Towards an adaptable QoS aware middleware for distributed objects

Aart T. van Halteren

Enschede, The Netherlands, 2003

CTIT PhD.-thesis series number 02-46
Telematica Instituut Fundamental Research Series, No. 008 (TI/FRS/008)

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

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

001 G. Henri ter Hofte, *Working apart together : Foundations for component groupware*
002 Peter J.H. Hinssen, *What difference does it make? The use of groupware in small groups*
003 Daan D. Velthausz, *Cost-effective network-based multimedia information retrieval*
004 Lidwien A.M.L. van de Wijngaert, *Matching media: information need and new media choice*
005 Roger H.J. Demkes, *COMET: A comprehensive methodology for supporting telematics investment decisions*
006 Olaf Tettero, *Intrinsic information security: Embedding security issues in the design process of telematics system*
007 Marike Hettinga, *Understanding evolutionary use of groupware*

Samenstelling promotiecommissie:

Voorzitter, secretaris:	prof. dr. W.H.M. Zijm (Universiteit Twente)
Promotor:	prof. dr. ir. L.J.M. Nieuwenhuis (Universiteit Twente)
Assistent promotor:	dr. L. Ferreira Pires (Universiteit Twente)
Leden:	prof. dr. ir. M. Aksit (Universiteit Twente)
	prof. dr. ir. C.A. Vissers (Universiteit Twente)
	prof. dr. J. Fischer (Humboldt Universität zu Berlin)
	prof. dr. ir. M.R. van Steen (Vrije Universiteit)
	dr. V.C.J. Gay Hdr. (Université Pierre et Marie Curie)

ISSN 1381-3617 (CTIT PhD.-thesis series number 02-46)
ISSN 1388-1795; No. 008
ISBN 90-75176-35-X

Centre for Telematics and Information Technology,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
Telephone: +31-(0)53-4898031; Fax: +31-(0)53-4891070

TOWARDS AN ADAPTABLE QOS AWARE MIDDLEWARE FOR DISTRIBUTED OBJECTS

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 9 Januari 2003 om 16.45 uur.

door
Aart Tijmen van Halteren
geboren op 13 mei 1970
te Bunschoten-Spakenburg

Dit proefschrift is goedgekeurd door:
prof. dr.ir. L.J.M. Nieuwenhuis (promotor)
dr. L. Ferreira Pires (assistent-promotor)

Preface

One of the most spectacular developments of the last decades is the Internet. The Internet offers instant access to an unmatched amount of information and as a result has greatly influenced society and the way business is conducted. Telecommunication providers realise they have to reposition their business in the new market, where service provisioning is subject to a higher degree of competition and more dynamic than before the Internet became big. Understanding, managing and marketing telecommunication services requires a telecommunication provider to 'think Internet'. This calls for a good understanding of the basic technologies that underlie the Internet. Telecommunication providers must apply the appropriate architectural concepts during the design and implementation of services.

Just as the current Internet, the future Internet will be a conglomerate of hardware and software systems often obtained from several vendors, which are interconnected through a variety of (tele)communication networks. The services that are designed, developed and deployed for this complex heterogeneous system are called telematics services. The physical location of the entities that constitute a telematics service, as well as the logical location of the functionality of a service, makes a telematics service an inherently distributed service. This distribution is the result of the physical allotment of functionality, i.e., distribution by nature, and the logical allotment of functionality, i.e., distribution by design. Both reasons for the distributed character of telematics services necessitate a careful design of the entities or components that constitute a service. In particular the collaborations of service components are of key importance to the behaviour of the service as a whole.

The inherent distribution of the functions of a telematics service requires some notion of the boundary where an entity that provides a function can interact with its environment. The boundaries of the entities of a telematics service are often defined as interfaces. It is commonly

accepted to apply object technology for the design and development of a software system. Features of objects, such as abstraction, encapsulation, polymorphism and extensibility, make object technology a suitable means for the design and implementation of a service. In the object technology approach an interface is used to describe how an object can interact with its environment. If this is applied to telematics services, we can design these services as a set of collaborating objects and thus benefit from all the advantages introduced by object technology.

Over the past few years middleware has become an important technology for telematics services. Middleware is a software infrastructure that masks distribution and technology aspects, such as the location of software components, the implementation language and underlying transport protocols. Middleware encompasses the mechanisms to exchange and transform data between independently developed and deployed software components. Middleware offers the software infrastructure that service components can use to collaborate. Examples of middleware platforms are Microsoft DCOM, Enterprise Java Beans (EJB), OMG CORBA and W3C Simple Object Access Protocol (SOAP). In case telematics services are designed as a set of collaborating objects, then middleware for distributed objects offers a software infrastructure that simplifies the design, development and deployment of telematics services.

Distributed object technology and middleware for distributed objects are useful means to realise service functionality. A logical next step is to also provide the means to realise the qualitative aspects of a telematics service. Service users expect a particular quality level in terms of availability, responsiveness, and safety. Users of a service get accustomed to a certain level of Quality of Service (QoS). Lack of QoS will dissatisfy existing users and prevent new users to start using a service. One way for service providers to offer a high level of QoS is by over-dimensioning the resources for processing, communication and storage in order to always have sufficient capacity to serve all users at all times. However, the increasing competition in the telecommunication market and the regulatory constraints require a service provider to compete with service offers. One way for a service provider to compete is to differentiate quality levels between various user groups, based on some marketing strategy, in order to discriminate from other service providers or to create a competitive edge.

Differentiation of quality levels can be realised several ways. One way that is economically not feasible is to build separate hardware and software infrastructures for each user group to achieve quality differentiation.

The resources for communication, processing and storage in a telematics system are generally scarce. A service provider will most likely aim for a single infrastructure that can serve various user groups with differentiated quality levels. Dimensioning the resources in a telematics

system is a big challenge. In a large-scale distributed system, a-priori calculation of how many resources are needed to guarantee QoS levels at all times is very difficult and often imprecise. This is because mathematical models of telematics systems assume a distribution function (with some mean and variance) for the arrival of service requests. In practice these assumptions do not match with reality. This is because the distributed system is subject to two kinds of changes:

- Run-time: this includes the number of users of the system, the number of services used, the load generated by a service, unavailability of resources due to planned or unplanned downtime.
- Evolutionary: this includes the availability of new hardware and software components, increased processing and communication resources as a result of the availability of new technology, and the availability of new mechanisms for resource reservation.

Next-generation middleware should offer facilities that mask the run-time and evolutionary changes in QoS as much as possible from telematics services. Middleware forms the execution environment of telematics services and it should offer an abstract view of the QoS offered by the underlying communication, processing and storage resources.

In this thesis we investigate how state-of-the-art object middleware can be improved in order to further simplify the design, development and deployment of telematics services. In our view, object middleware is the only cost-effective means for realizing telematics services in a fast, extensible and customisable way. Contemporary object middleware does not offer sufficient facilities to control the qualitative aspects of a service. This thesis builds on the premise that QoS support should be an intrinsic part of the middleware. A service provider that is capable to provide a QoS aware object middleware based software infrastructure can easily sell the use of such an infrastructure to other service providers. As a result the software infrastructure service provider will play a key role in the value chain of future telematics service provisioning.

Making QoS aware middleware adaptable to the run-time and evolutionary changes of a distributed system enables service providers to offer services with differentiated qualities in a cost effective manner.

This thesis aims to:

1. Construct a reference model of object middleware and clearly separate the qualitative aspects of the object middleware infrastructure from the QoS concerns of a telematics service;
2. Advance object middleware technology through the addition of facilities that can control the qualitative aspects of the objects deployed on the middleware;

3. Validate our objective to make middleware QoS aware by developing an infrastructure service that can leverage existing mechanisms for QoS establishment and control to the middleware level.

These objectives are achieved as follows:

- Analysing the structure of various object middleware platforms and construct a reference model for object middleware;
- Presenting a model that application designers use to express QoS aspects of an object middleware based telematics service;
- Defining the relationship between QoS design aspects and the QoS functions and mechanisms that realise a QoS requirement;
- Designing a prototype infrastructure service, called QoS Provisioning Service (QPS) for managing QoS aspects of object middleware;
- Comparing the QoS delivered by standard object middleware platform with the QoS delivered by an object middleware platform enhanced by QPS

Acknowledgements

About ten years ago, one of my dreams was to buy a mini-van ('busje' in Dutch) and take it for a long drive throughout Europe, with some of my friends. We called this undertaking the B. Usje enterprise. That enterprise never came into existence, due to a lack of funds and the urgent need to finish our studies.

About five years ago I encountered on another journey. The bus on the cover of this book can be seen as symbol for an alternative B. Usje enterprise. With the completion of this thesis I look back on a long trip full of exciting and character building experiences. During this trip I encountered many people that have contributed to the excitement of the mission. Now the time has come to express my appreciation.

This trip would not have been possible without my supervisor Bart Nieuwenhuis. He ensured that there were always sufficient funds to continue travelling, even when the going got tough. He enabled me to discover and explore and was only one phone call away in case I needed direction. I'm very thankful for his support and guidance and that he persisted until the very end.

The other main influence on the final destination of this trip originates from Luis Fereirra Pires. He got involved in a later phase of the journey and has made a significant impact on the quality of this work. His meticulous comments on my initially unstructured writings have been of great value.

Mehmet Aksit, Chris Vissers, Joachim Fisher, Maarten van Steen and Valerie Gay have evaluated the results of my journey, as they have been written down in this thesis. I value the time and effort they spent in ploughing through this book. In addition, the discussions with my committee members in preparation of the public defence were uplifting and motivated me to continue working in the area of open distributed systems.

The starting point of my PhD research voyage was at KPN Research. Before I was ready to start my journey KPN Research provided the

necessary education. The environment that KPN Research provided has been most stimulating in many ways. For over 5 ½ years I was able to enjoy a fruitful collaboration with the colleagues in Groningen. The encouragements of my roommate George Huitema have been fundamental in continuing with my research despite opposition from others. Maarten Wegdam also deserves a special mention for his role as a sparring partner throughout the entire journey.

Collaboration with the colleagues of KPN Research Enschede, under the leadership of Wim Jonker, has also been an interesting and learning part of the trip. Especially Gina Fábián proved fundamental in the production of papers and promoting our QuAM group to the outside world.

During my trip I visited many countries inside and outside Europe, due to the research projects I was involved in. The cooperation in national and international projects has resulted in significant groundwork for this thesis. More specifically, I'd like to thank the people that I worked with in the Amidst project and the EURESCOM P715 and P910 projects. Amongst others, Olaf Kath was one of those persons that I enjoyed working with.

A journey usually involves interesting excursions and side steps that are needed in order to discover the main road. I was privileged to supervise several graduate students. The efforts of Maurice Schreiner, Kristian Helmholt, Petra Oldengarm, Henk & Alwyn, Marcel Harkema, Dirk Jaap Plas, Jeroen Gommans, Marco van de Logt and Paul Koster have all been necessary to find the main road. I'd like to thank every one of you for travelling a part of the journey with me.

Finalising my journey would not have been possible without the support of the University of Twente. I'm especially indebted to Dimitri Konstantas for taking me on board of the Application Protocol Systems group and providing me with the fuel to finish the last mile.

The B. Usje enterprise would never have been successful without the support of many of my friends and family. This acknowledgment would become too extensive if I would mention each of you individually. But I want to mention my wife's parents because they have supported and encouraged me as if I was one of their own children.

Finally, I want to express my overwhelming gratitude to my best friend, partner in life, wife and mother of my children: Irene. You were there next to me all the way. You have shared with me al the trials and troubles that we had to overcome to reach this result. With the completion of this thesis, in fact I feel that we have just reached the beginning.

Aart van Halteren
Overdinkel, November 2002

Contents

Chapter 1

Introduction

This chapter presents the motivation for this thesis, its objectives and the approach taken to achieve these objectives.

1.1 Developments in the telecommunications industry

Over the past couple of years developments in the telecommunications industry have been far from stable. Mergers and acquisitions of telecom operators have reshaped the business relations between today's telecommunication service providers and their customers. Legislators require telecommunication service providers to open up their network and allow other players to operate on the market. Due to regulations and the economic climate, fierce competition between incumbent and new players in the telecom market has risen. Bankruptcy threats traditionally strong players. All in all the complexity of the telecommunication market is growing.

Besides these developments, the Internet has grown explosively and now governs today's service provisioning developments in many aspects. It is not surprising that the Internet has a major influence on our economy, on the telecommunications industry and on our Information and Communication Technologies (ICT).

Internet technology has a major impact on society and influences the business of telecommunication providers. Traditional telecommunication services such as telephone and value added voice services, i.e. the so-called Intelligent Networks (IN), are gradually outdated by new and advanced services. Telecommunication service providers have moved from offering communication services to higher value services such as content packaging, content delivery, location based information services and other personalised services [NiHa99, HNSW99].

Telecommunication service providers are generally required to provide new services faster, reduce cost of service development, deployment and operations and to personalise services to customer needs [BHK+96]. These requirements impact the infrastructure used to deliver services.

Traditional telecommunication infrastructures are the result of decades of development and technological change. Novel infrastructures are developed from standard off-the-shelf ICT components and constructed in the time frame of a couple of years. Telecommunication service providers are looking for technologies that are aligned with their business needs. Consequently, manufacturers of telecommunication products are incorporating Internet technologies and standards into the products offered to service providers.

1.2 Business justification

Telecommunication manufacturers are forced to move to standard ICT solutions and practices to construct their products, as a result of the developments discussed in the previous section.

In this thesis we focus on one of these standard ICT solutions called *middleware for distributed objects*. A potential spin-off from this thesis is that service providers have access to standard software components that enable differentiation of service quality between various user-groups.

Middleware for distributed objects is a technology that creates internal and external benefits to service providers. The internal benefits are gained through increased efficiency, lower operational costs and the ability to rapidly change business practices. The external benefits originate from the ability to enter new markets and to find new ways to reach customers [NiHa99]. Middleware is a software layer that integrates software components into services and thus plays a key role for an agile service provider. However, if a service provider is not capable to manage the qualitative aspects of its middleware based services, the middleware becomes a showstopper.

In a mature and open market, as in many other industries, the primary economic forces will be determined by customers selecting from a wide range of services and products that differ with respect to price and quality [NiWi00]. In the years to come, the quality of service will not primarily be determined by the available bandwidth of our networks anymore. Quality of Service (QoS) will be more and more determined by the availability of *all* the resources needed for service provisioning, e.g., the communication links, the routers, the computing devices, and data stores.

Middleware for distributed objects acts as a point of convergence for all these resources. Our goal with the results presented in this thesis, is to increase the use of middleware through the addition of mechanisms that establish and maintain some QoS level required by the users of a service.

1.3 The role of middleware in open systems

Currently, the Internet is a conglomerate of hard- and software systems obtained from several vendors, which are interconnected through a variety of (tele)communication networks. In the future this will not change. The Internet will remain an open system that has many vendors, consists of parts that are implemented with various technologies, can scale to a size beyond the telephone system, and evolves gracefully. The Internet enables disparately developed applications to collaborate and share data.

