are queued. This causes an increase in response time that is dependent on the application. Since we wait for ongoing invocations involving the affected component to finish, the expected increase in response time is proportional to the expected duration of these invocations. Therefore, this increase is higher for applications with long-lived invocations, see also Section 6.1.5.

Since the interceptors are in the invocation path for every invocation, their overhead is linear to the amount of invocations, and they cause an additional delay for every invocation. We therefore consider this the primary source of overhead. To quantify this category of overhead, we did four different tests:

1. Application scenario with no interceptors,

2. Application scenario with empty interceptors, i.e., no implementation code in the interception points,
3. Application scenario with Dynamic Reconfiguration Service interceptors,
4. Application scenario with a load meter implemented as interceptor.

Measurement 1 provides a baseline for the other measurements. Measurement 2 provides details about the costs of using interceptors incurred by the ORB implementation. Measurement 3 shows the cost incurred by the interceptors provided by the load distribution framework. Measurement 4 determines the cost for using a load meter implemented as interceptor.

These tests repeat some of the performance evaluation tests we did for the DRS tests (see Section 6.1.5). We repeat them here to give a complete overview of the overhead, because we made some minor changes to the DRS and we used a more accurate native timer here instead of the Java timer.

Overhead is measured by measuring the response time R of each invocation, observed at the client. Because the operation invoked on the server object contains no application code, R is approximately equal to the delay introduced by the ORB implementation and the network.

For test case 1, the delay introduced by the middleware layer as a whole is equal to the delay introduced by the ORB implementation (including network delay), because no interceptors are installed. For test case 2, the delay introduced by the middleware consists of the delay introduced by the ORB and the delay introduced by the implementation of portable interceptors. For test case 3, the delay consists of the ORB delay, the interceptor delay, and the delay introduced by the DRS implementation. Test case 4 shows the additional delay introduced by an interceptor load meter that monitors the number of requests made in the last 10 seconds. We make the assumption that other interceptor based load meters will cause a similar delay. Test case 4 also includes the meter agent running on the server machine and the exchange of load data between the meter agent and the load collector.

The following list summarizes how the response time is built up from the different delays in each test case:

1. $R = \Delta ORB$
2. $R = \Delta ORB + \Delta interceptors$
3. $R = \Delta ORB + \Delta interceptors + \Delta DRS$
4. $R = \Delta ORB + \Delta interceptors + \Delta DRS + \Delta loadmeter$

Five batches of 25000 requests for each of the four test cases have been executed, with two different sizes for parameter and return value of the invocations. The results obtained from tests 1 – 3 are shown in *Table 6-2*. The values are the averages of 5 x 25000 invocations.

Table 6-2 Results for test cases 1 – 3

	No interceptors	Minimal portable interceptors		Dynamic Reconfiguration Service			
Size of parameters and return value	ΔORB (μs)	Δinterceptors (μs)	increase from ΔORB	ΔDRS (μs)	increase from ΔORB + Δinterceptors	Δinterceptors + ΔDRS (μs)	increase from ΔORB
0 bytes	988.9	69.96	7%	184.9	17%	254.8	26%
2048 bytes	2392	75.02	3%	180.0	7%	255.0	11%

With a parameter and return value size of 0 bytes, overhead incurred by the DRS interceptors is 26%. This is a worst case scenario with a parameter size of 0 bytes, and no servant execution time. With a larger size for parameters and return values (2KB), the overhead becomes 11%.

Because the interceptor overhead is constant, relative overhead decreases as overall processing time for (de)marshalling of parameters and for the servant implementation increases.

Another issue to keep in mind is that the DRS and framework interceptor implementations are not optimised. Optimising the common execution path and the data that is sent with each request in the service context could further reduce the overhead associated with the interceptors. Another opportunity for optimisation lies in the fact that DRS interceptors are only needed for migration of objects. *Table 6-3* shows the results for test case 4.

Table 6-3 Results for test case 4

	No interceptors	Dynamic Reconfiguration Service and interceptor load meter			
Size of parameters and return value	ΔORB (μs)	Δloadmeter (μs)	increase from ΔORB + Δinterceptors + ΔDRS	Δinterceptors + ΔDRS + Δloadmeter (μs)	increase from ΔORB
0 bytes	988.9	45.52	4%	300.3	30%

The results indicate that adding an additional interceptor for monitoring the load increases the response time 4% from the situation where the DRS interceptors are installed. One should keep in mind that the results presented here are obtained for a single ORB implementation. Other ORB implementations may have a different overhead.

Initial Placement Measurements

The measurements for the initial placement mechanism are based on a single application scenario that is tested with two static and two dynamic initial placement strategies.

1. *Random* – a static strategy that uses a random algorithm, i.e., the strategy randomly chooses a location out of the set of available locations.
2. *Round-robin* – a static strategy that uses a round-robin algorithm, i.e., the strategy rotates through the available locations.
3. *Least-loaded CPU* – a dynamic strategy that selects the location with the lowest CPU idle time. In case of overload CPU idle time will be (almost) 0 for all locations, and a round-robin algorithm is used. This is thus actually an adaptive strategy.
4. *Least-loaded ORB* – a dynamic strategy that uses a middleware-layer load meter that monitors the amount of requests that are being processed. Since there is no hard upper limit for this, we do not need to adapt the behaviour in case of an overload situation.

The DRS interceptors are enabled for each measurement to create equal test conditions, even though since we do not migrate components in this test we do not need them. The random and round-robin strategies are strategies that are often used in commercial applications, and are therefore included in the initial placement measurements.

The application scenario simulates five clients on the same machine by executing five threads of execution within a single client on machine E. The threads are started consecutively with a one second delay between each start signal. Each thread creates a Session object using the initial placement mechanism. Each thread makes 1000 invocations on the `do_some_work()` operation of the object it has created. The `do_some_work()` operation has CPU-type service time with a service time length of 100ms.

The application scenario simulates a session-oriented application (with one client per server) in an overloaded situation. The client machine is fast, so that the execution of the five threads simulating the five clients is not limited by the available processing resources on the client machine. By starting the threads with a fixed time interval between them, the scenario tries to gradually increase the load on the two locations.

Random Initial Placement Strategy

Because this strategy assigns objects to locations in a random fashion, each run of the test scenario will produce a different object allocation. For large number of objects, this will typically not be an issue, but in our test scenario which has only five server objects, this can easily result in a non-optimal allocation of objects. For example, one test run resulted in a situation with four objects created on location 2, and one object on location

1. Because of the random nature of the strategy, in theory it is possible that all objects are created on a single location.

Round-robin Initial Placement Strategy
The round-robin initial placement strategy determines the target location of an object creation in a rotating fashion. This means that first location 1 is used as target when a client invokes `create_object()` on the object manager. The second object creation is delegated to the factory on location 2. The third object creation will be delegated to location 1, etc. Like the random initial placement strategy, the round-robin strategy is a static strategy that does not use load data from location 1 and location 2.

We ran the tests in an environment with homogenous nodes for the server objects, but in case of heterogeneous servers a static strategy with weights for each location is more appropriate. The weight determines the relative chance the location will be chosen as the target of an object creation, and can be related to the level of load a location can handle. For example, by giving location 1 a weight of 0.25 and location 2 a weight of 0.75, statistically 25% of the object creations happen on location 1, and 75% happen on location 2.

Initial Placement Strategy using CPU load
The CPU based initial placement strategy uses the CPU load data from location 1 and location 2 to determine which location should be the target of an object creation. For this purpose, a load meter is used that measures the average percentage of time the processor is busy.

The initial version of this strategy exhibited a *high static load instability* (see Chapter 5) type of flaw when all locations had a high load with a small variation. The strategy overloaded one of the nodes, instead of an even division of the load over the different nodes. A run of the application scenario would result in the first object being created on location 2, the second object on location 1, and the third, fourth, and fifth objects being created on location 2 again. This very unbalanced allocation of objects over the available locations resulted in an average response time for location 2 that was more than twice as high as for location 1.

The reason for this behaviour is largely dependent on the nature of the load type: the CPU load value has a maximum value of 100%. If all locations are overloaded, and thus report the maximum load value, no distinction can be made between the locations based on the reported load. We therefore made the strategy adaptive, and switch from a least-loaded to a random strategy in an overload situation.

Initial Placement Strategy using Number of Active Requests

This dynamic strategy makes its decision based on number of active requests, which means that both location 1 and location 2 have load meters that monitor this load information.

Strategies Comparison

We compare the different scenarios by comparing the average response times for the different strategies. We leave out the random strategy here because in this test scenario it at best gives a average response time that is equal to the round-robin strategy, and typically worse because of the small amount of server objects in this test scenario. All three strategies resulted in the most optimal distribution of objects for this scenario: 2 objects on one machines, and 3 objects on the other.

Table 6-4 shows the average response times for all invocations from all five simulated clients for each strategy.

Table 6-4 Average response time for the initial placement strategies

Strategy	Average response time (ms)
Round-robin	170.2
Active requests based	174.0
CPU-based	172.7

The table shows that the three strategies perform equally well, as could be expected since they divide the objects in the same way over the two locations. The minor differences can be caused by the overhead of collecting the load information, which is apparently almost negligible in this test scenario.

Migration Measurements

We did two tests with strategies that use the migration distribution method. The first test determines, for a specific scenario, the overhead of migration is general, and compares it to not using the migration distribution method at all. The second test determines the delay caused by a migration for objects, i.e., what the temporary performance penalty is for a migration.

Migration versus No Migration

The application scenario for this measurement is as follows: three objects are created on the two available locations: one object at location 2 (machine C), and two objects at location 1 (machine B). For each of the three objects `do_some_work()` has a CPU-type service time, and a service time length of 100ms. The objects each have a state of 1KB.