Heterogeneity in open systems is inevitable for historic and technological reasons. For example, Moore's law dictated that the number of transistors per integrated circuit would double every eighteen months [Mo65]. Even after several decades this law still holds. As a result, faster and more powerful computer systems become available. A similar growth in available network bandwidth can be observed due to more powerful routers and switches. The speed of technological developments make it necessary for new and existing hardware systems to interwork, because existing systems have not been written down when new systems arrive on the market. Therefore existing systems are not replaced even when new technological superior systems become available.

Not only the technological developments of hardware systems are moving fast, but also new operating systems and new programming languages emerge, accompanied with new development tools. Software developers usually have specific knowledge of a limited set of operating systems and development environments. This conflicts with the need to develop software for heterogeneous systems. As a result, companies are forced to hire highly trained specialists in order to develop applications for their heterogeneous systems. However, the understanding of how to build open applications cannot be in the hands of a few specialists, because these specialists are scarce which makes open application development very expensive.

Ideally, telematics services developed for open systems should be enabled to interwork with new and existing telematics services. Interworking is the ability of applications components that constitute a telematics service to collaborate and share data. Interworking enables open distributed systems to simply grow to large-scale distributed systems, because relatively small open distributed systems can be easily

interconnected with other distributed systems to form bigger ones. This requires application components to be developed using rules for interoperability, which are captured in standards. For example, the Word Wide Web (WWW) has grown to a global size information retrieval system, due to broadly accepted Internet standards [IETF97, W3C98].

The speed of progress in hardware systems, network technologies and operating systems, complemented with the need to develop software that can grow along with the size of the open system, leads to the following observations [He92]:

- New and existing hard- and software systems must interwork;
- Open application development should not be in the hands of a few specialists;
- Applications should comply with rules for interoperability.

A cost-effective solution to the development of telematics services in open systems is necessary. Middleware has emerged as such a solution. Middleware serves to shield application components from the heterogeneity of the underlying computer platforms and networks and to provide effective support to a diversity of telematics services.

The role of middleware in open distributed systems is to offer transparencies to the designer and developer of telematics services. This means that the mechanisms used to overcome problems and details of distribution are hidden to the application components. This is cost-effective, since designers of a telematics service do not have to re-invent and re-implement the mechanisms to overcome problems of distribution over and over again.

1.4 Objectives

The objectives of this thesis are derived from the developments and observations described in the previous sections and are reflected in its title: "*Towards an adaptable QoS aware middleware for distributed objects*".

The term *middleware for distributed objects* refers to a supporting infrastructure for telematics services. The focus of this thesis is on the simplification of the design, development and deployment of telematics services in distributed heterogeneous systems. We assume that object technology and middleware technology are important means for a designer to achieve such a simplification.

Designing telematics services is a complex task performed by many designers, each allocated with a specific part of the design process. In this thesis we distinguish between designers of telematics applications and

designers of the supporting infrastructure. The application designer focuses on the behaviour of the service towards the end-user. The infrastructure designer focuses on the allocation of resources to the application components and the communication between these components. In fact these aspects can be designed once and re-used by many applications.

If we assume that designers are designated a specific role in the design process, we first have to investigate which roles designers can have. If we have defined these roles, then for each of the designer roles specific aspects of the telematics service needs to be modelled. In fact, we assume that various models of the same system are needed, each model highlighting a different aspect of the system. In this thesis we study the relationship between designer roles and the models needed for this role.

Obviously, if we have various models highlighting different aspects, we need to be sure that these models are consistent and no contradictions exist. Moreover, we address the question how the various models are interrelated.

Middleware as it exists today is the result of many years of independent developments by different organisations and companies. Each development targeted for a number of concerns while developing and applying middleware technologies. Our objective is to search for the main concerns that guided the various developments and to identify the commonalities and differences. This is our basis for the construction of a reference model for middleware. We use this reference model as starting point for the development of QoS mechanisms, in order to guarantee that our solutions are not technology or platform dependent but sufficiently generic to be applied in many different environments.

To assist the designers of a telematics service in their task, we search for answers to questions such as:

- What roles can designers of a telematics service have?
- What aspects of a distributed system should designers model, depending on their role?
- How are the various aspects of a design related?
- What are the common concerns of early and contemporary middleware platforms?
- Can we construct a reference model that captures the common concerns of middleware for distributed objects?

The term *QoS aware* refers to the qualitative aspects of the middleware. Awareness of the qualitative aspects of a telematics service starts with the design of a telematics service. A designer states the qualitative properties required from a telematics service such as performance, availability and safety. The main challenge of this thesis is to make QoS support an intrinsic part of middleware, in order to facilitate the realisation of qualitative

properties of a design. Middleware that simplifies the realisation of QoS concerns of a telematics service is considered to be QoS aware.

To assist with the design of qualitative aspects of a telematics service, we search for answers to questions such as:

- How can designers model the QoS aspects of a telematics service?
- What modelling concepts are needed to express the QoS capabilities of a QoS aware middleware?
- What modelling concepts are needed to express the QoS requirements of the parts of a telematics service?
- Can we hide the means to achieve QoS awareness while still remaining flexible with respect to various QoS requirements?

The term *adaptable* refers to a property that QoS aware middleware for distributed objects must have, both on short and long terms.

Short term adaptability, i.e., concerning run-time changes, is needed because in a heterogeneous system the qualitative properties are subject to change due to a changing number of users of the system, the number of services used and unavailability due to planned or unplanned downtime.

Long term adaptability, i.e., concerning evolutionary changes, is needed because new computing and communication resources with additional functionality become available over time and object middleware should incorporate this functionality.

Our objective is to contribute to the design of adaptable QoS aware middleware that hides the run-time and evolutionary changes as much as possible from a telematics service.

To introduce adaptability into QoS aware middleware, we search for answers to questions such as:

- What are the generic means to establish agreements on the QoS that a telematics service requires?
- Can we construct a generic framework, adaptable to evolutionary changes, that simplifies the establishment of such agreements?
- What are generic means to maintain to QoS of a telematics service?
- Can we construct a generic framework, adaptable to run-time changes, that simplifies control of QoS?

The objectives of this thesis can be summarised as follows:

1. Construct a reference model of object middleware and clearly separate the qualitative aspects of the object middleware infrastructure from the QoS concerns of a telematics service;
2. Advance object middleware technology through the addition of facilities that can control the qualitative aspects of the objects deployed on the middleware;

3. Validate our objective to make middleware QoS aware by developing an infrastructure service that can leverage existing mechanisms for QoS establishment and control to the middleware level.

1.5 Approach

A QoS aware middleware is a software infrastructure that supports interactions between application components and allows these interactions to be subject to quality requirements of the application components. In this thesis a generic architecture for the specification and enforcement of QoS aware interactions is proposed. This architecture is validated by means of a design and implementation of a component that acts as a broker between the QoS capabilities of the middleware and the quality requirements of application components. This component is called the QoS Provisioning Service (QPS).

To increase acceptance of the architecture, the concepts and designs proposed in this thesis, we search for solutions that are in line with the already existing standards, architectures, technologies and engineering practices for open distributed systems. We therefore present an overview of the research area, which serves as a starting point for introducing QoS awareness into middleware for distributed objects.

To further justify the proposed extensions to existing object middleware and to ensure that these extensions are not tied to one specific object middleware, a reference model for object middleware is presented. This reference model captures the technological advancements of middleware systems over the past decades.

From this reference model, we develop a QoS aware object middleware. We consider the QoS aware middleware from an external and an internal view. In the external view the application components use the QoS capabilities of the middleware to express QoS requirements. The external view, depicted in Figure 1-1, hides the functions and mechanisms used to establish and maintain QoS requirements from application components.

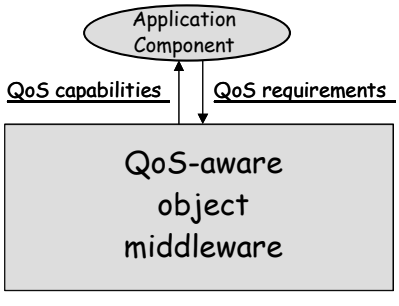


Figure 1-1 External view of a QoS aware object middleware

The object middleware is responsible to inform application components of its QoS capabilities and to see if it can meet the QoS requirements. The internal view exposes how QoS requirements are established and maintained. Figure 1-2 shows the internal view of a QoS aware middleware. The QoS Provisioning Service (QPS) uses a set of mechanisms to realise QoS requirements of application components.

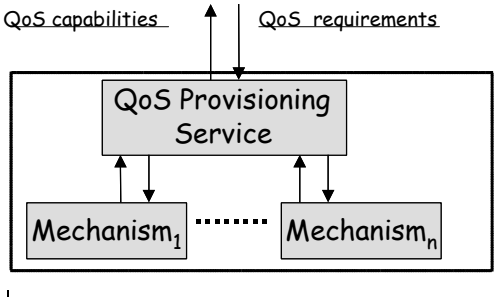


Figure 1-2 Internal view of a QoS aware object middleware

The architecture of a QoS aware object middleware and the design of the QoS Provisioning Service are validated with a prototype implementation. With this prototype the solutions proposed in this thesis are implemented and evaluated, by comparing the effects of the QPS with a non-QoS aware object middleware.

This thesis is structured as follows:

- Chapter 1 *Introduction* provides a global definition of the problem area. It defines the area of research, scope and objectives and presents the relevance of this work from the perspective of a telematics service designer and the perspective of a service provider;
- Chapter 2 *Modelling concepts and principles* introduces the modelling concepts and principles used throughout this thesis.
- Chapter 3 *Overview of the research area* presents the necessary background information that is relevant for this work and presents the technology

currently available in this area. This background forms the basis for a more detailed motivation of the thesis objectives;

- Chapter 4 *An object middleware reference model* describes the structure of current object middleware systems. It presents the key internal entities of an object middleware system, their functionality and their interworking. The view presented in this chapter is an abstract and generic representation of object middleware platforms as CORBA, Enterprise Java Beans and SOAP.
- Chapter 5 *Models for QoS aware middleware* provides a meta-model for modeling the QoS requirements and QoS capabilities of a software infrastructure for telematics services;
- Chapter 6 *Design of a QoS provisioning service* introduces the QoS Provisioning Service (QPS). This service is responsible for managing QoS aspects of an object middleware and is designed to cope with the changing quality levels of the middleware due to run-time and evolutionary changes;
- Chapter 7 *Conclusions* presents a summary of the conclusions drawn throughout this thesis, evaluates how our objectives have been achieved and identifies directions for further research.

Chapter 2

Modelling concepts and principles

This chapter introduces the modelling concepts and principles that are relevant for the rest of this thesis.

One of the objectives of this thesis is to develop support for the enforcement of qualitative aspects in applications. To be able to describe this support an appropriate set of modelling concepts is needed. The concepts and principles introduced in this chapter are tailored to the modelling of distributed applications, which are applications deployed on a set of geographically distributed computing systems.

The structure of this chapter is as follows: section 2.1 presents the typical characteristics of a distributed system. Section 2.2 outlines the fundamental concepts that are employed for the development of specifications of a distributed system. Section 2.3 discusses the use of objects as elementary units of specification. Section 2.4 introduces the notion of viewpoints that enables the separation of concerns for the specification of a distributed system. Section 2.5 introduces the notion of middleware for distributed objects and defines some additional terms related to middleware. Section 2.6 discusses an intuitive of notion of Quality of Service (QoS) in the context of middleware. Finally, section 2.7 presents the conclusions regarding the concepts and principles outlined in this chapter.

2.1 Distributed processing

For the past decades, the cost of processing power as well as the cost of network technology and (tele)communication services has been decreasing. As a result, companies and individuals can afford more processing power and are connecting an increasing amount of computing systems through communication networks. Computing systems are connected through networks, because the users of these systems have the need to share

resources, such as processing power or storage capacity. The need to share resources has become more apparent with the increasing amount of information that is made available online.

Distributed computing or distributed processing is concerned with sharing processing power and other resources through communication networks. Resources are shared to accomplish a task that cannot be performed on a single computing system. For example, consider a software development team that uses a set of personal computers that are connected through a network in order to collaboratively develop software. As a shared resource the team requires a storage space to exchange the pieces of software produced by the team members. Distributed processing is concerned with a set of interconnected computing systems that collaboratively accomplish a task by sharing resources.

Distributed processing implies that the systems involved are geographically spread. The computing systems involved in a distributed processing task are part of a distributed system. Distributed systems are employed to support the sharing of resources and the distribution of work over multiple computer systems.

2.1.1 Distributed systems

Several definitions of 'distributed systems' have been provided [BlSt97] [Mu93]. Leslie Lamport has been attributed the following famous definition: "A distributed system is one that stops you from getting any work done when a machine you've never heard of crashes".

For the purpose of this thesis, the following definition of a distributed system is employed:

Definition 1
Distributed System

A distributed system is a system that consists of multiple, autonomous processing elements that are geographically distributed and, therefore, cannot share primary memory, but cooperate by sending messages to each other over a communication network.

The processing elements of a distributed system are generally not owned by one individual or a single organisation. The resources assigned to a distributed processing task are only shared at the discretion of an owner of a resource. A prominent term in definition 1 is '*autonomous processing elements*'. Distributed systems, according to this definition, distinguish themselves from more tightly coupled systems such as parallel systems or multi-processor systems, which share primary memory.

2.1.2 General characteristics

The physical distribution of the processing elements that comprise the distributed system and the way these elements communicate entails in a number of characteristics, sometimes referred to as 'symptoms of a distributed system' [Mu93]. The following characteristics have been identified [ODP1]:

- *Remoteness*: the parts of a distributed system may be spread geographically; interactions between the parts may be either local (i.e., when parts are located in the same place) or remote (i.e., when parts are located in geographically disperse places);
- *Concurrency*: any part of a distributed system can execute in parallel with any other part;
- *Lack of global state*: the global state of a distributed system cannot be precisely determined;
- *Partial failures*: any part of a distributed system may fail independently of any other part;
- *Asynchrony*: there is no single global clock that drives communication and processing activities. Related changes in a distributed system cannot be assumed to take place at a single instance in time.

2.1.3 Characteristics of an open distributed system

The resources that constitute a large distributed system, such as, for example, the Internet, are not manufactured or owned by a single organization. Therefore, distributed systems often cross multiple technological and organizational boundaries. Collaborative processing between the parts of such distributed systems requires certain agreements between the manufacturers of the parts. This leads to the notion of *open* distributed systems.

Different vendors can build the parts that constitute an open distributed system. This gives the owners of a distributed system the option to obtain parts from various vendors. Each vendor can choose its own way to implement a part of a distributed system, yet the parts can interoperate. Openness of a distributed system implies that there must be consensus on the rules that guarantee the interoperability of the parts. This consensus can be established either through a formal standardisation process (dejure standardisation) or through a consortium of vendors (de-facto standardisation).

According to ODP-RM, open distributed systems have a number of additional characteristics [ODP1]:

- *Heterogeneity:* in an open distributed system, there is no guarantee that parts are built using the same technology and the set of various

technologies will certainly change over time. Heterogeneity applies to many aspects of an open distributed system: hardware, operating systems, communication networks and protocols, programming languages, etc.;
- *Autonomy:* an open distributed system can be spread over a number of autonomous management or control authorities, without any single point of control. The degree of autonomy specifies the extent to which processing resources and associated devices (printers, storage devices, graphical displays, audio devices, etc.) are under the control of separate organizational entities;
- *Evolution:* during its working life, an open distributed system generally has to face many changes, which are motivated by technical progress, enabling better performance at a lower price, by strategic decisions about new goals, and by new types of applications;
- *Mobility:* the sources of information, processing nodes, and users may move around in space. Programs and data may also be moved between processing elements, e.g., in order to cope with physical mobility or to optimize performance.

A designer that needs to design a distributed system has to take into account many, if not all, of these characteristics. Consequently, the design of a distributed system is a complex task.

2.2 Modelling distributed systems

Models are used to manage the complexity of a distributed system, making it easier to understand the characteristics of a distributed system. Modelling is the activity of capturing the characteristics of a system that are of interest for some specific goal, while abstracting from other characteristics. A modelling activity results in one or more models of the system. We use the following definition for a model:

Definition 2 Model

A model is a simplified representation of a system that accounts for some of its known, inferred or desired characteristics, while purposely abstracting from other characteristics with the intent to further study or define system characteristics.

A model is created in order to obtain an abstraction of a system. Abstraction is a technique that allows one to concentrate on aspects of a system that are considered essential for a certain purpose. Abstraction is related to the purpose for which an abstraction of a system is developed.

During the process of modelling a system one should be aware of the aspects that are essential for the purpose of studying or defining a system. A

model of a system should only consider characteristics of the system that are relevant for the purpose of developing the model.

For example, consider two models of a central processing unit (CPU). The first model is intended to study the performance of a CPU, e.g., how many floating-point operations per second the CPU can perform. The second model is intended to study the heat emission of the CPU for a given workload. Both models concern the same CPU, but are created with different purposes and therefore highlight different aspects of the system. Both models are abstractions of the same system.

2.2.1 Model, design and specification

The set of concepts that is used to create models is called a meta-model. A model developed from this set of concepts is an instance of the meta-model. Models are developed to either study an existing system (analysis) or to produce a new system (synthesis). When a model is developed for the synthesis of a system, such a model is called a design. A design is a prescriptive model of a system. A designer may create several designs for the synthesis of a system.

Designs are conceptual models that are conceived and manipulated in a designer's mind. Due to the limited capabilities of the human mind for capturing complex designs, we are forced to represent designs, in order to allow documentation, communication and reasoning about the characteristics being represented. In addition, when designs are represented in a form that can be interpreted by tools on a computer, the analysis and manipulation of a design can be (partially) automated.

The representation of a design in a(n) (electronic) document is called a specification. A specification consists of symbols that represent the modelling concepts from the design. The complete set of specification symbols and the rules that determine the arrangements of these symbols that are allowed is called a specification language.

The above observations lead to the following relations between design and specification [Qu98, Pi94] as depicted in Figure 2-1.

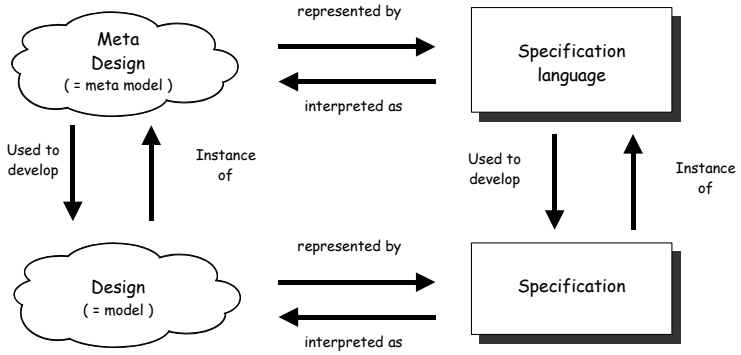


Figure 2-1
Relations between design and specification

The relation between a design and a meta-design can be applied recursively, such that a meta-design is developed from a meta-meta-design or that a design is used as a meta-design. To understand how a design must be used, the *meta-level* of a design must be known. In a similar way a specification language is designated a meta-level. Consequently, a specification that represents a design at a specific meta-level can be used as a specification language to develop a lower level specification.

Careful use of meta-levels in the design of an open distributed system enables a designer to create a representation of a design that can be interpreted and manipulated by the system itself. Run-time interpretation and manipulation of a design by a system is called *reflection*.

The distinction between a design and a specification enables an abstract set of design concepts to be represented using various specification languages. The choice for a specification language depends on how and by whom a specification is used. For example, a human-readable specification may use graphical symbols to represent a design, whereas a computer-readable specification consists merely of a sequence of bits somewhere in the memory of a computing system. Both specifications represent the same design.

2.2.2 Refinement, decomposition and abstraction

A designer may use several designs to model a system. Designers may choose to develop several designs because they need to further manage the complexity of a system. Designs can be developed at different levels of granularity. A coarse-grained design presents less detail than a fine-grained design.

A designer can develop a coarse-grained design and then create a more fine-grained design that conforms to the coarse-grained design. The fine-grained design shows a more detailed representation of the system. The

process of adding more detail to a design is called refinement. Refinement is a technique that allows a designer to address one concern at a time. Refinement is the opposite of abstraction.

Decomposition is a special case of refinement. Decomposition is achieved by decomposing some or all of the parts of a coarse-grained design into a design that is more detailed. In this refined design some original parts may be replaced by a number of subparts. Decomposition of parts into subparts can be repeated as often as needed and usually stops when the parts of a design are trivial to produce or can be bought from a vendor.

A designer refines a design by adding more details. One way to add more detail is to specify the internal structure of some or all of the parts of a design. Such a refinement of a design leads to a new level of decomposition of a system. Each level of decomposition forms an abstraction of the system. The various levels of decomposition that result from the refinement of a coarse-grained design into a more fine-grained design are closely related. On one hand, a design at level N results in a design at level N+1, through refinement. On the other hand, a level N design forms an abstraction of a level N+1 design. Related decomposition levels, are also called related *abstraction levels.*

To further illustrate the notion of abstraction levels, consider the design of an Automatic Teller Machine (ATM). In an initial design (design N) a designer considers the ATM machine as a black box. A customer can collect money from the ATM machine after a correct personal identification number (pin) and the amount of money have been entered. In a refined design (design N+1), the ATM machine is decomposed into a user interface part, a pin verification part and a part that controls the money collector. Both designs and their relationships are shown in Figure 2-2.

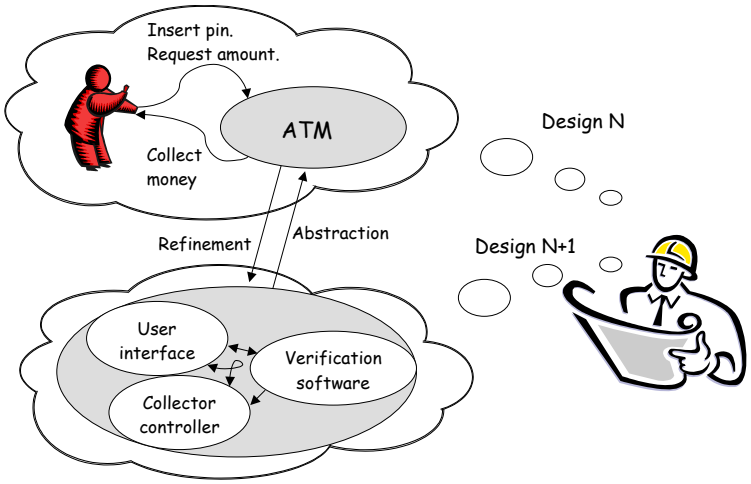


Figure 2-2 Two related designs of an ATM machine

Design N+1 is a refinement of design N. Both designs represent the ATM system at different abstraction levels, because design N+1 is a more detailed representation of the system as design N.

The stepwise refinement of a design entails a top-down approach to the design of a distributed system. However, in practice the trajectory followed by a designer of a distributed system is not simply a straight path of refinement steps until the parts can be realised in hardware or software. A system can be decomposed in many alternative ways. Therefore, a designer may create several alternative refinements, and through a process of trial and error decide which refinement is the best to continue from. The design process may be guided by bottom-up knowledge about already existing parts that are represented in a design.

2.3 Object modelling

A designer is free to choose which primitive modelling concepts are used to model a system. When a designer uses *object* as the primitive modelling concept, the resulting models are called *object models*. An object model is a model of a system whereby the parts of the system are represented as objects. Object models are used as abstractions of a distributed system.

Object models are developed to represent the concrete software parts of a distributed system or to capture the conceptual parts of a distributed system. In the latter case, the objects of the model may not be represented as corresponding software objects. In case an object model is developed to represent the software parts, the objects of that model can be found as software artefacts in the distributed system.

The object models developed in this thesis concentrate on the representation of concrete software parts of a distributed system.

2.3.1 Characterisation of an object

An object is an abstraction of a real world entity. An object is characterised by a set of actions that this object can perform. An action is an activity in which an object is involved. Actions can be internal, which means they occur inside the object. Actions can also be external, which means they occur outside the object. An object consists of one or more interfaces and an object core.

The *interface* of an object defines the potential external actions that can take place in which the object can be involved. The set of potential external actions is described as a set of named operations. An operation defines a potential action of an object and optionally has a set of typed parameters. The interface signature defines an interface. The interface signature is a

collection of the signatures of the operations. The name of an operation, the set of typed parameters and the return type define the operation signature. An operation signature may also contain an exception that is raised when an external action results in an abnormal condition. Operation signatures are equivalent to a function definition, i.e., an operation takes zero or more parameters as input, does some processing and returns an optional and possibly empty result. The object core determines the processing that results from calling an operation.

The *object core* represents the internal functioning of an object. An object core supports methods and attributes. Attributes can be inspected and modified by respectively get and set operations defined in the objects' interface. Methods implement the operations defined in an interface. Methods determine the behaviour of an object, i.e., the set of internal actions performed by an object as the result of an external action. The object core also manages the internal state of an object. Depending on the requirements on an object, its state can be maintained in volatile storage or written to persistent storage. In the latter case it is possible for an object to outlive the period in which it is active. In case the internal state of an object is empty it is called a stateless object.

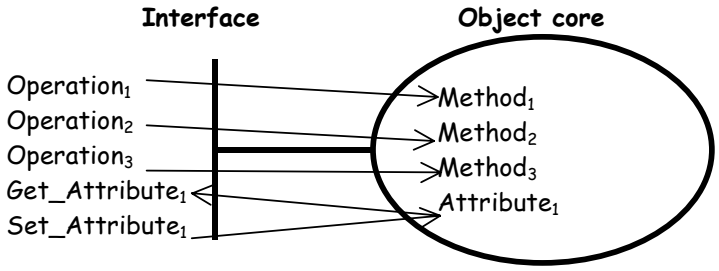


Figure 2-3 Graphical representation of an object

The environment of an object can only change the state of the object through its interface. The object core and interface are the primitive building blocks of an object, but both should coexist for proper operation. The close linkage between an interface and an object core is shown in Figure 2-3.

The typical characteristics of an object are the following [Sz97]:
- An object is a unit of instantiation; it has a unique identity;
- An object has state; this state can be persistent;
- An object encapsulates its state and behaviour.

From these characteristics of an object a number of observations can be made. Since an object is a unit of instantiation, it is a discrete item that cannot be instantiated partially. The unique identity of an object allows it to be identified despite changes in its state that can occur during its lifetime. The set of all potential actions that an object may perform is defined as the *behaviour* of the object. The s*tate* of an object is characterised by the condition of its internal data and represented by the content of its internal memory at a given instant. State and behaviour are closely related concepts. The current state of an object is determined by its past behaviour. Conversely, potential behaviour of an object is determined by its present state.

The state of an object can be changed through an internal action or through an external action. External actions happen at the interface of an object. The interface further emphasises that an object encapsulates its state and behaviour. These characteristics of an object lead to the following definition:

Definition 3 Object

An object is a discrete design concept, that encapsulates its state and behaviour and is used to model a software entity. An object is subject to internal actions, which occur at the object core, and external actions, which occur at one of its interfaces.

Since an object is a unit of instantiation, there must be some building plan that describes how it is instantiated, i.e., what are its initial state, state space and behaviour. Such a blueprint, or construction plan for an object is called a *template*. Because objects can be instantiated from a template, an object is also referred to as an *instantiation* of a particular template. Many unique objects can be constructed from a template and all objects constructed from a template have similar characteristics. However, two objects constructed from the same template have different identities. These objects may differ in their state and consequently they may behave differently as the result of an external action.

Objects that have a common structure for one or more of their attributes, operations or methods can be grouped together. Objects can be grouped into an object set based on a predicate. Such a predicate determines the *type* of the object. A set of objects for which the same predicate holds is called a *class*. Consequently, objects that have the same type belong to a class.

In object oriented programming languages the type of an object is determined by its template, i.e. objects instantiated from the same template have the same type. The predicate that determines the type of an object in this case, is that the object is an instance from a specific template. As a result the template also defines a class. Therefore, ‘class’ is a notion that

serves a dual purpose. A class defines the blueprint for the construction of objects with the same type. Conversely, objects that have the same type belong to the same class.

2.3.2 Polymorphism and inheritance

One of the most powerful concepts of object modelling is the notion of polymorphism. Polymorphism is a notion with a dual meaning. Polymorphism is the ability of an object to have multiple forms, depending on its context. Alternatively, polymorphism is the ability of objects instantiated from different templates to comply with the same type in a certain context.

To explain how object models can benefit from polymorphism, first we explain the notion of sub-typing and inheritance. Corresponding to the concept of class and type of an object, the concept of *subtype* and *subclass* can be defined. A subtype is a predicate on an object with more stringent constraints than its super type. Similarly, a subclass is defined as the set of objects for which the subtype holds true. Consequently, a class C_2 is a subclass of C_1 if and only if C_2 is a subset of C_1. A subtype predicate dictates that the object is constructed using a template that includes the template used to constructed objects in its superclass. For example, consider O_1 that is instantiated from T_1 and O_2 that is instantiated from T_2, then the type of O_2 is a subtype of the type of O_1 if and only if T_2 includes T_1.

The concept of subtype and subclass, leads to a clear separation between an *instance* of a template and an *instantiation* of a template. An instantiation of a template is always an instance of a template. However, an instance of a template is not necessarily an instantiation of that template. This also demonstrates the close linkage between a class and a template. Objects either directly or indirectly instantiated from the same template belong to the same class. Figure 2-4 shows the relation between instantiation, instance, class and object [Ru93].