The client (on machine E) consists of three threads. Each thread makes 1000 invocations of the `do_some_work()` operation on a single object.

Each thread is started with a 1 second interval. The framework components run on machine D.

Each of client threads initiates its requests in bursts, i.e., alternating periods with many and with few requests. The length of a burst is fixed to 100 requests. The time between each request is based on a random variable with an exponential distribution with rate of 0.05ms^{-1}, resulting in an expected value of 1 / 0.05 = 20ms.

For each period with few requests, the value obtained from the random variable is divided by a scale factor. For each period with many requests, the value obtained from the random variable is multiplied by the scale factor. This leads to alternating periods of a fixed number of requests with lower waiting time between requests and higher waiting time between requests.

In the application scenario the scale factor is set to 10. This results in alternating periods of requests with 20 / 10 = 2ms average between each request and requests with 20 * 10 = 200ms average between each request.

The application scenario is executed both with a migration strategy installed, and without a migration strategy installed. The migration strategy bases its decision to migrate an object on the CPU loads (idle time) measured on both locations. The strategy retrieves the load of the two locations from the load collector every 2.5 seconds. If most recent reported load of each location differs by more than 25 percentage points, a random object is migrated from the location with the higher load to the location with the lower load. To increase the stability for the strategy, and to try to avoid thundering herd or processor trashing effects, the strategy waits at least 5 seconds after finishing one migration before initiating a new migration.

Figure 6-14 shows the average response times from the client perspective for each of the client threads. The average response time over all threads for both scenarios is about equal (within 1%). However, in the case of migration there is less difference between the response times of the different threads. The reason for the difference in average response time in the test without migration is that one node will host two objects, and one node hosts one object. As can be expected, the node with two objects will have a higher average response time. In exact numbers, with migration the difference between the average response time of thread1 (minimum) and thread2 (maximum) is 5ms, without the migration strategy the difference between thread1 (minimum) and thread3 (maximum) is 10ms.

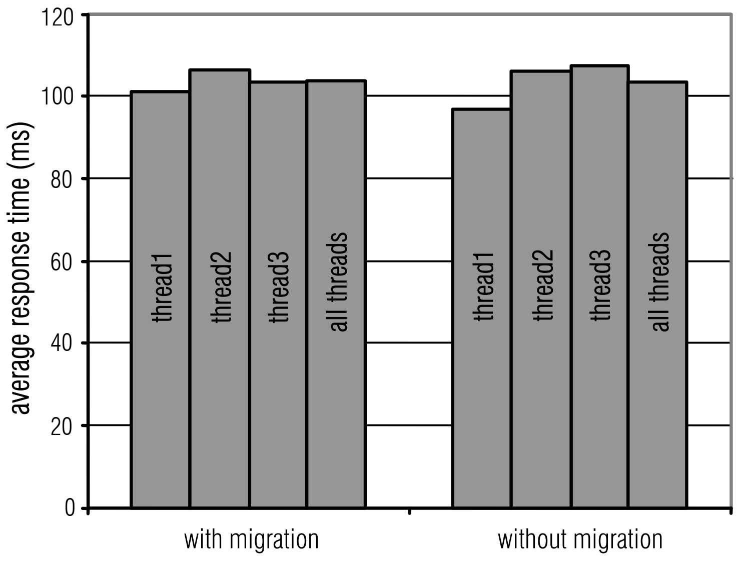


Figure 6-14 Average response times for a migration scenario

During the execution of the scenario with migration enabled, 22 migrations take place, and the average time for a migration is 312ms.

The measurements obtained from this application scenario indicate that migration in this case has an overhead that is equal to the performance gain from using the strategy. The application scenario with a migration strategy performs about equal to the same scenario without a migration strategy.

Migration of Stateless versus Stateful Objects

The application scenario used in this measurement is the following: an object (either stateless or stateful) is created at location 1. The object has a CPU-type service time with a service time length of 100ms. The object can be migrated between location 1 and location 2, which means that both locations have a factory that can create the object. A single client consisting of five threads invokes the object's `do_some_work()` operation. Each thread makes 1000 invocations after being started. The threads are started with a one second interval. The node configuration is the same as for the initial placement measurements.

The application scenario simulates a service-oriented application that is accessed by five clients. The service object is an object that does a CPU intensive task. The migration strategy used in this test is based on the percentage of time the processors are busy executing non-idle threads. The strategy retrieves the load of the two locations from the load collector every 2.5 seconds. If most recent reported load of each location differs by more than 25 percentage points, a random object is migrated from the location with the higher load to the location with the lower load. To increase the stability of the strategy somewhat the strategy waits at least 5 seconds after a migration before initiating a new one. The strategy however does not attempt to prevent oscillating behaviour, since we need many migrations to get better measurements.

Table 6-5 shows the average duration of a migration both for stateless and stateful objects. A stateful object has a state of 1024 bytes. The average values are obtained from executing the scenario three times.

Table 6-5 Duration of object migration

	Average duration of a migration (ms)	**Standard deviation (ms)**
Stateful (1024 bytes)	141.0	39.22
Stateless	130.7	78.20

These numbers shows a significant overhead for a migration. Because the processing of a single invocation takes the server object 100ms, and all invocations have to finish processing before the actual migration can take place, this is also to be expected.

Migration of a stateful object takes on average 7.88% longer than migration of stateless object in this particular scenario. The standard deviation for both stateless and stateful object migration is quite large, which means there is a large variation in the duration of migration. This can be explained by the fact that the duration will depend on the invocation which takes the longest to finish, which can be anywhere in the range of 0 till 100ms.

Because of the load imposed by invocation of the `do_some_work()` operation and the fact that only a single object exists, the migration exhibits oscillating behaviour: the object is constantly being migrated from the heavy loaded location to the lightly loaded location. The migration strategy we used for this test does not prevent this oscillating behaviour from happening. More advanced migration strategies should be more stable, and should detect this type of instable behaviour.

6.2.6 Evaluation

The Load Distribution Service prototype has been successfully tested, including the tests described in Section 6.2.5. Although the prototype does not implement our complete load distribution mechanism, it implements and thus validates the most essential parts.

We evaluate the prototype by discussing the performance and transparency, and we mention some future work on the LDS.

Performance and Overhead

The measurements for the overhead of the LDS show that the overhead of using the LDS appears to be large (±25%), but because the overhead is constant, and the tests were performed for a worst-case scenario (no service time, no parameters and return value), it is expected that overhead for most applications is acceptable. Moreover, the interceptors used by the DRS and LDS are not optimised. It is expected that optimisations are possible for the DRS and LDS interceptors.

The measurements performed for migration strategies are too limited to make conclusions for different applications in general. The measurements do show that migration can be an expensive operation, because it takes network capacity and processing power and, more importantly, temporarily suspends part of the system. The temporary degradation of part of the system during a migration might be unacceptable if the QoS requirements have a hard upper limit for the response time. More tests for different types of applications would be necessary to get more insight in when the performance gains outweigh the costs associated with a migration.

Transparency

The LDS is very transparent from the perspective of the client, which we consider essential. Also for a designer of a component there are only limited violations of the transparency. The main issues with respect to transparency have to do with the state access, as discussed in Section 6.2.4.

Future Work

The prototype should be extended to implement the missing part of our load distribution mechanism, especially the replication distribution method. The replication distribution method is subject to standardisation within OMG, as mentioned in Chapter 5. Assuming this standardisation is successful, our implementation should comply with this standard.

We expect that our load distribution mechanism and this prototype can easily be used for the newer generation of component middleware, such as Enterprise JavaBeans [EJB] and CORBA Component Model [CCM]. We plan to validate this by porting our current CORBA 2.x based prototype.

We had scalability as a requirement for our load distribution mechanism and prototype, which resulted, for example, in a hierarchical monitoring design. Some parts of the prototype however are now physically centralized, such as the Strategy Manager and the Central Factory. These are potential scalability bottlenecks. There are no inherent limitations to our mechanism or high level design that prohibit distributing these components, but we could modify our prototype to actually distribute them.

Chapter 7

Conclusions

This chapter presents the contributions and conclusions for the research presented in this thesis, and suggests directions for future research.

Section 7.1 briefly describes the background for this thesis. Section 7.2 presents the major contributions. Section 7.3 details the contributions per chapter. Section 7.4 presents our conclusions. Section 7.5 suggests directions for future research.

7.1 Introduction

The ability to control the Quality of Service (QoS) is essential for the success of distributed systems. Controlling QoS is a complex problem since it concerns all the functional layers we consider in this thesis, which are the application, middleware and resource layers. Controlling QoS is especially complex for large-scale systems such as telematics systems due to heterogeneity and scalability issues. The term QoS can denote a wide range of characteristics of a system, but in the context of this thesis we limit QoS to the performance characteristics (response time and throughout), and the availability characteristics (maintainability, reliability and availability).

Large-scale distributed systems are often built using component-middleware technology (e.g., CORBA) because of the distribution transparency it offers. With distribution transparency, complexities related to the distribution are hidden from the application developers by embedding the distribution aspects in the middleware. We focus on component-middleware-based applications in this thesis.

7.2 Major Contributions

This thesis describes *how to develop middleware-layer QoS mechanisms that improve the availability and performance QoS characteristics of component-middleware-based applications*. By embedding the QoS mechanisms in the middleware layer we hide their complexity from the developer of the application components, and allow re-use of the QoS mechanisms.