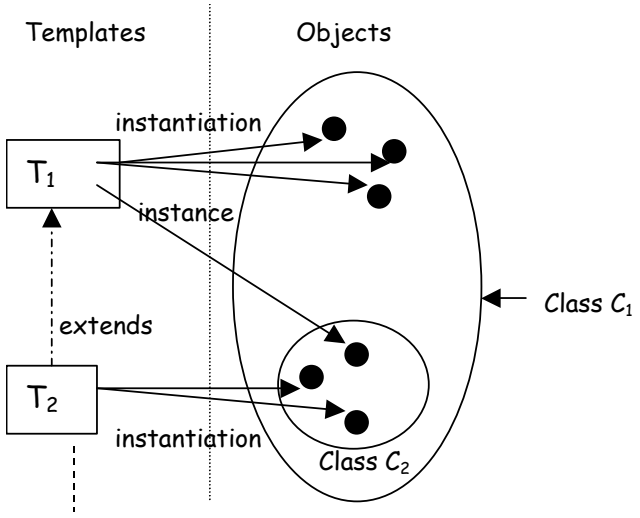


Figure 2-4
Template, class and object

Figure 2-4 illustrates that class C_2 is a subset of the class C_1. As a result, objects of type T_2 are also objects of type T_1. This is the consequence of the subtype relation between objects in C_2 and C_1, which follows from the extension of T_1 in T_2.

A template can be extended in two ways: through *structural sub-typing* and through *inheritance*. Structural sub-typing concerns the definition of a sub-type T2 of T1 by repeating template T1. Sub-typing through inheritance concerns the definition of sub-type T2 by referring to template T1. This form of sub-typing is preferred in this thesis.

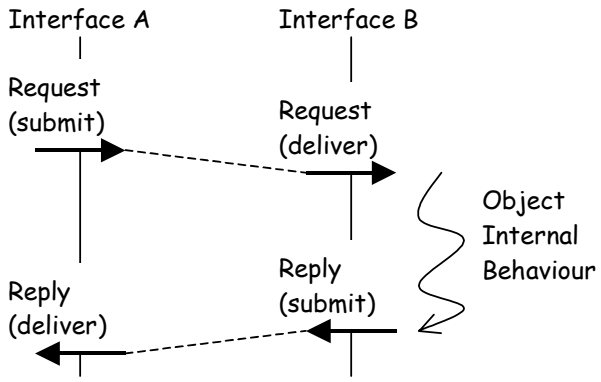
The concept of inheritance enables a designer to create new types by extending already existing types. As a result a sub-class of an already existing class of objects can be created. Objects of a subclass have an extended interface and are also instances of their superclass. In that sense objects for which a subtype T_2 of T_1 holds are polymorph, i.e., these objects can be addressed as instances of their super class. Polymorphism is the ability of an object to be an element of one class and at the same time of another class.

Polymorphism concerns the outside of an object, which is defined by the interface of an object. Polymorph objects are defined by creating a subtype of an interface. This is called *interface inheritance*. The specification of an object core may also be inherited from the specification of an object for which a super type holds. This is called *implementation inheritance*. Polymorph objects must be defined using interface inheritance but are not required to use implementation inheritance. When implementation inheritance is not used the object core *overrides* the behaviour of its super type.

2.3.3 Object interactions

When a distributed system is modelled as a set of objects, the collaborations between these objects are specified as *interactions*. An interaction is a sequence of actions that occur at the interfaces of the involved objects. An interaction between objects A and B is initiated through a request submitted at the interface of object A that is directed towards the interface of object B. As a result, a request action occurs at the interface of object B. This action triggers the behaviour of object B and may result in a reply action. The reply submitted at interface B results in a reply delivered at interface A. The primitive actions that constitute an interaction are shown in Figure 2-5.

Figure 2-5 Decomposition of an interaction



An interaction can be unidirectional or bi-directional. A unidirectional interaction consists of request actions (e.g., a request submitted at interface A and a request delivered at interface B). For a bi-directional interaction reply actions follow the request actions (e.g., a reply submitted at interface B and reply delivered at interface A). In both cases the delivery of a request triggers the behaviour of the object to which the request is delivered. Figure 2-5 shows the sequence of actions involved in a bi-directional interaction. An interaction is also known as an *invocation* of (an interface of) an object.

An object involved in an invocation can have two roles: it can either initiate or respond to an invocation. In case an object responds to an invocation, the object plays the server role. If an object initiates an invocation it plays the client role. The role of an object may change over time, when it is involved in different interactions. For example, an object playing the server role may change to a client role after it has received an invocation.

The signature of the interface defines the set of allowed invocations. An operation defines the structure of the data that can be exchanged in a potential interaction between a client and a server object. The structure of the data that is exchanged during an interaction is defined in the *operation signature*. A signature of an operation consists of the name of the operation (i.e. an identifier), a set of parameters and for each parameter its type and causality. The causality of a parameter determines whether a parameter is passed from a client to a server, from a server to a client, or in both directions.

Figure 2-6 shows the interface signature of an object. The signature consists of operations Op_1, Op_2 and Op_3. The first operation has two parameters of type t_1 and t_2. The second operation has one parameter of type t_3 and the third operation has no parameters.

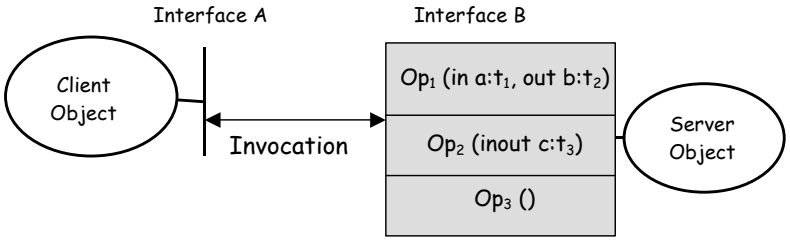


Figure 2-6 Client-server interaction

A client and server object may be running on the same computing system, but they may also be running on different computing systems. In the latter case some means of communication is needed to ensure that a request submitted by a client is delivered to the server object.

In both situations however a reply is delivered some time after a request has been submitted. When client and server objects are located on different computing systems this delay may be significantly larger than when they are located on the same computing system. A client may perform additional (inter)actions after it has submitted a request and before it receives the reply. This is called an *asynchronous* interaction. When a client waits until it has received a reply, the interaction is called *synchronous*.

2.3.4 The benefits of objects

The use of objects for modelling systems, offers a number of benefits to the designer. Object technology is designed to offer a high degree of modularity, extensibility and re-usability [Me88]. Each of these benefits is briefly reviewed, in order to demonstrate the benefits of employing objects as primitive modelling concepts.

An object encapsulates its state and behaviour and is an abstraction of part of a system. The state of an object can only be changed through an interaction at the interface of the object. Therefore, objects are the *modular* building blocks with which a system can be designed and built. The modularity that results from the use of objects ensures that large and complex systems can be composed from smaller objects.

The behaviour of an object is initiated by the invocation of an operation. How an object reacts to an invocation depends on its current state and the behaviour specification of the object core. Two objects, instantiated from different classes, can have the same behaviour specification for a method. This is enabled through polymorphism and implementation inheritance. Consequently, the interface and behaviour specification of an object are *re-usable*.

A subclass can be defined through interface inheritance. Inheritance is a way to extend the functionality of an object by adding extra operations, attributes or both. Overriding the methods of a super class can change the behaviour specification of the subclass. This is another way of introducing polymorphism into a design. Through interface inheritance and polymorphism, object models that already exist become *extensible*.

The main benefits of objects are modularity, re-usability and extensibility. Modularity is achieved through encapsulation, abstraction and composition. Re-usability is achieved through polymorphism. Extensibility is accomplished through inheritance and polymorphism. These characteristics of object models give the designer a large degree of flexibility and freedom. The designer is free to re-use already existing and proven object models and has the flexibility to extend and change the behaviour of the objects in these models when needed. This *late binding* between interface specification and behaviour specification introduces a large degree of flexibility.

2.4 Viewpoints

A designer of a distributed system has to take into account many characteristics of a distributed system, such as geographical distribution, lack of global state, etc. The development and manipulation of a design of a distributed system is a complex task. Such a task becomes even more complex when the design concerns an open distributed system, because the designer then also has to take into account the rules that guarantee the interoperability between parts of the system.

Abstraction and refinement are important means to manage complexity. Modelling system parts as objects offers a number of benefits to a system designer and helps to divide and conquer the complexity of a design. An

additional design principle is needed that enables a designer to deal with the various aspects of a distributed system in different models. This is supported by the notion of viewpoints.

Definition 4
Viewpoint

A viewpoint on a distributed system results in an abstraction that is achieved using a selected set of concepts in order to focus on relevant concerns within that system.

The need for viewpoints arises from the multiple perspectives from which a design is often developed and manipulated. These perspectives originate from the roles that a designer can assume during the design process. The need for viewpoints becomes even more apparent when a team of designers is considered, where each member of the design team is responsible for different aspects of a design. As a result, a model for each perspective of the system is developed. Viewpoints are also used in other disciplines. For example, consider the technical drawings of a three dimensional object in which one drawing shows a top perspective and another drawing shows a side perspective. Both drawings concern the same object, but model the three dimensional object from different perspectives, because a three dimensional object cannot be represented on a two dimensional piece of paper with sufficient detail for its construction.

A design of a system developed from a particular viewpoint is in an abstraction of the system. Another design of the same system developed from another viewpoint offers yet another abstraction. Both abstractions must be related, but they may not be ordered in the sense that one design is a refinement of the other. Viewpoints can be related by a partial overlap between the set of concepts used to create a design from one viewpoint with the set of concepts used to create a design from another viewpoint. Another way to relate viewpoints is to established consistency rules between some of the concepts used in each viewpoint. These consistency rules are expressed as correspondence relations.

The RM-ODP [ODP2] defines five viewpoints: enterprise, information, computational, engineering and technology viewpoint. We adopt the *computational* and *engineering* viewpoint for this thesis. We introduce the *deployment* viewpoint that is concerned with the deployment of software components in a run-time environment. An overview and motivation for the use of these three viewpoints follows below.

2.4.1 Computational viewpoint

The computational viewpoint considers the logical partitioning of applications into a set of interacting objects. The partitioning is logical in the sense that applications are structured independent of the computing systems on which their interacting objects run.

The computational viewpoint specifies the individual, logical parts, which collaborate with each other through interactions. The computational parts are also referred to as *computational objects*. Computational objects can interact with each other after a *binding* between their interfaces has been established. Figure 2-7 shows an example of an application that is partitioned into a set of computational objects. The bindings between these computational entities are depicted as lines between the interfaces of the computational objects.

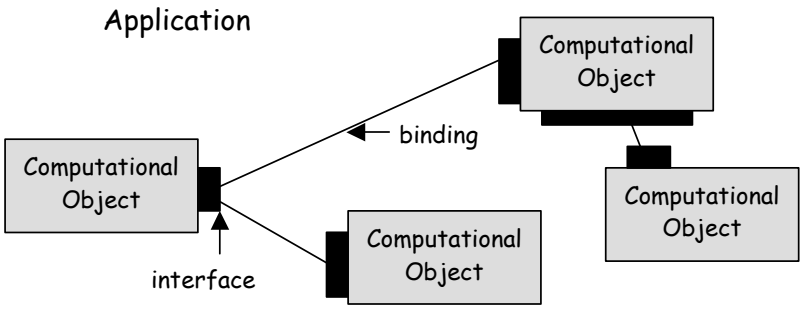


Figure 2-7 Computational view on an application

A binding between two computational objects must be established before they can interact. When the binding establishment is modelled explicitly, in terms of the actions that results in a binding we talk about *explicit binding*. Otherwise, a designer may omit the binding establishment from a design and assume that a binding is established upon the first interaction between two objects. This is called *implicit binding.* In certain cases, the properties of the binding must be observed or modified during the lifetime of the binding. When this is the case the control over the binding is modelled as a *binding object*. Usually a binding object is the result of an explicit binding, but this is not always the case.

The computational viewpoint on a distributed system can be organised as a set of specifications, where each specification deals with a different level of abstraction. Through composition and decomposition, a computational specification can be organised into a hierarchy. A computational object that is specified at a higher level of abstraction can be decomposed into a set of computational objects not represented in a specification at a lower level of abstraction. In a similar way, complex computational objects can be composed from a set of less complex computational objects.

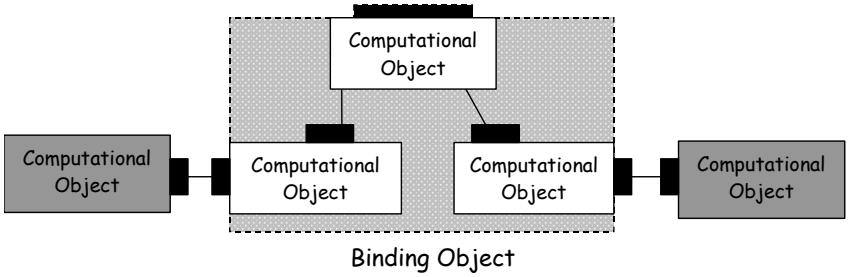


Figure 2-8 Example decomposition of a binding object

Figure 2-8 shows an example of a decomposed binding object. The binding object is decomposed into three computational objects. These objects use direct bindings to interact.

2.4.2 Distribution transparencies

Distribution transparencies are used to *hide* aspects of open distributed systems that arise from the physical and logical distribution of functionality across the resources of the system. To simplify the design task of distributed applications, a distributed system should offer an infrastructure that supports a set of distribution transparencies. A designer selects the transparencies that must be provided by the infrastructure. Aspects of distribution that are not covered by the infrastructure should be handled by application designers themselves.

Application designers can abstract from the mechanisms necessary to deal with the different aspects of distribution and can therefore focus on the application design. During application design only the required transparencies must be expressed and no design effort is needed for the realisation of these transparencies. The following definition is used in this thesis:

Definition 5 Distribution transparency

A distribution transparency is a property offered by a distributed system, to hide one or more of the characteristics of the system caused by the distribution of resources, with the purpose to simplify the task of distributed application design.

A list of distribution transparencies is found in [ODP2]. The following constitutes a non-exhaustive list of distribution transparencies:

– *access transparency*, which masks differences in data representation and invocation mechanisms that enables interworking between objects. This transparency hides many of the problems of interworking between heterogeneous systems, and is generally provided by most object middleware platforms.

- *failure transparency*, which masks from an object the failure and possible recovery of other objects (or itself) to enable fault tolerance. When this transparency is provided, the designer can assume an idealized world in which the class of failures hidden by this transparency does not occur.
- *location transparency*, which masks the use of information about *location in space* when identifying and binding to interfaces. This transparency allows objects to refer to each other by logical names, independent of their actual physical location.
- *migration transparency*, which masks from an object the ability of a system to change the location of that object. Migration is often used to achieve load balancing and reduce latency.
- *relocation transparency*, which masks relocation of an object from other objects bound to it. Relocation allows system operation to continue even when migration or replacement of some objects creates temporary inconsistencies in the view seen by other objects.
- *replication transparency*, which masks the use of a group of mutually behaviourally compatible objects to service a single interface. Replication is often used to enhance performance and availability.
- *persistence transparency*, which masks from an object the deactivation and reactivation of other objects (or itself). Deactivation and reactivation are often used to maintain the persistence of an object when the system is unable to keep processing, storage and communication resources continuously allocated.