The major contributions are:

1. This thesis proposes an overall approach for the design of dynamic QoS mechanisms for component-middleware-based applications. These middleware-layer QoS mechanisms do not rely on resource-layer QoS mechanisms (Chapter 3).
2. This thesis describes how message reflection is used to obtain separation of concerns between component developers, QoS mechanism developers and middleware developers. Message reflection is essential in ensuring the transparency of our QoS mechanisms. We have identified, compared and evaluated the suitability of different techniques to implement message reflection in middleware (Chapters 3, 4, 5).
3. This thesis proposes a new dynamic reconfiguration mechanism that is suitable for component-middleware-based applications, and that maximizes transparency. It preserves correctness and is embedded in the middleware layer. Dynamic reconfiguration improves the availability characteristics of an application (Chapter 4).
4. This thesis proposes a new load distribution mechanism that incorporates different distribution methods, is extensible with new load distribution strategies and load information types, and allows QoS differentiation. Load distribution improves the performance characteristics of an application (Chapter 5).
5. This thesis describes an integrated prototype of our dynamic reconfiguration and load distribution mechanisms that serves as a proof of concept. This prototype is based on CORBA [CORBA], a common off-the-shelf middleware technology, and uses Portable Interceptors to implement the message reflection. Measurements with the prototype give insight into the performance aspects, and show that the overhead is acceptable for most applications (Chapter 6).

7.3 Contributions per Chapter

In this section, we detail the contributions per chapter.

Chapter 2 – QoS and Component Middleware: an Overview

1. Proposes a model of component middleware, and defines QoS for component-middleware-based applications. Our model generalizes existing component-middleware technologies, and uses parts of RM-ODP [RMODPPart3] and Szyperski's [Szyperski98] component definitions.
2. Gives a state-of-the-art overview of approaches in the area of QoS for middleware-based applications.

Chapter 3 – QoS Mechanisms in the Middleware Layer

1. Identifies generic requirements for QoS mechanisms from the perspectives of the application-component developer and the QoS mechanism developer.
2. Discusses the merits of static versus dynamic QoS mechanisms for large-scale distributed systems, and motivates why we select a dynamic approach.
3. Makes a division of QoS mechanisms in middleware-layer-internal and mapping mechanisms, and motivates our choice for middleware-layer-internal mechanisms.
4. Discusses how message reflection can be used to achieve a strict separation of concerns for our QoS mechanisms, and evaluates and compares different techniques to implement message reflection.

Chapter 4 – Dynamic Reconfiguration

1. Presents a model for dynamic reconfiguration that is used to describe existing approaches and that is used as the basis for our dynamic reconfiguration mechanism.
2. Gives an overview of the issues that arise when implementing dynamic reconfiguration, and how they affect component-middleware-based applications. We give special attention to the correctness requirements: structural integrity, mutually consistent states and application-state invariants.
3. Contains an extensive overview and comparison of the state-of-the-art in dynamic reconfiguration, including their (lack of) suitability for component-middleware-based applications.
4. Proposes a new mechanism for dynamic reconfiguration of component-middleware-based applications that is more transparent for application developers than existing approaches.
5. Proposes a refinement of the model for dynamic reconfiguration. This refined model separates configuration information into information that can be obtained at run-time, and information that has to be provided by the application developer.

Chapter 5 – Load Distribution

1. Presents a model for load distribution, and, based on this model, presents the issues that arise when implementing load distribution.
2. Discusses the limitations related to state synchronization when using the replication distribution method, and compares replication and non-replication distribution methods.
3. Identifies the different distribution methods, and their suitability for different types of components.
4. Gives an extensive overview and comparison of the state-of-the-art in load distribution, which includes identifying issues that are not covered in existing approaches.
5. Proposes a new mechanism for load distribution in component-middleware-based applications that incorporates a wide range of load distribution methods, and is extensible with new load distribution strategies and new types of load information.
6. Proposes how to use our load distribution mechanisms for QoS differentiation. We do this by dividing the available nodes in classes. We support both class-based performance requirements and quantitative performance requirements.

Chapter 6 – Proof of Concept

1. Describes an integrated prototype that serves as a proof of concept for the proposed dynamic reconfiguration and load distribution mechanisms. This prototype is developed in CORBA and Java.
2. Presents measurements on the performance and overhead of the prototype.

7.4 Conclusions

The conclusions are structured based on the objectives for this thesis as they are identified in Chapter 1.

Our Approach for QoS Mechanisms

We advocate a dynamic approach to QoS provisioning using the middleware layer. By putting the QoS functionality in middleware-layer QoS mechanisms, it is possible to better separate the QoS concern from the application logic. The middleware-layer QoS mechanisms hide as much as possible the complexities of QoS provisioning from the application component developer. In addition, this facilitates re-use of the QoS functionality.

Mapping and middleware-layer-internal QoS mechanisms

Middleware-layer QoS mechanisms can be divided into middleware-layer-internal QoS mechanisms and mapping QoS mechanisms. Middleware-layer-internal QoS mechanisms are internal to the middleware layer, and enhance the QoS by using the functions that the middleware provides. They do not rely on resource-layer QoS mechanisms. Mapping QoS mechanisms on the other hand do map to resource-layer QoS mechanisms. These two categories of mechanisms are complementary and they can be used together to provide the required QoS. This thesis focuses on the design of middleware-layer-internal mechanisms.

We chose to apply our dynamic and middleware-layer-internal approach to two different QoS mechanisms: a dynamic reconfiguration QoS mechanism that improves availability, and a load distribution QoS mechanism that improves performance.

Dynamic Reconfiguration QoS Mechanism

We have shown that it is possible to develop a dynamic reconfiguration QoS mechanism that allows runtime upgrades of a component-middleware-based application while preserving correctness. Since dynamic reconfiguration prevents disruptions of an application, it increases the availability characteristics (mean-time-between-disruptions, uptime, mean-time-to-repair).

Our mechanism supports replacement of a component with a newer version, migration of a component to another node, adding a component and removing a component, without taking the application instance as a whole offline. It also supports composite reconfigurations in which several components are reconfigured in an atomic action from the perspective of the rest of the system.

Multi-threading , state and re-entrance

Our mechanism is applicable to a broader range of applications than existing solutions. We support multi-threaded components, re-entrant components, and stateful components. Special care is taken to minimize the impact on execution during reconfiguration, and to be scalable with respect to the number of clients.

Our mechanism provides full reconfiguration transparency to client developers, requires only minimal reconfiguration expertise from the reconfigurable-component developer and does not require the use of additional formalisms for application development.

An important characteristic of our dynamic reconfiguration mechanism is that it preserves correctness, viz., mutually consistent states, structural integrity and application-state invariants.

Driven safe state

To preserve correctness, components need to be in a reconfiguration-safe state before a reconfiguration can be applied. We drive the component to a reconfiguration-safe state in which the component is not involved in any invocations. Ongoing invocations are completed. New incoming invocations are intercepted, and queued if they can be processed after the migration. To prevent a deadlock, re-entrant invocations cannot be queued, i.e., the queuing has to be selective. We detect a re-entrant invocation by adding an implicit parameter to (nested) invocations that identifies the invocation path. Queuing and selection of invocations, and adding implicit parameters to invocations, can be done using message reflection, e.g., in CORBA by using Portable Interceptors.

Selective queuing

Load Distribution QoS Mechanism

We have shown that it is possible to develop a load distribution QoS mechanism that distributes components over a set of nodes in such a way that certain performance requirements are met. There are different distribution methods, viz., initial placement, migration and replication.

Framework-based solution

Since the optimal load distribution is dependent on specific characteristics of an application and of the environment it is deployed in, we propose a framework-based solution in which it possible to easily add new load distribution strategies and load information types.

QoS differentiation

An important aspect of our load distribution mechanism is that it allows both load sharing and QoS differentiation, while other solutions are limited to load sharing. The QoS differentiation is based on classes of nodes, and components and replicas are placed in or migrated to a certain class based on the QoS requirements.

Replication

When replication is used as a distribution method, the states of all replicas have to be consistent. This means that every change to the state of one replica has to be applied simultaneously to all other replicas. Since this can cause an unacceptable overhead, this makes little sense if performance improvement is the goal of the replication. For replication to actually improve performance, the consistency requirements have to be relaxed. This is usually referred to as loose consistency. Most types of loose consistency however violate transparency because they expose the component developer to the complexity of the consistency mechanisms that are used. We therefore propose to use replication only for those types of components that allow a loose consistency without violating transparency: components that have only session state, and components that are stateless. Components with session state only have state between invocations in the same session. Because of this, state changes are local to a specific replica, which from a consistency perspective is similar to replication for stateless components. Transparent replication for stateful components that have state between invocations (global state) is not beneficial from a performance perspective.

Using Reflection to Achieve Separation of Concerns

Reflection, and in particular message reflection, can be used to achieve separation of concerns. It allows the instrumentation that is required for QoS mechanisms to be separated from the application code, and also from the middleware code. This increases the transparency of the QoS mechanism. Although current common middleware technologies only provide limited reflective capabilities, we have shown that for our purpose we can use message reflection with common middleware.

Transparency

Our evaluation of possible techniques to implement message reflection shows that the middleware interceptors are currently the most suitable technique. Middleware interceptors can intercept incoming and outgoing requests and replies. Benefits of middleware interceptors are that (i) they can be added at deployment or even run-time, (ii) they also work for components located in one capsule or on one node, and (iii) they do not require changes in the middleware code.

Proof of Concept

We provide a proof of concept of the research presented in this thesis through an integrated prototype: the Dynamic Reconfiguration Service implements our dynamic reconfiguration mechanism, and the Load Distribution Service implements our load distribution mechanism. This prototype is implemented in CORBA and Java.

CORBA-based prototype

For dynamic reconfiguration, the overhead during normal operation is fixed, i.e., there is a fixed extra delay for each invocation. This delay depends on the used ORB and the test environment, and for our test setup (see Chapter 6) results in a 0.13ms delay. For a worst-case scenario this causes a relative increase of 12.4%, but typically this relative increase will be smaller, and we expect that it will be acceptable for most applications.

Overhead for dynamic reconfiguration

The impact on execution caused by a reconfiguration can be quantified as the increase in response time from the perspective of the unaffected components. This increase in response time depends on application characteristics and especially on the time it takes to finish the ongoing invocations.

The performance overhead for our Load Distribution Service consists of three categories: the overhead for the creation of components, the overhead caused by Portable Interceptors, and the overhead caused by the load monitoring. The overhead for the creation of components, and thus for the initial placement load distribution method, is minimal. The overhead caused by the Portable Interceptors and monitoring can be considerable, depending on the application and the chosen frequency for collecting load information. Measurements show that because the overhead associated with the migration load distribution method (which relies on the Dynamic Reconfiguration Service) can be considerable, this distribution method

Overhead for load distribution

should only be used for long-lived components that have short-lasting invocations.

7.5 Future Research

We suggest here some possibilities to continue the research presented in this thesis.

Quantitative modeling

Quantitative modeling of middleware-based applications, and of the middleware itself, would allow better predictions on the QoS (given a certain application and a certain environment), and more optimal adaptations in resource allocations for dynamic QoS mechanisms. Research on how to make quantitative models of middleware and middleware-based application is needed for this.

Platform independent QoS

The *Model Driven Architecture* [MDA01] advocates designing distributed systems in a platform independent manner, and then with the help of development tools transform such a platform independent design into a platform dependent design that can then be implemented. This design approach should in our opinion also consider the QoS aspects of a distributed system. QoS thus has to be expressed in platform independent manner, including the adaptations that might be required to the design to obtain this QoS, and this then has to be translated into platform dependent concepts. As an example, a component might need a 99% uptime, which is a platform independent QoS concept. When such a component is implemented in CORBA, this can be translated to parameters for the Fault-Tolerant CORBA specification, such as the number of replicas and whether to use active or passive replication. Related to this is the ongoing work for a syntax and semantics for QoS formalisms, e.g., the UML QoS Profile RFP [UMLQoS].

Fault tolerance

Mapping QoS mechanism

An additional middleware-layer-internal QoS mechanism that could be added to improve availability is a *fault tolerance mechanism* based on redundancy. Distributed systems with very high availability requirements need such a mechanism, in which replicas of components are created to cope with, for example, network or node failures. In addition, it would be interesting to also add *mapping QoS mechanisms*, which enforce QoS by relying on resource-layer QoS mechanisms.

Feature interaction

Feature interaction between QoS mechanisms is an area of research that is not addressed in this thesis. For example, mechanisms that improve performance might degrade availability, or the instrumentation might not mix. In case of feature interaction issues, a QoS manager could be made responsible for dealing with this. Research in this area would require the availability of a wider range of QoS mechanisms than is provided by our prototype.

Extend load distribution with loose consistency

The load distribution QoS mechanism can be extended with possibilities for *replication with some form of loose consistency*, thereby extending the range of components for which the replication distribution method is suitable.

Extend dynamic reconfiguration

By extending our dynamic reconfiguration QoS mechanism with the possibility to abort ongoing interactions, it is possible to reduce the impact on execution. In addition, it can be investigated to what extend *architectural description formalisms* can aid a reconfiguration designer in preserving correctness.

Samenvatting (Dutch)

Het beheersen van de kwaliteitskarakteristieken (‘Quality of Service’, ‘QoS’) van gedistribueerde systemen is essentieel voor het succes van deze systemen. De QoS-karakteristieken die we in dit proefschrift beschouwen, zijn de prestatiekarakteristieken (namelijk reactietijd en doorvoercapaciteit) en de beschikbaarheidskarakteristieken (namelijk percentage van de tijd dat het systeem operationeel is, gemiddelde tijd tussen storingen en gemiddelde tijd die nodig is voor een reparatie). Het beheersen van de QoS is een complex probleem, omdat het alle functionele lagen betreft die we onderscheiden in dit proefschrift. Dit zijn de applicatie-, ‘middleware’- en ‘resource’-lagen. Het beheersen van de QoS is met name complex voor grootschalige systemen, zoals telematicasystemen, vanwege de heterogeniteit en schaalbaarheidsproblemen.

QoS-mechanismen beheersen een bepaalde QoS-karakteristiek. QoS-mechanismen doen dit door netwerk- of ‘processing’-resources toe te wijzen, of door het gedrag van de applicatie aan te passen. Dit proefschrift richt zich op QoS mechanismen in de middleware-laag.

Bestaande benaderingen om de QoS te beheersen zijn vaak statisch. Statische benaderingen wijzen resources toe op of voor het moment dat een applicatie-instantie gecreëerd wordt. Deze resourcetoewijzing blijft ongewijzigd gedurende de levensduur van de applicatie-instantie. Een statische benadering baseert de resourcetoewijzing op de maximaal benodigde hoeveelheid resources. Om dit te kunnen bepalen, is een gedetailleerde kennis over de applicatie en het gebruik van de applicatie-instantie vereist. Bovendien leidt een statische benadering tot een verspilling van resources, aangezien de daadwerkelijk benodigde hoeveelheid resources varieert gedurende de levensduur van de applicatie-instantie.

Een dynamische benadering varieert de toewijzing van resources gedurende de levensduur van een applicatie-instantie, afhankelijk van de behoefte aan resources op een bepaald moment in de tijd. Bij een dynamische benadering, in tegenstelling tot een statische benadering, is het niet nodig dat de hoeveelheid benodigde resources bepaald wordt voordat de applicatie-instantie gecreëerd wordt. Een dynamische benadering baseert de resource-toewijzing op het 'monitoren' van de huidige QoS van een applicatie-instantie, en het aanpassen van de resource-toewijzing wanneer dat nodig is. Een gevolg hiervan is, dat bij een dynamische benadering de QoS niet gegarandeerd is. Bijvoorbeeld, terwijl de applicatie instantie draait, kan op een bepaald moment het QoS-mechanisme constateren dat er een tekort aan processing resources is. Echter, voor een grote categorie gedistribueerde applicaties, zoals telematica-applicaties, weegt het voordeel van efficiënter gebruik van resources op tegen het nadeel dat er geen harde QoS-garanties mogelijk zijn. In dit proefschrift richten we ons alleen op dynamische QoS mechanismen.

Grootschalige, gedistribueerde systemen worden vaak ontwikkeld met behulp van component-middleware technologieën (bijvoorbeeld CORBA), vanwege de distributie-transparanties die deze technologieën bieden. Deze distributie-transparanties verbergen de complexiteit van het ontwikkelen van gedistribueerde systemen voor de applicatie-ontwikkelaar door de distributie-aspecten in de middleware-laag af te handelen. We breiden dit concept van distributie-transparantie uit door ook QoS-mechanismen toe te voegen aan de middleware-laag en daardoor de complexiteit van het beheersen van de QoS te verbergen voor de applicatieontwikkelaar. Bovendien, maken we het makkelijker deze mechanismen te hergebruiken, door de QoS-mechanismen in de middleware-laag af te handelen.

Een belangrijk aspect van onze benadering is dat we niks veronderstellen over de resource-laag. Onze QoS-mechanismen bevinden zich in de middleware-laag en zijn niet afhankelijk van QoS-functionaliteit in de resource-laag, zoals bijvoorbeeld IntServ, DiffServ of 'real-time' besturingssystemen. Onze QoS-mechanismen gebruiken alleen de functionaliteit van de middleware. Bijvoorbeeld, onze QoS mechanismen kunnen de middleware gebruiken om dynamisch de allocatie van componenten over de verschillende computersystemen te veranderen. De belangrijkste uitdagingen voor het realiseren van deze mechanismen zijn het behouden van de correctheid van de applicaties, en de beperkingen voor het applicatie-ontwerp te minimaliseren.

We hebben onze benadering voor dynamische QoS-mechanismen in de middleware-laag toegepast op twee QoS-mechanismen: een dynamische herconfiguratie mechanisme, en een 'load'-distributie mechanisme.

Het dynamische herconfiguratie mechanisme kan applicaties die ontwikkeld zijn met behulp van component middleware 'upgraden' terwijl ze draaien. Dit mechanisme kan een component vervangen door een nieuwere versie, een component migreren naar een ander computersysteem, een component toevoegen of een component verwijderen, zonder dat de applicatie-instantie gestopt hoeft te worden.

Aangezien dit ervoor zorgt dat de applicatie minder vaak gestopt hoeft te worden, verhoogt dit de beschikbaarheid. Een belangrijk aspect van ons dynamische herconfiguratie mechanisme is dat het de correctheid van de applicatie behouden blijft. Hiervoor brengt ons dynamische herconfiguratie mechanisme een component in een toestand waarin er geen actieve 'invocaties' zijn en de component veilig geherconfigureerd kan worden. Dit gebeurt door selectief de binnenkomende invocaties in een wachtrij te zetten.

Het load-distributie QoS-mechanisme distribueert de component op een zodanige manier over de beschikbare computersystemen dat er aan de prestatie-eisen wordt voldaan. Een zogenaamde load-distributie-strategie maakt de load-distributie-beslissingen, gebaseerd op de beschikbare load-informatie. Het uitvoeren van deze load-distributie-beslissingen wordt gedaan door de volgende load-distributie-methoden: initiële plaatsing, migratie en replicatie van componenten. De optimale load-distributie is afhankelijk van specifieke karakteristieken van de applicatie en de omgeving waarin de applicatie draait. Daarom stellen we in dit proefschrift een oplossing voor waarbij het makkelijk is nieuwe load-distributie-strategieën en nieuwe types load-informatie toe te voegen. Een belangrijk aspect van ons load-distributie mechanisme is dat het naast het evenredig verdelen van load ook QoS-differentiatie toestaat. In het geval van QoS-differentiatie verdelen we de beschikbare computersystemen in klassen, en plaatsen, migreren en repliceren componenten op een zodanige manier over deze klassen dat er aan hun prestatie-eisen wordt voldaan.