In each case, the use of the transparencies by an application designer involves the definition of a set of transparency requirements. The set of requirements states where the transparency is needed (i.e., which interactions it affects). Transparencies may apply throughout a system, or only to some specific interfaces. For example, a designer can indicate the objects and interfaces to be supported by replication.

The solution that realises a transparency is not the responsibility of the application designer, but must be provided by the infrastructure. That solution takes the form of a set of rules for transforming the specification of a requested transparency into a specification in which selected interactions or objects are expanded to include mechanisms that provide that transparency.

2.4.3 Engineering viewpoint

The engineering viewpoint considers the infrastructure that is needed to offer distribution transparencies to computational objects. This comprises support at the end systems as well as the support for the communication between end systems.

Whereas a computational design hides aspects of distribution and does not consider the physical distribution of computational objects, an engineering design considers the system parts that offer distribution transparency. A design from the engineering viewpoint reveals the functions and mechanisms needed to realise the support for the transparencies that a computational specification requires. These functions and mechanisms are modelled as objects and therefore referred to as *engineering objects.* An engineering specification describes the behaviour and collaborations of a set of engineering objects that realise a run-time environment for the execution of a physically distributed application. An engineering specification aims for an efficient support run-time environment.

RM-ODP defines a number of concepts that can be used to develop an engineering viewpoint model of a distributed system. In this thesis we use the concepts of engineering object, capsule and channel. Engineering objects are used to structure the functions and mechanisms revealed from the engineering viewpoint. Each engineering object is located in one single capsule. The capsule is a unit of failure, which means that failure of a capsule implies that all objects located in the capsule fail. A capsule represents an operating system process. In case an engineering object interacts with another object that is in a different capsule, a channel for communication is needed. A channel between engineering objects in different capsules represents the services offered by a communication protocol.

Figure 2-9 depicts the engineering concepts used in this thesis.

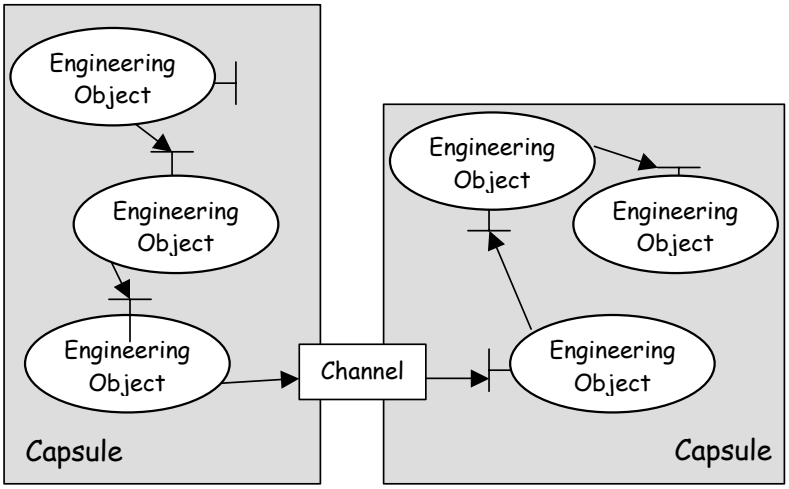


Figure 2-9 An engineering view of a distributed system

2.4.4 Deployment viewpoint

While the computational view defines the functional partitioning of an application and the engineering view reveals the infrastructure support and the geographical distribution of objects, none of these views considers the deployment of software on computing systems. Deployment concerns post-development activities such as configuring, installing, updating and even de-installing a software artefact [HHW97, RAC+01]. The deployment viewpoint is a meta-model that describes the concepts for constructing a deployment design. The main concepts of the deployment view are the unit of deployment and the deployment descriptor.

A class could be used as a deployable unit since it offers the 'building plan' or blueprint for creating an object. However, a class lacks context and the necessary information for deployment. The objects instantiated from one class may depend on objects instantiated from another class. In addition, a third party cannot configure a class independently. Therefore, a class is not a suitable unit of deployment.

A *component* is a unit of independent deployment [Sz97]. A component consists of one or more classes and optionally contains a set of initial interfaces, which are the starting point for interacting with the component. A deployment view reveals the distributed resources, i.e., computer systems and network elements, where components can be deployed. Components are deployed in a run-time environment, such as an operating system or a virtual machine, which offers basic functions for managing processing, storage and communication resources. A deployment specification shows the deployment characteristics of a component in a deployment descriptor. The deployment descriptor defines the policies that constrain the execution, the initial state of the component and the functions required from the run-time environment.

The concepts used in this thesis to create a deployment view model are component, run-time environment, node and deployment description. Figure 2-10 shows a component subject to a deployment description, deployed in a run-time environment at a node. We have decided not to use the term 'node' in the engineering viewpoint in this thesis, although it belongs to the RM-ODP engineering viewpoint, because it is considered a deployment aspect that can better be omitted from an engineering design. The engineering viewpoint does not reveal any deployable units; therefore it does not seem desirable to reveal nodes in the engineering view.

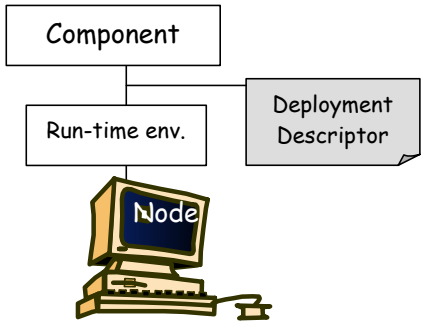


Figure 2-10 Concepts of the deployment viewpoint

2.4.5 Correspondence

Viewpoints are used to capture different aspects of the same system. The benefit of using viewpoints is that the complexity of a distributed system becomes more manageable. However, viewpoint specifications must be consistent in order to ensure a sound design without contradictions. Therefore the correspondence between the viewpoints must be defined precisely. A correspondence relation between the elements of viewpoint specifications allows the mutual consistency of these specifications to be checked.

Concerning the correspondence between the computational and engineering viewpoint, a relation exists between a computational object and one or more engineering objects. This means that the behaviour that occurs at the interface of a computational object corresponds to the behaviour that occurs at the interfaces of a set of engineering objects. An example of such a correspondence relation is shown in Figure 2-11. In this example, two computational objects correspond to two basic engineering objects. The set of engineering objects to which a computational object is mapped is called a *basic* engineering object. Other engineering objects realise the distribution transparencies that a basic engineering object requires. Transparency requirements originate from the computational specification. For example, location transparency for the computational objects is realised by a number of engineering objects and a channel.

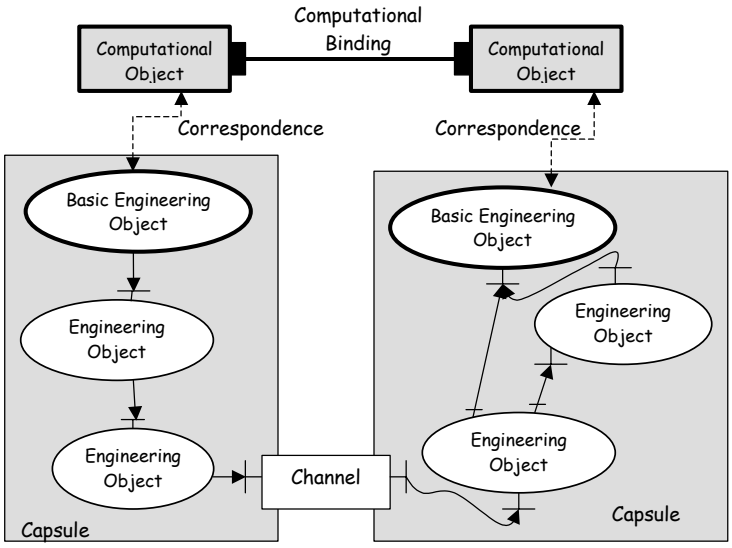


Figure 2-11 Example of a correspondence between viewpoint specifications

The correspondence relation allows a designer to check if the behaviour of the computational objects corresponds to the engineering objects to which these computational objects correspond.

Concerning the correspondence between the engineering and the deployment view, a relation must be defined between a set of engineering concepts (classes, objects and channels) and a component. An example of such a correspondence is shown in Figure 2-12.

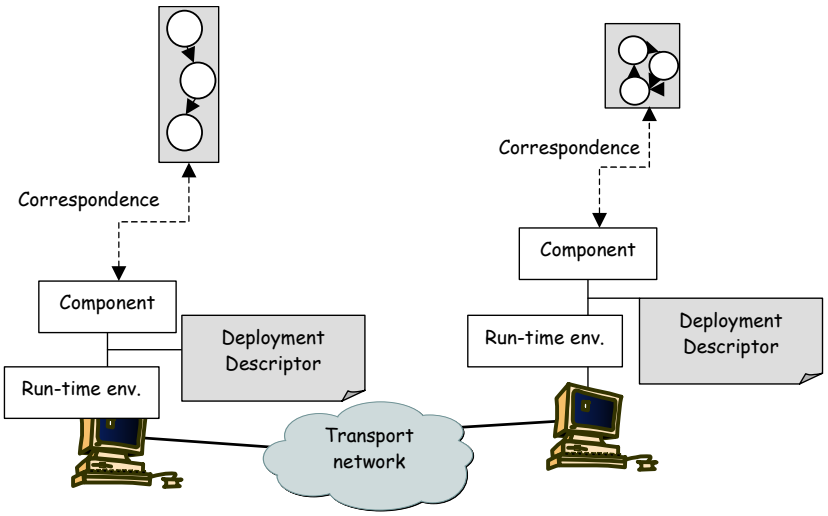


Figure 2-12 Another correspondence example

2.4.6 Viewpoint usage

Each viewpoint is related to a set of modelling concepts. A viewpoint design is an instance of the modelling concepts related to that viewpoint. The set of modelling concepts related to a viewpoint is a meta-model of that viewpoint. RM-ODP defines a viewpoint language for each viewpoint. The elements of a viewpoint language can be used to construct a viewpoint meta-model.

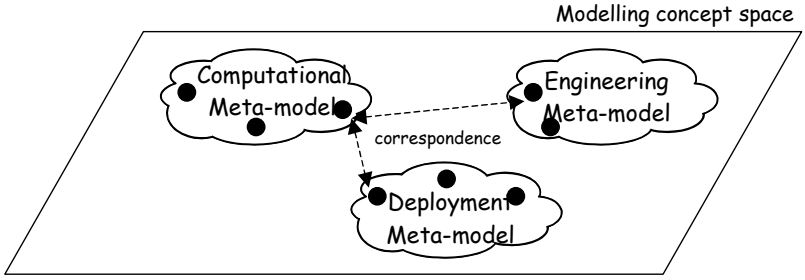
Some have adopted parts of the RM-ODP viewpoints, defined their own viewpoints and created a meta-model for each viewpoint [HKB01]. Usage of a meta-model to define the modelling concepts for a viewpoint, leads to a notion that describes a set of meta-models. This is defined by a concept space.

Definition 6 Concept space

A concept space is a set of meta-models that may be related through a correspondence relation between some of the modelling entities of these meta-models.

A graphical representation of the modelling concept space used in this thesis is shown in Figure 2-13. The black dots represent modelling concepts associated with each viewpoint. A correspondence relation can be defined between modelling entities that are in different meta-models.

Figure 2-13 An example of a modelling concept space



A computational design is an instance of the computational meta-model. In the same way the engineering meta-model is used to develop an engineering design. A deployment design is a design that is developed using the deployment meta-model. The computational meta-model used in this thesis is based on the computational viewpoint language. The engineering meta-model is based on the engineering viewpoint language, but introduces some modifications to the engineering language. The deployment meta-model is added to model components.

Viewpoints are used to separate the concerns of a designer and are useful because a designer can play different roles in the design of a distributed system. The viewpoints described in the previous sections can each be associated with a role that a designer can play. Three roles are distinguished: application designer, infrastructure designer and deployment designer. These roles can be fulfilled by three individuals, but may just as well be fulfilled by a single person.

The application designer is responsible for developing a computational design of a distributed system. A computational object of a design can be decomposed into several computational objects, resulting in a design at a lower level of abstraction. Repeating decomposition results in multiple levels of abstraction of a design. The application designer imposes a number of requirements on the distribution transparencies supported by the infrastructure.

The infrastructure designer is responsible for developing an engineering design of a distributed system. An engineering object of a design can also be decomposed into several engineering objects, resulting in a design at a lower level of abstraction. The engineering design defines what distribution transparencies are supported and how.

The deployment designer assembles computational classes into more coarse grained components. A component corresponds to one or more computational classes. The deployment designer also decides which of computational interfaces correspond to the external interfaces of a component. The deployment designer also derives the external interfaces of a run-time environment from the supporting infrastructure as defined in an engineering design. Components created by a deployment designer must be deployable on this run-time environment, therefore must be able to interact with the external interfaces of a run-time environment.

The flow of specifications for a typical development process and how they are used by each designer role is shown in Figure 2-14.

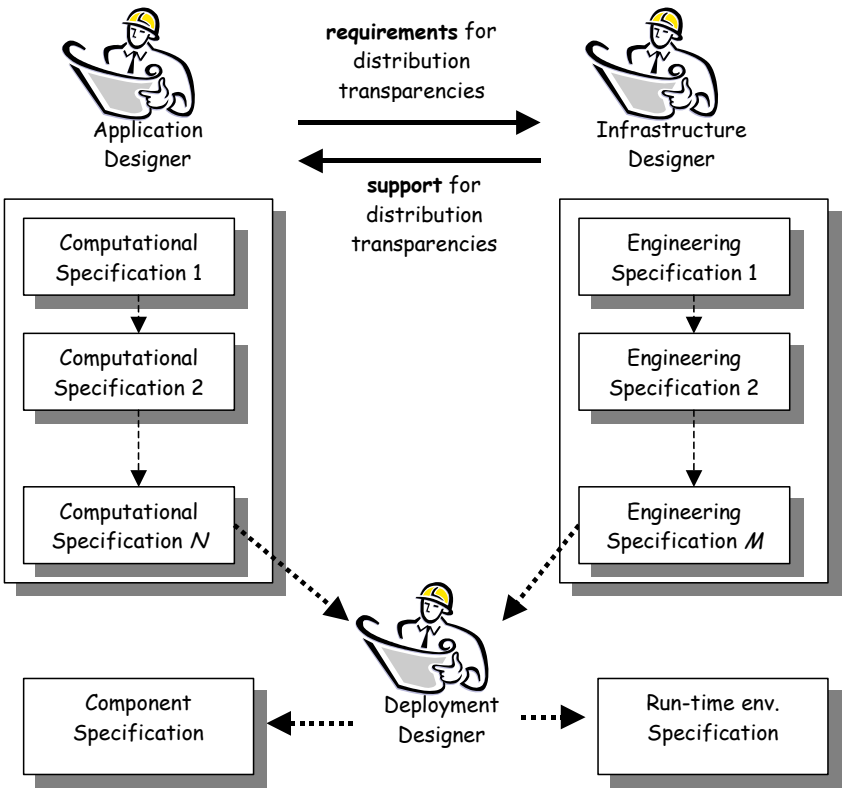


Figure 2-14 Usage of viewpoint specifications

The separation between the three roles as depicted in Figure 2-14 enables the development of a generic infrastructure that supports distribution transparencies that are applicable for many distributed applications. Infrastructure designers can specialise in the development of such an infrastructure and concentrate on optimising the infrastructure design. Application designers can focus on the logic of a distributed application without spending any effort on solving problems caused by distribution.