Voor beide QoS-mechanismen gebruiken we 'message'-reflectietechnieken om een duidelijke scheiding te bewerkstelligen tussen applicatiecode, QoS-mechanismecode en de middleware-code. Het gebruik van message-reflectie verbetert de transparantie en de mogelijkheden om ons QoS-mechanisme samen te stellen.

Een op CORBA gebaseerd prototype implementeert onze Dynamic Reconfiguration Service and onze Load Distribution Service. Dit prototype toont aan dat onze benadering voor QoS-mechanismen in de middleware-laag, en onze dynamische reconfiguratie- en load-distributie QoS-mechanismen, uitvoerbaar zijn. We gebruiken CORBA Portable Interceptors om de message reflectie te implementeren. Metingen met het prototype geven inzicht in de prestatie aspecten, en tonen aan dat de 'overhead' acceptabel is voor de meeste applicaties.

Het onderzoek dat in dit proefschrift beschreven is, kan gebruikt worden voor commercieel toepasbare QoS-mechanismen, die de QoS verbeteren van applicaties die ontwikkeld zijn met component-middleware technologieën.

References

[Almeida01A]	J. P. A. Almeida, M. Wegdam, L. Ferreira Pires, M. van Sinderen, “An approach to dynamic reconfiguration of distributed systems based on object-middleware”, in *Proceedings of the 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001.Also appeared as CTIT Technical Report TR-CTIT-01-06 in February 2001.
[Almeida01B]	J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis, “Transparent Dynamic Reconfiguration for CORBA”, in *Proceedings of the 3rd International Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, September 2001.
[Almeida01C]	J. P. A. Almeida, “Dynamic Reconfiguration of Object-Middleware-based Distributed Systems”, *MSc Thesis*, Faculty of Computer Science, University of Twente, 2001. Supervisors: M.Wegdam, M.J. van Sinderen, L.J.M. Nieuwenhuis and L. Ferreira Pires. nr. UT-CS-ARCH-2001-03, http://arch.cs.utwente.nl/assignments/thesis/ARCH-2001-03.pdf. Also appeared as deliverable AMIDST/WP1/N021 of the AMIDST project.
[Andersen01]	Anders Andersen, Gordon Blair, Vera Goebel, Randi Karlsen, Tage Stabell-Kulø, Weihai Yu, “Arctic Beans: Configurable and Re-configurable Enterprise Component Architectures”, *Work in Progress session at Middleware 2001*, Heidelberg, Germany, November 2001. Published in IEEE Distributed Systems Online, Vol. 2, No. 7, 2001.
[Andersen02]	Anders Andersen, “OOPP - A Reflective Middleware Platform including Quality of Service Management”, *PhD Thesis*, University of Tromso, Norway, 2002.
[Aurrecoech98]	C. Aurrecoechea, A.T. Campbell, L. Hauw, “A Survey of QoS Architectures”, *Multimedia Systems Journal, Special Issue on QoS Architecture*, ACM/Springer Verlag, Vol. 6, No. 3, pg. 138-151, May 1998.
[Avizienis00]	A.Avizienis, J.C.Laprie, B.Randell, “Fundamental concepts of dependability”, *Third Information Survivability Workshop (ISW 2000)*, October 24-26, 2000, Boston, Massachusetts, USA
[Badidi99]	E. Badidi, R. K. Keller, P. G. Kropf, V. V. Dongen, “The Design of a Trader-Based CORBA Load Sharing Service”, *Proc. of the 12th Int’l Conf. On Parallel and Distributed Computing Systems (PDCS’99)*, August 1999.

[Bakken01] David E. Bakken, “Middleware”, in *Encyclopedia of Distributed Computing*, Kluwer Academic Press, 2001.

[Barth99] Thomas Barth, Gerd Flender, Bernd Freisleben, Frank Thilo, “Load Distribution in a CORBA Environment”, *Int’l Symposium on Distributed Objects and Applications (DOA 99)*, pp. 158-166, Edinburgh, UK, 1999.

[Bergmans96] Lodewijk Bergmans, Mehmet Aksit, “Composing Synchronization and Real-Time Constraints”, *Journal of parallel and distributed computing*, 1996, Vol. 36, pp. 32-52, Academic Press.

[Bergmans00] L. Bergmans, A. van Halteren, L. Ferreira Pires, M. van Sinderen, M. Aksit, “A QoS-Control Architecture for Object Middleware”, in *Proceedings of the 7th Intl. Conf. on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2000)*, Enschede, Netherlands, October 2000. LNCS 1905, Springer, 2000, pp. 117-131.

[Bergmans01] Lodewijk Bergmans, Mehmet Aksit, “Composing crosscutting concerns using composition filters”, *Communications of the ACM*, October 2001, Vol. 44, No.10, pp. 51-57.

[Berman96] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, Gary Shao, “Application-Level Scheduling on Distributed Heterogeneous Networks”, in *Proceedings of Supercomputing 1996*. Also published as UCSD CS Tech Report #CS96-482.

[Bidan98] C. Bidan, V. Issarny, T. Saridakis, A. Zarras, “A dynamic reconfiguration service for CORBA”, in *Proc. IEEE International Conference on Configurable Distributed Systems*, May 1998.

[Blair98A] Gordon S. Blair, Geoff Coulson, “The case for reflective middleware”, *internal report*, MPG-98-38, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1998.

[Blair98B] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, “An Architecture for Next Generation Middleware”, *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, UK.

[Bloom93] T. Bloom, M. Day, “Reconfiguration and module replacement in Argus: Theory and Practice”, *IEE Software Engineering Journal*, Vol 8, No 2, March 1993.

[Boyd02] Tom Boyd, Partha Dasgupta, “Process Migration: A Generalized Approach using a Virtualizing Operating System”, in *Proc. 22nd International Conference on Distributed Computing Systems (ICDCS-2002)*, Vienna, July 2002.

[Cardellini02] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, Philip S. Yu, “The state of the art in locally distributed Web-server systems”, *ACM Computing Surveys (CSUR)*, Vol. 34, Nr. 2, June 2002.

[Carter99] C. Carter-Schwendler, “Component Load Balancing with COM+”, *COM+ Resource CD*, 1999, http://www.microsoft.com/com/resources/compluscd/slides/LoadBalancing.zip (Current as of 5 December 2002).

[Casavant88] T.L. Casavant, J.G. Kuhl, “Effects of Response and Stability on Scheduling in Distributed Computing Systems”, *IEEE Transactions on Software Engineering*, Vol. 14, Nr. 11, Nov. 1988, pp. 1578-1588.

[CCM]	Object Management Group, "CORBA Component Model", version 3.0, formal/2002-06-65, June 2002.
[Chapin96]	Steve J. Chapin, "Distributed and multiprocessor scheduling", *ACM Computing Surveys*, Vol. 28, Nr. 1, 1996.
[Chockler00]	G. Chockler, D. Dolev, R. Friedman, R. Vitenberg, "Implementing a Caching Service for Distributed Objects", in *Proc. IFIP/ACM Int'l Conf. on Distr. Systems Platforms and Open Distr. Processing (Middleware 2000)*, April 2000.
[COM+]	M. Kirtland, "Object-Oriented Software Development Made Simple with COM+ Runtime Services", *Microsoft Systems Journal*, Nov. 1997, http://www.microsoft.com/msj/1197/complus.htm.
[CORBA]	Object Management Group, "The Common Object Request Broker: Architecture and specification", Rev. 2.4.1, formal/00-11-07, Nov. 2000.
[CORBALB]	Object Management Group, "Load Balancing Request For Proposal", http://www.omg.org/techprocess/meetings/schedule/Load_Balancing_RFP.html, April 2002.
[CORBANot]	Object Management Group, "Notification Service Specification", Rev. 1.0, formal/00-06-20, June 2000.
[CORBANS]	Object Management Group, "Naming Service Specification", Rev. 1.1.1, formal/01-02-65, February 2001.
[CORBAPSS]	Object Management Group, "Persistent State Service", Rev. 2.0, formal/2002-09-06, 2002.
[CORBATime]	Object Management Group, "Time Service Specification", Rev. 1.0, formal/00-06-26, May 2000.
[CORBATS]	Object Management Group, "Trading Object Service Specification", Rev. 1.0, formal/00-06-27, May 2000.
[Dahlin00]	Michael Dahlin, "Interpreting Stale Load Information", *IEEE Transactions on Parallel and Distribution Systems*, Vol. 11, No. 10, Oct. 2000.
[DCOM]	Microsoft Corporation and Digital Equipment Corporation, "The Component Object Model Specification", Version 0.9, Oct. 1995, http://www.microsoft.com/com/resources/comdocs.asp.
[DiffServ98]	S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An Architecture for Differentiated Services", IETF RFC 2475, December 1998
[Eddon99]	Guy Eddon, "COM+: The Evolution of Component Services", *IEEE Computer*, Vol. 32, No. 7, pp. 104-106. 1999.
[EJB]	Sun Microsystems, "Enterprise JavaBeans Specification", Version 2.0, 22 August 2001.
[Eliassen02]	Frank Eliassen, Thomas Plagemann, Brita Hafskjold, Tom Kristensen, Hans Ole Rafaelsen, Robert H Macdonald, "QoS management in the MULTE-ORB", *IEEE Distributed System Online*, http://dsonline.computer.org/middleware/articles/dsonline-MULTE-ORB.html, March 2002.
[Elrad01A]	Tzilla Elrad, Robert E. Filman, Atef Bader, "Aspect-oriented Programming: Introduction", *Communications of the ACM*, October 2001, Vol. 44, No. 10, pp. 29-32.

[Elrad01B]	Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, Harold Ossher, "Discussing aspects of AOP", *Communications of the ACM*, Oct. 2001, Vol. 44, No.10, pp. 33 – 38.
[Emmerich02]	Wolfgang Emmerich, "Distributed Component Technologies and their Software Engineering Implications", in *Proc. of the 24th International Conference on Software Engineering*, Orlando, Florida, pp. 537-546, 2002.
[Endler94]	M. Endler, "A language for implementing generic dynamic reconfigurations of distributed programs", in *Proc. of the 12th Brazilian Symposium on Computer Networks*, 1994.
[Ferber89]	J.Ferber, "Computational Reflection in Class based Object Oriented Languages", in *Proc. OOPSLA 1989*, SIGPLAN Notices, pp. 317-326, 1989.
[Filman02]	Robert E. Filman, Stuart Barrett, Diana D. Lee, Ted Linden, "Inserting ilities by controlling communications", *Communications of the ACM*, Vol. 45, No. 1, 2002.
[Foster00]	I. Foster, A. Roy, V. Sander, "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation", *8th International Workshop on Quality of Service*, Pittsburgh, USA, June 2000.
[Franken96]	Leonard Franken, "Quality of Service Management: a Model-Based Approach", *CTIT Ph.D.-thesis series*, ISSN-1381-3617, No. 96-10, ISBN 90-72125-56-8, 1996.
[Frolund99]	S.Frolund, J.Koistinen, "Quality of Service Aware Distributed Object Systems", in *Proc. of the 1999 USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1999. Also appeared as HP Labs Technical Report HPL 98-142.
[FTCORBA]	Object Management Group, "Fault Tolerant CORBA Specification", Rev. 1.0, ptc/2000-04-04, April 2000.
[Gamma94]	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", published by Addison Wesley, ISBN 0-201-63361-2, 1994.
[Geihs01A]	K. Geihs, "Middleware challenges ahead", *Computer*, June 2001, Vol. 34, Nr. 6, pp. 24 – 31.
[Geihs01B]	K. Geihs, C. Becker, "A Framework for Re-use and Maintenance of Quality of Service Mechanisms in Distributed Object Systems", in *Proc. of International Conference on Software Maintenance (ICSM 2001)*, Italy, 2001.
[Goudarzi99]	K. Moazami-Goudarzi, "Consistency preserving dynamic reconfiguration of distributed systems", Ph.D. thesis, Imperial College, London, March 1999.
[Halteren01]	A.T. van Halteren, G. Fábián, E. Groeneveld, "Design and evaluation of a QoS provisioning service", in *Proceedings of 3rd Intl. Conf. on Distributed Applications and Interoperable Systems (DAIS 01)*, Krakow, Poland, September 2001.
[Halteren03]	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.

[Halteren99A] Aart T. van Halteren, Lambert J.M. Nieuwenhuis, Mike R. Schenk, Maarten Wegdam, “Value Added Web: Integrating WWW with a TINA Service Management platform”, in *Proc. of Telecommunications Information Networking Architecture Conference 1999 (TINA '99)*, Apr. 1999.

[Halteren99B] A.T. van Halteren, A. Noutash , L.J.M. Nieuwenhuis, M. Wegdam, “Extending CORBA with specialised protocols for QoS provisioning”, in *Proc. of International Symposium on Distributed Objects and Applications (DOA'99)*, Sept. 1999.

[Hellenthal01] J.W. Hellenthal, F.J.M. Panken, M.Wegdam, “Validation of the Parlay API through prototyping”, in *Proc. IEEE Intelligent Network Workshop 2001 (IN2001)*, 6-8 May, Boston, USA.

[Henning98] M. Henning, “Binding, migration, and scalability in CORBA”, *Communications of the ACM*, Vol. 41, No. 10, Oct. 1998.

[Henning99] M. Henning, S. Vinoski, “Advanced CORBA Programming with C++”, 3rd printing, Addison-Wesley, Reading, MA., 1999.

[Hofmeister93] C. Hofmeister, “Dynamic Reconfiguration of Distributed Applications”, *PhD thesis*, 1993, University of Maryland, USA.

[IEEE610] IEEE, “Standard Glossary of Software Engineering Terminology”, IEEE Std. 610.12-1990, 1990.

[IntServ94] R. Braden, D. Clark, S. Shenker, “Integrated Services in the Internet Architecture: An Overview”, IETF RFC 1633, July 1994.

[ISO-QoS] ITU/ISO, “Quality of Service – Framework”, ISO/IEC CD 13236, 1998.

[Janowiak03] R.M. Janowiak, “Computers and communications: a symbiotic relationship”, *IEEE Computer*, Vol. 36, No. 1, Jan 2003, pp. 76-79.

[Jansen01] M. Jansen, E. Klaver, P. Verkaik, M. van Steen, A.S. Tanenbaum, “Encapsulating Distribution in Remote Objects”, *Information and Software Technology*, Vol. 43, No. 6, pp. 353-363, May 2001.

[Java] Sun, Java 2 Platform, http://java.sun.com.

[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.

[Kiczales97] G.Kiczales, J. Lamping, A.Mendhekar, C. Lopes, J. Loingtier, J. Irwin, “Aspect-Oriented Programming”, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, June 1997.

[Kramer85] J. Kramer, J. Magee, “Dynamic configuration for distributed systems”, *IEEE Transactions on Software Engineering*, Vol. 11, No. 4, pp. 424-436, April 1985.

[Kramer90] J. Kramer, J. Magee, “The evolving philosophers’ problem: dynamic change management”, *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, pp. 1293-1306, November 1990.

[Kuhns99] Fred Kuhns, Carlos O’Ryan, Douglas C. Schmidt, Ossama Othman, Jeff Parsons, “The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware”, *IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN ’99)*, Aug. 1999, Salem, MA, USA.

[Lagerberg02] Ko Lagerberg, Dirk-Jaap Plas, Maarten Wegdam, "Web Services in 3G Service Platforms", *Bell Labs Technical Journal, Special issue on Wireless Networks*, online ISSN 1538-7305, print ISSN 1089-7089, published by Wiley Periodicals Inc., Vol. 7, No. 2, pp. 167-183, Dec. 2002.

[Leijdekkers97] Peter Leydekkers, "Multimedia Services in Open Distributed Telecommunications Environments", *CTIT Ph.D.-thesis series ISSN-1381-3617*, No. 97-12, ISBN 90-72125-04-6, University of Twente, The Netherlands, 1997.

[Linderm00] M. Lindermeier, "Load Management for Distributed Object-Oriented Environments", in *Proc. 2nd Int'l Symp. Distr. Objects and Appl. (DOA 2000)*, pp. 59-68, IEEE CS Press, Antwerp, Belgium, 2000.

[Liskov88] Barbara Liskov, "Data Abstraction and Hierarchy", *ACM SIGPLAN Notices*, Vol. 23, No. 5, pp. 17-34, May 1988.

[Magee95] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, "Specifying Distributed Software Architectures", in *Proc. of the 5th European Software Engineering Conference*, ESEC '95, Barcelona, 1995.

[Man00A] Ronald de Man, Jeroen Schot, Jack Verhoosel, "Load balancing in a TINA based service deployment environment", *ISS 2000*, Birmingham, UK, 2000.

[Man00B] Ronald de Man, Rudynell Millian, Maarten Wegdam, Aniruddha Gokhale, Shalini Yajnik, "Transparent Fault Tolerance for CORBA based Distributed Components", Poster at *15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, two page extended abstract appeared in the conference companion and ACM Digital Library, Oct. 2000, Minneapolis, Minnesota USA.

[MDA01] OMG Architecture Board MDA Drafting Team, "Model Driven Architecture - A Technical Perspective", editors Joaquin Miller and Jishnu Mukerji, ormsc/01-07-01, July 2001.

[Miller01] Sandra Kay Miller, "Aspect Oriented Programming takes Aim at Software Complexity", *IEEE Computer*, April 2001.

[Milojicic00] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, "Process migration", *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241-299, Sept. 2000.

[Molenkamp01] Gary Molenkamp, Michael Katchabaw, Hanan Lutfiyya, Michael Bauer, "Distributed Resource Management to Support Distributed Application-Specific Quality of Service", *4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MMNS) 2001*, LNCS 2216, pp. 142-159, 2001.

[Molenkamp02] G. Molenkamp, H. Lutfiyya, M. Katchabaw, M. Bauer, "Diagnosing quality of service faults in distributed applications", in *Proc. of 21st IEEE International Conference on Performance, Computing, and Communications 2002*, pp. 375 - 382, April 2002, Phoenix, AZ, USA.

[Moser00] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, V. Kalogeraki, "Eternal: Fault Tolerance and Live Upgrades for Distributed Object Systems", in *Proc. of the IEEE Information Survivability Conference*, Hilton Head, SC, Jan. 2000.