Eventually, models derived from the modelling concept space must be converted to software. Preferably this should be (partially) automated. Automated support for the conversion of models to software can be provided if transformation rules are defined from the entities in the meta-model to the target environment. This requires that a meta-model for the target environment is available or that it is constructed. The set of meta-models for possible target environments is called the implementation concept space. Figure 2-15 shows a set of models that is converted to software using transformation rules defined between the meta-models.

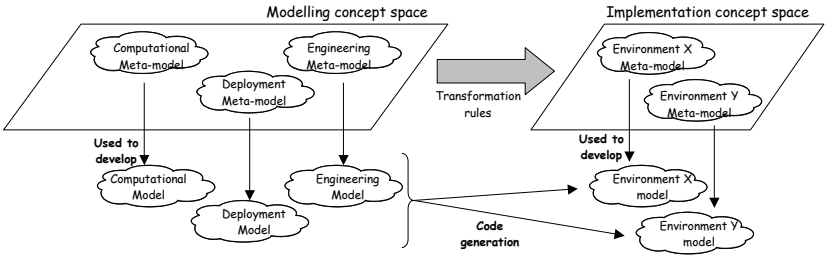


Figure 2-15 Automated code generation based on meta-model transformations

Code generation can easily be automated as code generators that implement the transformation rules between the modelling concept space and the implementation concept space. The feasibility of this approach has already been demonstrated [BoKa02]. A benefit of this approach to create software is that existing models can be transformed to software even when new target environments become available.

2.5 Middleware for distributed objects

The viewpoints and the designer roles described in the previous section entail a number of responsibilities in the overall design of a distributed system. The infrastructure designer is responsible for the design of the supporting infrastructure for application objects. The level of support that must be provided by the infrastructure is determined by the distribution transparencies that an application designer requires.

The application designer designs distributed object applications, i.e., distributed applications modelled as objects. The supporting infrastructure for these applications is denoted as a *middleware for distributed objects*. This leads to the following definition:

Definition 7 Middleware for distributed objects

Middleware for distributed objects is a software layer that provides distribution transparencies, with the purpose to support computational objects.

The middleware is not part of an application and is also not part of the system software, i.e., firmware or operating system, which runs on a computing system. In literature, the term middleware is an overloaded term, as it does not always concern a supporting infrastructure for distributed object applications. In this thesis, the focus is on middleware that simplifies the design of applications that consist of objects. Other types of middleware, such as transaction monitors and message-oriented middleware fall outside the scope of this thesis.

2.5.1 Positioning middleware

To position the middleware for distributed objects in the context of a distributed system, we propose a structure for a distributed system. The structure has been chosen in accordance with the viewpoints introduced in section 2.4.

Distributed systems consist of a conglomerate of hardware and software components that are designed to automate some business process or some end-user task. The automation of a process or task is achieved by the deployment of one or more software components in a processing environment. At this abstraction level, the *(distributed) applications* consist of one or more *application component*s that execute in a *distributed processing environment*. A deployment designer constructs the deployment specification using the computational specification and resource description of the distributed processing environment as input.

A refinement of the distributed processing environment (DPE) exposes the *middleware* and a *distributed resource platform*. The middleware consists of a number of software components. These software components are called *infrastructure components*. A deployment designer constructs the infrastructure components using the engineering specification and the resource description of the distributed resource platform as input.

The distributed resource platform (DRP) consists of various hardware systems, such as server systems, desktop computers, portable computers and personal handheld devices, and operating systems that belong to the hardware. The DRP is potentially heterogeneous because different vendors can produce the hardware systems and each hardware system has its own firmware and/or operating system. These hardware systems are interconnected by a *transport network*, which offers communication services. The resources offered by the DRP are represented by the *native communication and communication environment* (NCCE). Figure 2-16 shows how the elements of distributed systems are related.

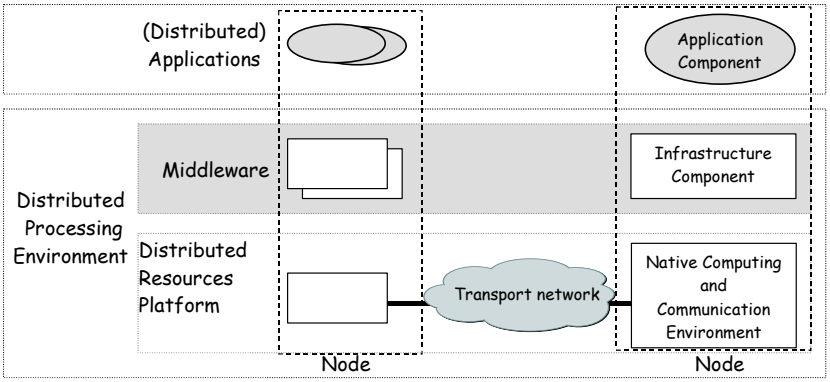


Figure 2-16 Overview of elements in distributed systems

The NCCE provides the run-time environment to the infrastructure components. Multiple instances of NCCEs and infrastructure components that are connected by a transport network constitute the DPE. The DPE provides the run-time environment to the application components.

2.5.2 Functions and mechanisms

The middleware can be characterised by the *functions* it offers. Functions are needed to support the interactions between computational objects. The capabilities of the middleware are described in terms of functions supported by the middleware. Functions are also referred to as *infrastructure services*. Functions can be fundamental, i.e., providing a minimal required subset of functionality, or widely applicable to the construction of a middleware, i.e., providing functionality that is convenient for many applications.

The functions supported by the middleware are exposed in an engineering viewpoint of the system. However, some functions may have transparency requirements and could therefore best be designed from a computational viewpoint. In this case a function becomes the user of other functions offered by the middleware. A designer of a function with transparency requirements has to switch to the role of application designer in order to deliver a computational specification.

The RM-ODP specifies a number of functions that can be used to construct parts of a middleware. The standard does not prescribe how these functions must be combined to construct a middleware neither does it specify the exact signature of the interfaces to objects that realise a function. Some of these functions are:

– Node management functions: control and manage the processing, storage and communication functions within a run time environment.

- Object management functions: concern the activation, deactivation, check pointing and recovery of objects.
- Event notification functions: record events and manage event histories for future reference.
- Replication functions: ensure that a group of objects appear as a single object.
- Transaction functions: coordinate and control a set of transactions.
- Trading functions: mediate advertisement and discovery of interfaces.
- Repository functions: store data, type information and meta-data.
- Security functions: manage security aspects such as non-repudiation, integrity and confidentiality.

The functions describe the middleware capabilities, but not how these capabilities are realised.

An infrastructure designer uses *mechanisms* to refine the specification of a function. The Oxford Dictionary defines a mechanism as "the machinery by means of which some particular effect is produced". However, the Concise English Dictionary gives a more suitable definition:

Definition 8
Mechanism

A mechanism is a system of correlated parts, working reciprocally together, as a machine.

A node management function, for example, can be realised by a set of engineering objects. The engineering objects work together to provide a basic engineering object with control and management of processing, storage and communication resources.

2.5.3 Benefits

Studies have shown that the use of middleware to support distributed applications offers several benefits. The following benefits have been identified [BHK+96]:

- *Interoperability* is the ability of two or more objects to communicate and co-operate despite differences in their implementation language and execution environment. Interoperability allows one object to use the service of another object existing in some system, without knowing the physical details of the other system. Middleware enables interoperability by hiding the mechanisms used to communicate between disperse computing nodes.
- *Portability* allows the re-use of components on various computing nodes. Binary portability means that components can execute on computing nodes with different hardware architectures. Source code portability means that the source code of a component can be compiled to native

machine code for various computing nodes, without changing the code. A middleware enables both forms of portability by encapsulating system specific interfaces in objects with a standardised interface.

- *Coexistence* means that old and new components of the system can co-exist. Parts of the middleware and applications, i.e., the infrastructure components and application components, can be replaced with new components. This is enabled through abstraction, interface sub typing and polymorphism.
- *Reliability* and *availability* of distributed applications can be enhanced by the middleware. Critical application components can be replicated on multiple computing nodes. The middleware can contain the mechanisms needed to manage the number of replicated objects and to synchronise the state of replicas.
- *Extensibility* is the property that functionality can be added on demand. Adding new mechanisms can extend the functionality of the middleware and therefore offer support for additional transparencies to application designers. This is enabled through inheritance and polymorphism.
- *Configurability* is the property to configure and reconfigure the application components on-line. The middleware can provide support to relocate objects to different computing systems and thus provide the means to change the configuration of a distributed application.
- *Implementation language independence* is the ability of programmers to implement application components in any suitable language they choose. The middleware can support multiple languages, because the infrastructure services are offered through interfaces that can be mapped to several implementation languages.

2.6 QoS aware middleware

End users of a distributed application have a perception of the quality of that application. For example, a user may perceive the responsiveness of a application as quick or slow. An end user may emphasise different quality characteristics of the application. The quality that an end user perceives is to a large extent determined by how many processing, storage and communication resources have been assigned to the application components that constitute the distributed application. Quality of Service (QoS) concerns the quality characteristics that an end user perceives of an application.

Currently, most commercially available infrastructure components are still limited to the support of best-effort QoS to applications. This means that these infrastructure components attempt to assign resources to components as much as possible, but without any commitment that

sufficient resources will be available all the time. Best-effort QoS constitutes an obstacle to the use of middleware systems in QoS critical applications, or in case services are offered in the scope of Service Level Agreements with strict QoS constraints.

This section introduces an initial set of concepts that is used for modelling a QoS aware middleware and identifies the role of a QoS aware middleware in the support of distributed applications.

2.6.1 Quality of Service

The notion of QoS is broad and is applied to many areas, such as end-user quality perception, ergonomic quality of user interfaces, network performance, system performance. Several generic definitions of QoS have been provided, with the purpose to cover the many areas to which QoS is applied.

Some of these generic definitions are:

- QoS is user-perceived performance or service as experienced by the user [Fr96];
- QoS is a set of qualities related to the collective behaviour of one or more objects [ISO X.641];
- Quality: the totality of features and characteristics of a product or services that bear on its ability to satisfy stated or implied needs [ISO8402];
- QoS is a set of user-perceivable attributes, which describe a service the way it is perceived. It is expressed in a user-understandable language and manifests itself as a number of parameters, all of which have either subjective or objective values. Objective values are defined and measured in terms of parameters appropriate to the particular service concerned, and which are customer-verifiable. Subjective values are defined and estimated by the provider in terms of the opinion of the customers of the service, collected by means of user surveys [Me91, Mej92];
- QoS is the degree of conformance of the service delivered to a user by a provider with an agreement between them [P806]

A common property of these QoS definitions is the user perception of the quality characteristics of a service. However, user perception is influenced by many subjective parameters, which are outside the control of an application or infrastructure designer. QoS aspects such as user needs, customer satisfaction or price/quality ratios are not considered in this thesis, because they cannot be controlled by an application or infrastructure designer.

The focus in this thesis is on the application QoS requirements and how these requirements can be supported by the middleware. We develop facilities to make middleware QoS aware. A middleware system provides distribution transparencies to applications that are active in a large heterogeneous distributed system.

There is a wide variety of literature written on QoS and many authors of recent work capture QoS into a framework [SeCa00], [BeGe97], [NWX00]. Most QoS definitions involve the user perceived quality. In the case of a distributed application deployed on an object middleware platform, the user perceived quality is directly influenced by the quality characteristics of the applications objects. Application objects are modelled as computational objects and include binding objects. Therefore, we focus on the QoS of a computational object and use the following definition:

Definition 9 Quality of Service of a distributed application

The QoS of a distributed application is characterised by collection of values (e.g., a ratio, a maximum, an average, a variance, a probability distribution) acting on the properties (e.g., a loss, a delay, a failure rate, availability) of its computational objects.

From this definition it follows that *QoS requirements* are the requirements that an application designer imposes on a collection of measures that act on a set of properties of computational objects. QoS requirements can be imposed on client, server and binding objects.

2.6.2 QoS support for application objects

During the design of a distributed application, the client and server interfaces of the application objects are specified. In principle, this specification should define the attributes and operations of these interfaces. Some have referred to this approach as 'design by contract' [Me92]. A contract aims to specify the service provided by an interface in a precise way. Contracts are divided into four different levels [BJPW99]:

- Syntactical contracts
- Behavioural contracts
- Synchronisation contracts
- QoS contracts

When considering QoS aware middleware, we suppose that the interface specifications are extended with QoS contracts that can be associated with the whole interface or with individual operations and attributes. In the case of a client interface, these statements describe the *required QoS*, while for a server interface these statements describe the *offered QoS*.

Objects life cycle

After the objects of a distributed application have been implemented and assembled into components, the application components are deployed. We consider the general case in which persistent objects and late binding is supported by the middleware. In this case, an object has the following life cycle:

1. Object creation, in which interface references for the server interfaces of an object are created and can be referred to by other objects;
2. Object activation, in which an object starts execution, which implies that all local resources necessary for the object to execute should be properly allocated;
3. Object deactivation, in which local resources allocated to an object may be released, although the interface references may still be valid in case persistent objects are supported;
4. Object destruction, in which the object is deactivated (if it is still active) and its interface references become invalid.

At object creation time, an initial offered QoS is established as specified during design time. This is the QoS that a server object intends to offer if there are sufficient resources available at the time a client binds to it. A QoS aware middleware can use object activation to refine the offered QoS, by restricting the ranges originally described for the offered QoS at design time. The run-time status of the middleware and the distributed resources platform should make it possible to determine this offered QoS more precisely.

Explicit binding

Object interfaces have to be bound to each other in order to allow these objects to interact through the middleware. In some cases, this binding happens implicitly when the client object issues a request (implicit binding).

For QoS aware middleware platforms, however, implicit binding is not desirable, since the QoS requirements may demand that resource allocation procedures are performed just before the request is executed. Unfortunately, we cannot predict the speed and reliability of these procedures. In the worst case, we may still have to activate the server object. This means that we cannot always guarantee the QoS requirements by using implicit binding. Therefore, in QoS aware designs *explicit binding* is necessary, which consists of taking explicit actions at the computational level in order to establish the binding before interacting.