[Nahrstedt01]	K. Nahrstedt, X Dongyan, D. Wichadakul, L. Baochun, "QoS-aware middleware for ubiquitous and heterogeneous environments", *IEEE Communications Magazine*, Vol. 39, No. 11, Nov. 2001, pp. 140 –148.
[Narasimh99A]	Priya Narasimhan, Louise E. Moser, P.M. Melliar-Smith, "Using Interceptors to Enhance CORBA", *IEEE Computer*, Vol. 32, No. 7, pp. 62-68, 1999.
[Narasimh99B]	P. Narasimhan, "Transparent Fault Tolerance for CORBA", *Ph.D. thesis*, University of California, Santa Barbara, Dec. 1999.
[Nasika00]	R. Nasika, P. Dasgupta, "Transparent Migration of Distributed Communicating Processes", *13th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS-2000)*, Aug. 2000
[Natarajan00]	Balachandran Natarajan, Aniruddha Gokhale, Shalini Yajnik, Douglas C. Schmidt, "DOORS: Towards High-performance Fault Tolerant CORBA", in *Proc. 2nd Int'l Symp. Distr. Objects and Appl. (DOA 2000)*, IEEE CS Press, Antwerp, Belgium, 2000.
[NetRemoting]	Piet Obermeyer, Jonathan Hawkins, "Microsoft .NET Remoting: A Technical Overview", Microsoft Corporation, July 2001, http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp.
[Neuman94]	B. Neuman, "Scale in Distributed Systems. Readings in Distributed Computing Systems", pp. 463–489. IEEE CS Press, Los Alamitos, CA., 1994.
[Nieuwenh91]	L.J.M. Nieuwenhuis, "Fault Tolerance through Program Transformation", PhD Thesis, University of Twente, 1991.
[ORBacus]	IONA, "ORBacus CORBA ORB", http://www.iona.com.
[Orbix]	Iona Technologies, "Orbix 2000 1.2.1 Administrator's Guide", http://www.iona.com.
[Oreizy98]	P. Oreizy, N. Medvidovic, R. Taylor, "Architecture-based runtime software evolution", in *Proc. of the International Conference on Software Engineering*, April 1998.
[Othman01A]	O. Othman, C. O'Ryan, D. C. Schmidt, "Strategies for CORBA Middleware-Based Load Balancing", *IEEE Distributed Systems Online*, Vol. 2, Nr 3, March 2001.
[Othman01B]	O. Othman, C. O'Ryan, D. C. Schmidt, "Designing an Adaptive CORBA Load Balancing Service Using TAO", *IEEE Distributed Systems Online*, Vol. 2, Nr. 4, April 2001.
[Othman01C]	O. Othman, D. C. Schmidt, "Optimizing Distributed System Performance via Adaptive Middleware Load Balancing", in *Proc. ACM SIGPLAN Workshop on Optimization of Middleware and Distr. Systems*, Snowbird, Utah, June 2001.
[P715]	Eurescom P715, "Middleware for the Open Services Market, A boolet for Executives", Eurescom P715 project (British Telecom, Deutsche Telekom, France Telecom, FINNET Group, KPN, Telecom Eireann), http://www.eurescom.de, 1999.
[Pierre01]	G. Pierre, M. van Steen, "Globule: a Platform for Self-Replicating Web Documents", in *Proc. 6th Int'l Conf. on Protocols for Multimedia Systems*, Enschede, The Netherlands, October 2001.

[Pierre02] G. Pierre, M. van Steen, A.S. Tanenbaum. "Dynamically Selecting Optimal Distribution Strategies for Web Documents", *IEEE Transactions on Computers*, Vol. 51, No. 6, June 2002.

[Plas99] D.J. Plas, “Using Message Reflection for the Management of CORBA”, *MSc thesis*, University of Groningen, Supervisors: M. Wegdam. A.T. van Halteren, L.J.M. Nieuwenhuis, Dec. 1999.

[Post02] E. Post, “Load Distribution in Object Middleware”, *MSc thesis*, Faculty of Computer Science, University of Twente, 2002. Supervisors: M.Wegdam, M.J. van Sinderen, L.J.M. Nieuwenhuis, N. Diakov, nr. UT-CS-ARCH-2002-01, http://arch.cs.utwente.nl/assignments/thesis/ARCH-2002-01.pdf.

[Putman01] J. Putman, “Architecting with RM-ODP”, ISBN 0-13-019116-7, Prentice Hall, 2001.

[Rackl01] G. Rackl, “Monitoring and Managing Heterogeneous Middleware”, *PhD thesis*, Institut für Informatik, Technische Universität München, 2001.

[Raymond95] Kerry Raymond, “Reference Model of Open Distributed Processing (RM-ODP): Introduction”, *IFIP International Conference on Open Distributed Processing (ICODP '95)*, Brisbane, Australia, Febr. 1995.

[Raynal96] M. Raynal, M. Singhal, "Logical Time: Capturing Causality in Distributed Systems", *IEEE Computer*, Vol. 29, No. 2, pp. 49-57, Febr. 1996.

[Ren03] Yansong (Jennifer) Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, Mouna Ser, “AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects”, *IEEE Trans. on Computers*, Vol. 52, No. 1, pp. 31-50, Jan. 2003.

[RMI] Sun, “Java Remote Method Invocation”, http://java.sun.com/products/jdk/rmi/.

[RMODPPart1] ITU-T / ISO, “Open Distributed Processing Reference Model. Part 1 – Overview”, ITU-T X.901 | ISO/IEC 10746-1, Aug. 1997.

[RMODPPart2] ITU-T / ISO, “Open Distributed Processing - Reference Model - Part 2: Foundations”, ITU-T X.902 | ISO/IEC 10746-2, Nov. 1995.

[RMODPPart3] ITU-T / ISO, “Open Distributed Processing - Reference Model - Part 3: Architecture”, ITU-T X.903 | ISO/IEC 10746-3, Nov. 1995.

[Rodriguez99] Noemi Rodriguez, Roberto Ierusalimschy, “Dynamic Reconfiguration of CORBA-Based Applications”, *SOFSEM'99: Theory and Practice of Informatics, 26th Conference on Current Trends in Theory and Practice of Informatics*, LNCS 1725, Milovy, Czech Republic, Nov./Dec. 1999.

[RTCORBA] Object Management Group, “Real-Time CORBA Specification”, Rev. 1.1, formal/2002-08-02, Aug. 2002.

[Santos01] Luis Paulo Peixoto dos Santos, “Application Level RunTime Load Management: A Bayesian Approach”, *PhD thesis*, Universidade do Minho, Portugal, 2001.

[Santos96] L. P. P. dos Santos, “Load Distribution: a Survey”, *Tech. Report*, UM/DI/TR/96/03, Universidade do Minho, Escola de Engenharia, Departamento de Informática, Oct. 96.

[Schantz02] Richard E. Schantz, Douglas C. Schmidt, “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications”, *Encyclopedia of Software Engineering,* Wiley and Sons, 2002.

[Schmidt98] Douglas C. Schmidt, “Evaluating architectures for multithreaded object request brokers”, *Communications of the ACM*, Vol. 41, No. 10, pp. 54 – 60.

[Schnekenb00] T. Schnekenburger, “Load Balancing in CORBA: A Survey of Concepts, Patterns, and Techniques”, *The Journal of Supercomputing*, Vol. 15, No. 2, 2000, pp. 141-161.

[Schnekenb96] T. Schnekenburger, “Automatic Load Distribution for CORBA Applications”, in *Proc. 1st Component User’s Conf. (CUC 1996)*, SIGS, 1996.

[Schnekenb97] T. Schnekenburger, G. Rackl, “Implementing Dynamic Load Distribution Strategies with Orbix”, in *Proc. Int’l Conf. Parallel and Distr. Processing Techniques and Appl. (PDPTA’97)*, CSREA, 1997, Vol. 2, pp. 996-1005.

[Sellin99] Eric Sellin, Peter Loosemore, Sohail Rana, Jürgen Dittrich, Maarten Wegdam, “Audio/Video Stream Binding in a Pan-European Service Management Platform”, in *Proc. of Sixth International Conference on Intelligence in Services and Networks (IS&N '99)*, Apr. 1999.

[Shivaratri92] N. Shivaratri, P. Krueger, M. Singhal, “Load Distributing for Locally Distributed Systems”. *IEEE Computer*, Vol. 25, No. 12, Dec. 1992, pp. 33-44.

[Siqueira00] Frank Siqueira, Vinny Cahill, “An Open QoS Architecture for CORBA Applications”, in *Proc. of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2000.

[SOAP] World Wide Web Consortium, “Simple Object Access Protocol (SOAP)”, version 1.1, W3C Note 8 May 2000, <http://www.w3.org/TR/SOAP/>.

[Steen98] M. van Steen, A.S. Tanenbaum, I. Kuz, H.J. Sips, “A Scalable Middleware Solution for Advanced Wide-Area Web Services”. in *Proc. Middleware '98*, Sept. 1998.

[Szyperski98] C. Szyperski, “Component software”, Addison-Wesley, 1998.

[Szyperski99] C. Szyperski, “Components and Objects Together”, *Software Development Magazine*, Document ID 11334, May 1999.

[TAO] TAO homepage, <http://www.cs.wustl.edu/~schmidt/TAO.html>.

[Tewksb01A] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, “Automatically Generated State Transfer and Conversion Code to Facilitate Software Upgrades”, *Maintenance and Reliability Conference*, Gatlinsburg, Tn, USA, May 2001.

[Tewksb01B] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, “Coordinating the Simultaneous Upgrade of Multiple CORBA Objects”, *3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, Rome, Italy, Sept. 2001.

[Tewksb01C] L.A. Tewksbury, L.E. Moser, P.M. Melliar-Smith, “Live Upgrade Techniques for CORBA Applications”, *Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 01)*, Krakow, Poland, Sept., 2001.

[Tewksb01D] L.A. Tewksbury, L.E. Moser, P.M. Melliar-Smith, “Live upgrades of CORBA applications using object replication”, in *Proc. IEEE International Conference on Software Maintenance*, Nov. 2001. Florence, Italy, pp. 488-497.

[Thissen00] D. Thissen, H. Neukirchen, “Managing Services in Distributed Systems by Integrating Trading and Load Balancing”, in *Proc. 5th IEEE Sym on Computers and Communications (ISCC 2000)*, IEEE CS Press, Los Alamitos, Calif., 2000.