When explicit binding is used, the client object requests the establishment of the binding, giving to the middleware a reference to a server interface. This request also contains the required QoS, which could

be retrieved from a QoS specification repository. The middleware platform then locates the server object. In case the server object has not been activated, the middleware platform activates this object and continues the establishment procedure. After the middleware is sure that the object is active, it compares the offered QoS with the required QoS and uses its internal information to determine an agreed QoS. This process is called *QoS negotiation*. Negotiation is only successful if the agreed QoS falls within the ranges prescribed by the client object in the required QoS. In case the binding establishment has been successful, the client and server objects are informed that a binding has been built. From this moment on these objects can interact through the binding. Figure 2-17 shows the establishment of a binding using a QoS aware middleware.

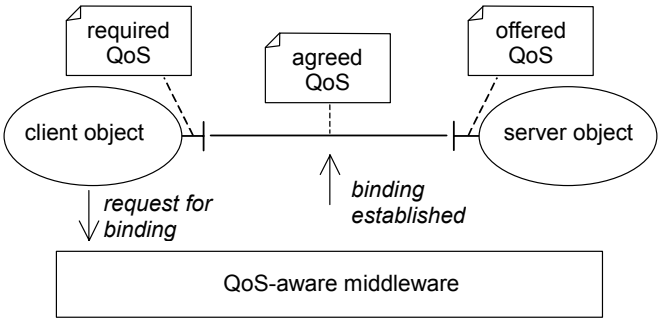


Figure 2-17 Binding establishment using a QoS aware middleware

The agreed QoS is determined by considering the required QoS on one hand, and the composite QoS capabilities of the server object (the offered QoS) and the middleware on the other hand. The agreed QoS serves as a *contract* between the application objects and the middleware platform, which should be respected during the operational phase when the objects interact through the binding.

The binding establishment may also result in the creation of a binding object. This object binds the client object and the server object, and offers a control interface that allows, for example, the inspection and modification of the agreed QoS.

Our approach considers that a binding has been successfully established and that the agreed QoS has to be maintained. The QoS aware middleware is responsible for that, and is constantly adjusting its internal characteristics and the usage of computing and communication resources in order to achieve it.

2.7 Conclusions

This chapter introduces the notion of a distributed system and outlines a number of general characteristics of distributed systems. The design of a distributed application for a potentially heterogeneous distributed system is a complex task. Therefore, design concepts and principles that are powerful enough to express various aspects of a distributed system are necessary to perform this task. These concepts are defined in this chapter.

The modelling concepts presented in this chapter are derived and adapted from the RM-ODP standards. A key design principle of RM-ODP is the use of distribution transparencies. Distribution transparencies enable a distributed application designer to hide distribution aspects in a selective manner. Through the use of viewpoints on a distributed system, an application designer can focus on certain aspects of a distributed system, while abstracting from other aspects. Three viewpoints are used in this thesis: the computational, engineering and deployment viewpoint. A computational viewpoint design selectively hides aspects of distribution. The engineering viewpoint is concerned with resolving the symptoms of a distributed system, such as remoteness, heterogeneity and autonomy. The deployment viewpoint concerns the physical hardware and software components that constitute a distributed system at run time.

The notion of 'object' is used as an elementary unit of specification for all viewpoints. The benefits of using objects to design distributed systems have been identified in this chapter.

Finally, the concept of middleware for distributed objects is defined. This middleware is a software layer that offers a set of functions to application designers, which can be used to support a distributed application. In case of a QoS aware middleware, some of the functions enable the establishment of a binding between objects that is subject to an agreed QoS. The infrastructure designer is responsible to design the functions offered by the QoS aware middleware.

Chapter 3

Overview of the research area

This chapter situates the area of research to which this thesis contributes.

One of the aims of this thesis is to advance the technological developments of object middleware platforms through the addition of facilities that can control the qualitative aspects of the objects deployed on the middleware. Through these facilities future object middleware platforms can offer support for Quality of Service (QoS). This chapter identifies and discusses the technological and scientific developments that contribute to the advancement of distributed processing environments.

Observations presented in this chapter are used in subsequent chapters to derive requirements, models and solutions for a QoS aware distributed processing environment. Ideally, the models and solutions for a QoS aware distributed processing environment should be in line with the already existing standards, architectures and technologies for such environments. Such an alignment identifies the areas of improvement and increases the acceptance of the models and solutions proposed.

This chapter is organised according to the subjects identified in section 3.1. Sections 3.2 to 3.5 present the main issues and developments in the areas of object middleware architectures, network technology, QoS architectures and software engineering technologies, respectively. Section 3.6 presents related work that is also concerned with the architecture and design of a QoS aware DPE. Section 3.7 identifies requirements and opportunities for advancement of distributed processing environments.

3.1 High-level overview

The parts that constitute a distributed processing environment (DPE) can be obtained from various vendors or open source communities. As a result, a DPE consists of a potentially heterogeneous set of hardware and software components.

Business organisations take independent decisions from which sources their hardware and software components are obtained. Distributed applications often cross the boundaries of one or more business organisations and therefore require information processing and communication systems to interwork. To facilitate the interoperability among these systems, standards have to be established that manufacturers must use to build information processing and communication systems. Several standardisation organisations and industrial consortia have recognized the need for interoperability standards. The standardisation organisations of concern to this thesis are IETF, W3C, ISO-ITU, SUN JCP and OMG.

Developments in the architectures and technologies produced by research communities and standardisation organisations have an impact on the structure of a DPE. Research efforts aimed to advance the one or more parts or aspects of a DPE, should take into account these developments.

The design of a QoS aware distributed system, developed in chapter 6, takes into account the developments of several architectures and technologies. Current object middleware architectures are considered for an engineering viewpoint design of the distributed system. Current software engineering technologies are considered for a computational, engineering and deployment viewpoint design of the distributed system. Current QoS architectures are considered for the definition of QoS concepts and models that can capture the QoS aspects of the distributed system. Current network technologies are considered for the choice of the parts that constitute the distributed resource platform of the distributed system. The subjects of concern to this thesis are categorised into object middleware architectures, software engineering technologies, QoS architectures and network technologies.

Figure 3-1 shows the subjects and standardisation organisations that are covered in this chapter.

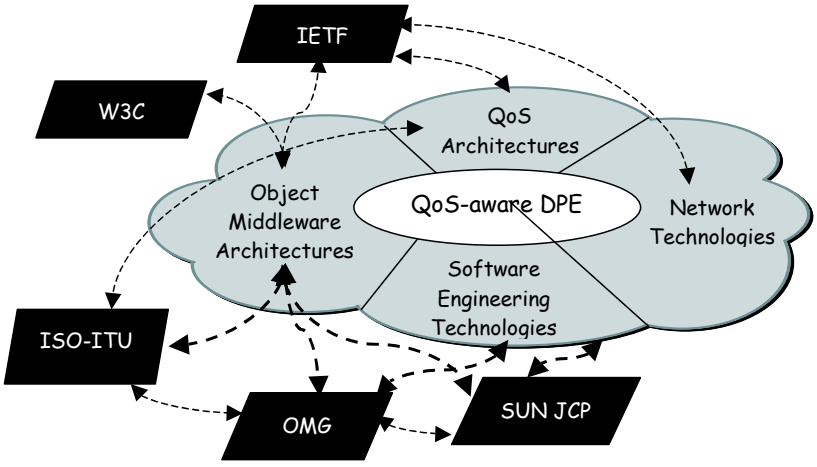


Figure 3-1 Subjects and standardisation organisations concerning a QoS aware DPE

The standardisation organisations and consortia depicted in Figure 3-1 have a number of common goals. They all influence the development of technologies that can be used to realise a distributed resource platform or a middleware layer. Most consortia produce reference implementations to ensure the feasibility of standards and to increase the widespread acceptance of these standards.

Standardisation is directed at building consensus on some technical issue. This requires that experts are gathered and that the ideas of these experts converge into a single specification. Every organisation has defined its own process and rules that must be followed in order to establish a standard, but each process is based on iterations and multiple negotiations to converge and build the consensus. Reference implementations are often required before a standard is accepted and a growing number of these implementations become available as open source.

3.2 Object middleware architectures

A middleware platform is an infrastructure that offers support to distributed applications running in heterogeneous distributed systems. This section reviews the origin and features of the architectures of three of today's most widespread used middleware platforms: CORBA, SOAP and Java 2 Enterprise Edition. These middleware platforms support interactions between object-based software components; hence they fall in the category of object middleware. Before the architectures of these middleware platforms are explored, the basic functions of object middleware are discussed.

3.2.1 Basic facilities of object middleware

Object middleware realises the transparencies that distributed object applications require. In that sense, object middleware is a supporting infrastructure that application designers assume to be present for the design of distributed object applications. Ideally, object middleware should support a computational design with facilities for interface specification, interface binding and the life cycle management of objects. These basic facilities and the parts of the computational model that they ideally should support are shown in Figure 3-2.

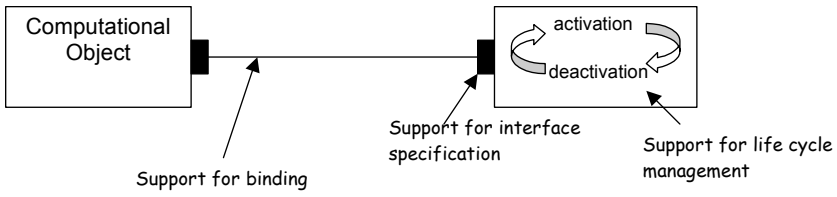


Figure 3-2 Ideal basic object middleware facilities

Object middleware platforms such as CORBA, SOAP and J2EE provide support for interface binding, interface specification and life cycle management. These object middleware platforms also define additional facilities. The standardisation of the functions defined for these platforms is guided and directed by several standardisation organisations.

3.2.2 CORBA

The Common Object Request Broker Architecture (CORBA) is developed and maintained by the Object Management Group (OMG). The OMG produces standards for object technology. The CORBA standards are directed by the guidelines found in the Object Management Architecture (OMA). The CORBA standards include interfaces specifications defined using the OMG Interface Definition Language (OMG IDL).

This section describes the basics of OMG IDL, gives an overview of the Object Request Broker, presents the OMA and discusses the standardisation organisations that have an impact on CORBA.

OMG IDL

The OMG IDL is a descriptive language, for the specification of the interface exposed by a server object. With IDL it is not possible to specify the behaviour of an object; it only allows the definition of the signature of an interface. The signature of an interface consists of an interface name, the operation names and for each operation a) the type of the request parameters, b) the type of the response parameters and c) the type of the exceptions that can be raised.

The mapping of IDL to most popular programming languages, such as C++, Java, COBOL, Smalltalk, Ada, Lisp and Python, has been standardised by the OMG. An IDL compiler uses an IDL specification to automatically generate a programming language specific interface according to the mapping rules defined for that programming language. Consequently, IDL is called programming language neutral, since mappings have been defined to nearly every programming language.

Object Request Broker

A CORBA server object gets requests from a client object through the Object Request Broker (ORB). The ORB is responsible to locate a server object to deliver a request. A CORBA server object is identified, located, and addressed by its object reference. Within the context of a CORBA operation invocation, the server object to which the request is sent is called the "target object" [ScVi97]. Clients can issue requests that are transferred by the ORB to the appropriate target object, which processes the request

and returns a response. The complicated task of how these messages are transferred is hidden from the application objects.

CORBA enables client objects to invoke a target object's methods regardless of whether the target object is in the same address space as the client or located in a different address space on a remote node. The ORB finds the target object for the request, prepares the target object to receive the request, and transports the request data. The client interface is independent of where the target object is located, in what programming language the target object is implemented, or any other aspect that is not reflected in the target object's interface. Figure 3-3 shows the structure of the ORB and interfaces between the ORB and application objects.

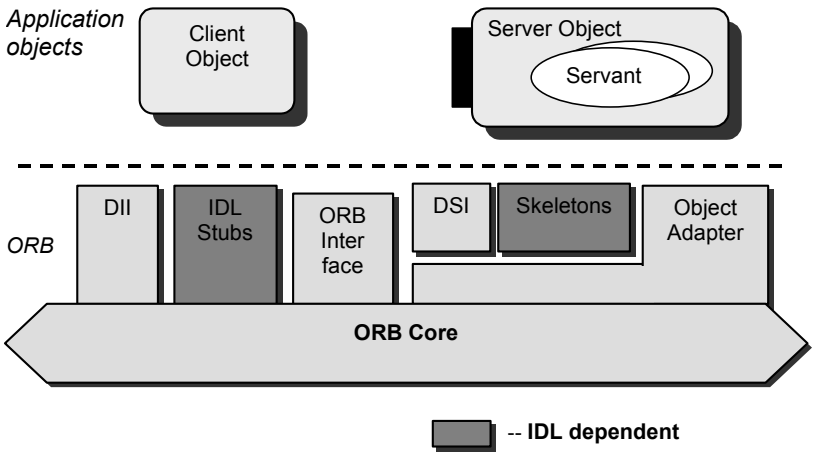


Figure 3-3 Parts of an ORB

The interface of a CORBA object can be defined statically, i.e., at design-time, or dynamically, i.e., at run-time. Interfaces are defined statically in an IDL specification. From this specification an IDL compiler can generate a programming language specific stub for client-side access and a programming language specific skeleton for server-side access to an object.

Alternatively, interfaces can be registered with an Interface Repository (IR) service, which stores the elements of the signature of an interface as objects, permitting access to these elements. Clients can use the IR to discover the signature of an interface at run-time and then use the Dynamic Invocation Interface (DII) to construct a request and then initiate a request/response sequence. The Dynamic Skeleton Interface (DSI) is the server-side equivalent of the DII. Server objects can use the DSI to dynamically define an interface signature at run-time and receive requests on that interface.

A servant is a programming language entity that implements one or more CORBA objects. In procedural languages like C and COBOL, a servant is a collection of functions that manipulate data (e.g., an instance of a struct or record) and represent the state of a CORBA object. In OO languages like C++ and Java, servants are object instances of a particular class. The Object Adapter (OA) binds the programming language concept of servants to the CORBA concept of objects.

The relationship between a CORBA object and a servant is like the relationship between virtual memory and physical memory in an operating system. Just as a virtual address space is a virtual entity that is bound to physical memory, a CORBA object is a virtual entity that is bound to a servant. A virtual memory location can be read and written by a computer program because of the work performed by the computer's memory management unit (MMU). The MMU maps virtual memory addresses into physical memory addresses and ensures that each valid virtual memory address is mapped to a physical memory storage location. Similarly, the ORB and the OA co-operate to allow client applications to invoke requests on CORBA objects and ensure that each valid CORBA object is mapped to a servant. In addition, the ORB and the OA co-operate to transparently locate and invoke the proper servants, using the addressing information stored in CORBA object references [ScVi97].

The Object Management Architecture

The Object Management Architecture (OMA) is composed of an Object Model and a Reference Model. The Object Model defines how objects distributed across heterogeneous environments can be described, while the Reference Model defines, in addition to the ORB, four categories of object specifications: application interfaces, domain CORBA facilities, CORBA services and horizontal CORBA facilities.