[UMLQoS] Object Management Group, “Request for Proposal - UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms”, ad/2002-01-07.

[Vanegas98] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, David E. Bakken, “QuO's Runtime Support for Quality of Service in Distributed Objects”, in *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Sept. 1998.

[Visser00] C.A. Visser, L. Ferreira Pires, D.A.C. Quartel, M.J. van Sinderen, “The Architectural Design of Distributed Systems”, *Reader for The Design of Telematics Systems course*, University of Twente, The Netherlands, Nov. 2000.

[Waldo94] Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall, “A Note on Distributed Computing”, Sun Labs SMLI TR-94-29, Nov. 1994.

[Wang00] Nanbor Wang, Kirthika Parameswaran, Douglas C. Schmidt, “The Design and Performance of Meta-Programming Mechanisms for ORB Middleware”, in *Proc. of the 6th Conference on Object-Oriented Technologies and Systems*, San Antonio, TX, pp. 103-118, USENIX, Jan./Feb. 2000.

[Weblogic] BEA, “Using WebLogic Server Clusters”, BEA Weblogic Server 6.1, <http://e-docs.bea.com/wls/docs61/cluster/>, 24 June 2002.

[Wegdam00A] M.Wegdam, A.T. van Halteren, “Experiences with CORBA interceptors”, *Position paper for the Workshop on Reflective Middleware (RM 2000)*, co-located with the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), April 2000.

[Wegdam00B] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, “ORB Instrumentation for Management of CORBA”, *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Las Vegas, USA, June 2000.

[Wegdam00C] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, “Using message reflection in a management architecture for CORBA”, *11th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2000)*, LNCS 1960, Dec. 2000, Austin, Texas, USA.

[Wegdam01A] M. Wegdam, J. P. A. Almeida, “Lucent response to OMG ORBOS RFI on online updates”, orbos/01-01-01, Jan. 2001.

[Wegdam01B] Maarten Wegdam, Dirk-Jaap Plas, Musa Unmehopa, “Validation of the Open Service Access API for UMTS Application Provisioning”. in *Proc. of the 6th International Conference on Protocols for Multimedia Systems (PROMS 2001)*, LNCS 2213, pp. 210-221, Oct. 2001, Enschede, The Netherlands.

[Wegdam03A] Maarten Wegdam, João Paulo A. Almeida, Marten J. van Sinderen, Lambert J.M. Nieuwenhuis, “Dynamic Reconfiguration for Middleware-Based Applications”, *CTIT Technical Report*, TR-CTIT-03-09, ISSN 1381-3625, March 2003. Also submitted to IEEE Transactions on Parallel and Distributed Systems - Special Issue on Middleware (Fall 2003).

[Wermel99] M. A. Wermelinger, “Specification of software architecture reconfiguration”, *PhD thesis*, Universidade Nova de Lisboa, September 1999.

[WSDL] World Wide Web Consortium, "Web Service Description Language (WSDL)", version 1.1, W3C Note, 15 March 2001, <http://www.w3.org/TR/wsdl>.

[XML] World Wide Web Consortium, "Extensible Markup Language (XML)", version 1.0 (Second Edition), W3C Recommendation, 6 October 2000, <http://www.w3.org/TR/REC-xml>.

[Yokote92] Yasuhiko Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation", *OOPSLA'92 Proceedings*, October 1992.

[Zinky97] J.A. Zinky, D.E. Bakken, R.E. Schantz, "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*. April 1997.

Publications by the Author

In reverse chronological order:

- Maarten Wegdam, João Paulo A. Almeida, Marten J. van Sinderen, Lambert J.M. Nieuwenhuis, “Dynamic Reconfiguration for Middleware-Based Applications”, *CTIT Technical Report* TR-CTIT-03-09, ISSN 1381-3625, March 2003. Also submitted to *IEEE Transactions on Parallel and Distributed Systems* - Special Issue on Middleware (Fall 2003).

- Ko Lagerberg, Dirk-Jaap Plas, Maarten Wegdam. “Web Services in 3G Service Platforms”, *Bell Labs Technical Journal* - Special issue on Wireless Networks, online ISSN: 1538-7305, print ISSN: 1089-7089, Published by Wiley Periodicals Inc., 7(2), pp. 167-183, December 2002.

- Maarten Wegdam, Dirk-Jaap Plas, Musa Unmehopa, “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, Enschede, The Netherlands, October 2001.

- J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis, “Transparent Dynamic Reconfiguration for CORBA”, In *Proceedings of the 3rd International Symposium on Distributed Objects & Applications (DOA 2001)*, ISBN 0-7695-1300-X, pp. 197-207, Rome, Italy, September 2001.

- M. Wegdam, J. P. A. Almeida. “Lucent response to OMG ORBOS RFI on online updates”, orbos/01-01-01, January 2001.

- J.W. Hellenthal, F.J.M. Panken, M.Wegdam, “Validation of the Parlay API through prototyping”, *IEEE Intelligent Network Workshop 2001 (IN2001)*, ISBN 0-7803-7047-3, pp. 58-63, 6-8 May, Boston, USA.

- J. P. A. Almeida, M. Wegdam, L. Ferreira Pires, M. van Sinderen, “An approach to dynamic reconfiguration of distributed systems based on object-middleware”, In *Proceedings of the19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Santa Catarina, Brazil, May 2001. Also appeared as CTIT Technical Report TR-CTIT-01-06, ISSN 1381-3625, February 2001.

- Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, “Using message reflection in a management architecture for CORBA”, *11th IFIP/IEEE International Workshop on Distributed Systems: Operations &Management (DSOM 2000)*, Springer-Verlag Lecture Notes in Computer Science Volume 1960, pages 230-242, Austin, Texas, USA, December 2000.

- O. Kath, A. v. Halteren, F. Stoinski, M. Wegdam, Mike Fisher, “Integrated Middleware Platform Management based on Portable Interceptors”, *11th IFIP/IEEE International Workshop on Distributed Systems: Operations &Management (DSOM 2000)*, Springer-Verlag Lecture Notes in Computer Science Volume 1960, pages 107-118, Austin, Texas, USA, December 2000.

- Ronald de Man, Rudynell Millian, Maarten Wegdam, Aniruddha Gokhale, Shalini Yajnik, “Transparent Fault Tolerance for CORBA based Distributed Components”, Poster at *15th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, two page extended abstract appeared in the conference companion and ACM Digital Library, ISBN 1-58113-307-3, Minneapolis, Minnesota, USA, October 2000.

- Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, “ORB Instrumentation for Management of CORBA”, *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, Las Vegas, USA, June 2000.

- M.Wegdam, A.T. van Halteren, “Experiences with CORBA interceptors”, Position paper for the *Workshop on Reflective Middleware (RM 2000)*, co-located with the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), April 2000.

- A.T. van Halteren, A. Noutash, L.J.M. Nieuwenhuis, M. Wegdam, “Extending CORBA with specialised protocols for QoS provisioning”, In *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, ISBN 0-7695-0182-6, pages 318-327, September 1999.

- Eric Sellin, Peter Loosemore, Sohail Rana, Jürgen Dittrich and Maarten Wegdam, “Audio/Video Stream Binding in a Pan-European Service Management Platform”, In *Proceedings of Sixth International Conference on Intelligence in Services and Networks (IS&N '99)*, Springer-Verlag Lecture Notes in Computer Science Volume 1597, pp. 357-372, April 1999.
- Aart T. van Halteren, Lambert J.M. Nieuwenhuis, Mike R. Schenk and Maarten Wegdam, “Value Added Web: Integrating WWW with a TINA Service Management platform”, In *Proceedings of Telecommunications Information Networking Architecture Conference 1999 (TINA '99)*, ISBN 0-7803-5785-X, pages 14-23, April 1999.

About the author

Maarten Wegdam holds a M.Sc. degree (December 1996) in Computer Science from the University of Groningen, The Netherlands.

He worked as an applied scientist for KPN Research for three years (1997-1999), and since then is a member of technical staff at the Bell Labs Advanced Technologies EMEA Twente department, Lucent Technologies in The Netherlands. He specializes in middleware, Quality of Service and service platforms. He has worked as an architect and project manager in several internal and collaborative projects in these areas.

For his Ph.D. research, Mr. Wegdam was a visitor of the Architecture of Distributed Systems group, which is part of the Department of Computer Science at the University of Twente, The Netherlands. The research of the Department of Computer Science is embedded in the Centre for Telematics and Information Technology.

Dynamic Reconfiguration and Load Distribution in Component Middleware

Maarten Wegdam

Large-scale distributed systems, such as telematics systems, are often built using component-middleware technologies (e.g., CORBA). Middleware offers distribution transparencies. This means that complexities related to the distribution are hidden from the application developers by embedding the distribution aspects in the middleware. Component middleware is middleware that uses component concepts such as encapsulation and well-defined interfaces.

This thesis proposes a new approach for the design of Quality of Service (QoS) mechanisms in component middleware. The specific QoS mechanisms that we propose in this thesis are (i) a new dynamic reconfiguration mechanism, which improves the availability by allowing online replacements and migrations of application components, and (ii) a new load distribution mechanism, which improves the performance of application components. Important characteristics of these QoS mechanisms are: (i) they are dynamic, (ii) they do not rely on specific network or operating system QoS functionality, and (iii) they are transparent for the developer of application components. We achieve transparency by using message reflection techniques in the middleware layer.

A CORBA-based prototype serves as a proof of concept for our approach and our QoS mechanisms.

ISBN 90-75176-36-8

Lucent Technologies
Bell Labs Innovations

Centre for Telematics and Information Technology

Telematica Instituut