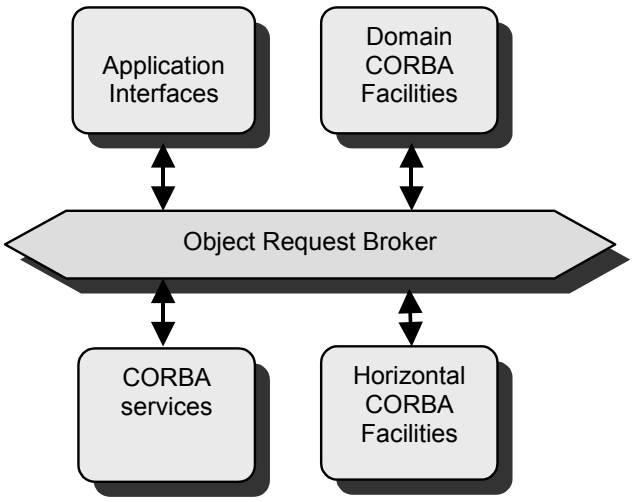


Figure 3-4 OMA object categories and the ORB

Figure 3-4 shows the ORB and object specification categories of the OMA reference model. Objects that interact through the ORB are classified into the following categories:

- CORBA services: domain-independent infrastructure services that offer basic functionality considered essential to support computational objects. An example of a CORBA service is the Naming Service, which allows clients to find objects based on a name.
- Domain CORBA facilities: business domain-specific infrastructure services that are for general-purpose use within a specific business domain. Business domains include healthcare, transportation, telecommunications and other industry groups that benefit from the OMG process to standardise domain specifications.
- Horizontal CORBA facilities: application services that cover computational aspects found in many distributed applications. Unlike the Domain CORBAfacilities these facilities are potentially useful across business domains. Examples of horizontal CORBAfacilities: the Printing Facility, the Secure Time Facility, the Internationalization Facility, and the Mobile Agent Facility.
- Application Interfaces: object specifications developed for specific applications. Since the OMG does not develop specification applications (only generic interface specifications), these interfaces are not standardised by the OMG.

The OMG specifications for CORBA facilities and CORBA services define the interfaces to objects in OMG IDL.

Standardisation process

The Object Management Group (OMG) is an open consortium that produces and maintains standards for interoperable applications. Among the best-known standards produced by the OMG are CORBA (including OMG IDL and IIOP), UML and XMI. The OMG was founded in April 1989, with the goal to create a marketplace for component-based software by furthering the introduction of standardised object software. The OMG has around 800 members including nearly every large company in the computer industry, many small companies, research institutes and universities.

The OMG is structured into three major bodies: the Platform Technology Committee (PTC), the Domain Technology Committee (DTC) and the Architecture Board. The PTC oversees the advancement of the ORB and CORBA services. The DTC oversees the development of the domain CORBA facilities. The Architectural Board manages the consistency and technical integrity of work produced in the PTC and DTC.

OMG members initiate the OMG standardisation process for some object specification by writing a Request for Proposal (RFP). Other members can respond to an RFP by writing a submission. The technical committees and the architecture board review these submissions in several iterations. During each iteration the submission is refined and finally, when consensus is reached, the submission is adopted as an OMG standard. Adopted specifications are only accepted as *formal OMG standards* if one or more member companies have a commercial implementation of the standard.

Through its liaison with other consortia and standardisation organisations some of the key OMG specifications have become internationally accepted. For example, the OMG IDL specification has also been accepted as an ISO standard.

The International Organization for Standardization (ISO) is a worldwide federation of national standards bodies from around 140 countries. ISO was established in 1947. The mission of ISO is to promote the development of standardization. The scope of ISO includes standards in virtually all areas, such as, for example, chemistry, photography, textiles and many other technical fields. ISO carries out standardisation in the field of information technology together with the International Electrotechnical Commission (IEC) in the joint ISO/IEC technical committee.

A national member body can initiate an ISO standardisation activity by proposing a new work item. Once the need for a new work item is established, a working group of technical experts defines the scope of the future standard. After that, the national member bodies negotiate the detailed specification of the standard and build consensus on what should be in the standard, resulting in a draft standard. In the final phase, a

specification becomes an ISO standard when two-thirds of the participants that produced the draft and 75% of the members that vote approve the standard.

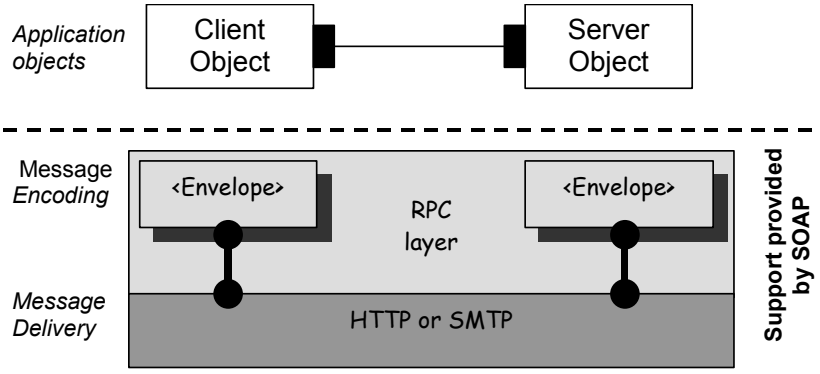
3.2.3 SOAP

In 1998 a few companies, such as DevelopMentor, IBM and Microsoft, initiated the development of Simple Object Access Protocol (SOAP). The SOAP specification version 1.1 is currently submitted to the Worldwide Web Consortium (W3C) and will be further developed and maintained by W3C.

SOAP is a protocol for the exchange of data in a distributed resource platform. The protocol has been designed to invoke functions on servers, services, components and objects. Although SOAP does not define an object model itself, it certainly is an important development in the area of object middleware because it offers a supporting infrastructure to application objects.

The SOAP specification can be divided into three parts: a description of the encoding for messages, a description of how to use messages in a remote procedure call (RPC) and procedures to exchange messages by HTTP or SMTP. These parts of the SOAP specification and their relationship are shown in Figure 3-5.

Figure 3-5 Parts of SOAP



This section describes the basics of the encoding rules and delivery of SOAP messages, what basic facilities have been omitted from SOAP in order to keep it simple and the standardisation organisation that furthers the development of SOAP.

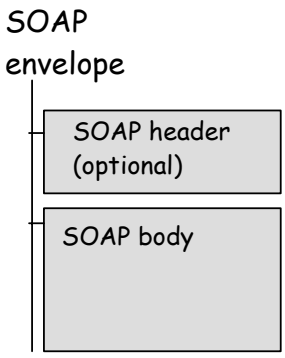
Message encoding and delivery

SOAP messages are encoded using the eXtensible Markup Language (XML). XML is a tag-based meta-language that enables a designer to define

data structures using tags. The SOAP specification builds on the W3C specifications for XML namespaces [Br99] and XML schema [Fa01]. SOAP messages must be structured according to the definition of a SOAP envelope, which is defined in an XML schema. Application data must be represented in a message using the SOAP serialisation rules.

The SOAP envelope consists of a header, which is optional and a mandatory body (see Figure 3-6). The header contains information that a recipient uses to determine if and how it can process a message. The body contains the topic or the payload of the message.

Figure 3-6 SOAP envelope structure



The encoding rules of a SOAP envelope are based on a simple type system. It has a number of primitive (or scalar) data types and allows more complex data types to be constructed from the primitive types.

SOAP messages are exchanged using a request-response style message distribution scheme. The RPC layer is responsible to deliver request messages to the server, which processes the request and returns a response message. The RPC layer can use several protocols for message delivery. Currently, the binding between the RPC layer and HTTP and the binding between the RPC layer and SMTP is standardised.

The simplicity of SOAP

SOAP offers a lean middleware for distributed objects. A number of issues have been omitted from the SOAP specification.

- First, SOAP does not have an object model or an interface definition language for expressing the methods of an object. As a result it becomes impossible for tools to check the design-time compatibility of a client and server. A SOAP server must include functions that check the compatibility of an incoming message and decide if the server is capable to process it.
- Second, the specification does not define a language mapping. This means that SOAP does not specify how a data structure, e.g., in Java,

should be mapped to a SOAP encoded body or how a SOAP message should be bound to a particular method of a Java object. The developer that implements a SOAP service establishes these relationships and must do its own administration for dispatching incoming messages to implementation objects. As a result, SOAP server objects are most likely not portable to other SOAP implementations.

– Third, the SOAP specification has no means for object activation or life-cycle management of objects. There is no standard interface for registering an implementation object with a SOAP server. This is a consequence of the two issues mentioned above. Lack of a language neutral interface specification language and the lack of a mapping of such a specification language to a programming language, inhibits the definition of a standard interface to register an implementation object with a SOAP server. The developer that implements a SOAP service must also develop code for object activation and life-cycle management. This too compromises the portability of SOAP server objects.

Standardisation process

The World Wide Web Consortium (W3C) was created in October 1994 to promote and advance the World Wide Web. The W3C develops common protocols that ensure interoperability between software systems that need to share information. Traditionally the activities of the W3C have been focussed on defining specifications for HTML [W3C98], the URL [Mo97] and HTTP. W3C has more than 500 member organizations from around the world and has earned international recognition for its contributions to the growth of the Web.

The W3C is organised in working groups and has defined a consensus driven process that leads to W3C standards. The result of a standardisation activity is a *W3C recommendation*. Such a recommendation goes through a number of phases before it becomes an accepted standard. As a complement to recommendations and standards, W3C releases open source software as a proof of concept.

3.2.4 Java 2 Enterprise Edition

The Java 2 Platform Enterprise Edition (J2EE) technology provides a component-based approach to the construction and deployment of distributed applications. J2EE is an umbrella of specifications that have been implemented by several vendors. A reference implementation of the specifications can be downloaded from Sun Microsystems. Sun facilitates a worldwide community that contributes to the specifications of J2EE.

At the core of J2EE lays the Java language. This section focuses on the object middleware aspects of J2EE, which are Java, Java Remote Method

Invocation (RMI) and Enterprise Java Beans (EJB). In addition, the standardisation organisation for J2EE is discussed.

Java

On 23 March 1995 the latest innovative software of Sun Microsystems Inc. was announced in a front-page article in the San Jose Mercury News, as "*New software designed to make World Wide Web's 'home pages' more useful...* ". The innovation that made the headlines was the Java programming language.

The Java programming language is an object-oriented language, that was designed to have the "look and feel" of the C++ language, but it is simpler to use than C++. Java can be used to create complete applications that may run on a single computer or be distributed among servers and clients in a network. It can also be used to build a small application component or applet for use as part of a Web page. Applets make it possible for a Web page user to interact with the page.

Java is based on the principle that the same piece of software should run on a wide variety of computer systems, consumer devices and other pieces of hardware. Programs written in the Java programming language run on so many different kinds of systems because of the Java Virtual Machine (JVM). The JVM hides hardware and operating system specific features from the software. Java programs are pieces of object-oriented software that are converted to *bytecode* by a Java compiler. Bytecode is a machine independent binary run-time representation of the Java program that can then be (down)loaded and executed by the JVM.

A Java programs is robust, here meaning that, unlike programs written in C++ and perhaps some other object oriented languages, Java objects cannot contain references to data external to themselves or other known objects. This ensures that an instruction is inhibited to contain the address of data storage in another application or in the operating system itself, either of which would cause the program and perhaps the operating system itself to terminate or "crash." The JVM makes a number of checks on each object to ensure integrity.

Java RMI

The Java Remote Method Invocation (RMI) system enables an object running in one JVM to invoke methods on an object running in another JVM. RMI provides for remote communication between programs written in the Java programming language. Objects that are called through RMI have made their object reference available to client objects. An RMI object reference is a generic pointer to a server object that is used by the client to locate and communicate with a server object. A server object may publish

its reference in the RMI registry. An RMI registry maps names to object references.

Figure 3-7 shows a Java client invoking a Java server that runs in a JVM on a remote system. The figure also shows that the JVM hides the heterogeneity of the underlying distributed resource platform. In this case the DRP consists of a consumer device that runs the QNX operating system and a server PC that runs the Linux operating system.

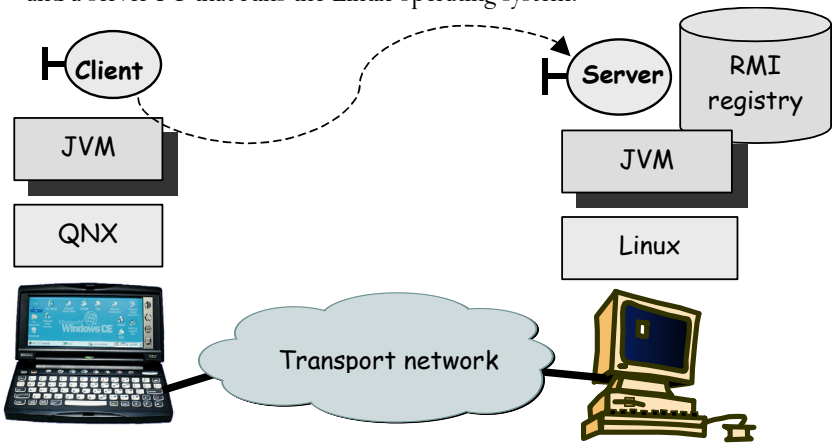


Figure 3-7 Java RMI in a distributed resource platform

Since Java bytecode can execute in any JVM, the RMI system allows bytecode to be downloaded over the network. This makes it possible to transfer objects and their behaviour across the network and execute a program in the vicinity of a client.

Java RMI uses the Java Remote Method Protocol (JRMP) to turn standard method invocations into remote method invocations. However, with JRMP both client and server objects must be written in Java. In order to support remote method invocations between Java objects and CORBA objects, RMI can also use the IIOP protocol from the OMG to transport method invocations.

Enterprise Java Beans

Enterprise Java Beans (EJB) is a specification [DYK01] for Java based distributed object computing. According to the EJB specification an enterprise bean is a part of a distributed application. In this thesis the term EJB component refers to an enterprise bean.

An EJB component is a computational object for which attributes, operations and method implementations have been defined in Java. EJB applications are distributed applications that consist of EJB components deployed in an EJB container. A container is the run-time environment of one or more EJB components.

The developer of an EJB component is freed from programming general-purpose services, such as naming, transactions and security. These services are configured through a deployment descriptor when an EJB component is deployed in a container. As a result, the developer of an EJB component, as opposed to a standard Java developer, no longer needs to write code that handles transactional behaviour, security, connection pooling or threading.

In essence, EJB is a server component model for Java and was design to support the development of server-side, scalable applications. It is the first example of an object middleware that supports a standardised execution environment for software components. A typical EJB server is shown in Figure 3-8 and consists of EJB containers, which run within the EJB server, and infrastructure services for naming, transaction and security. The container offers a run time environment to the EJB components.

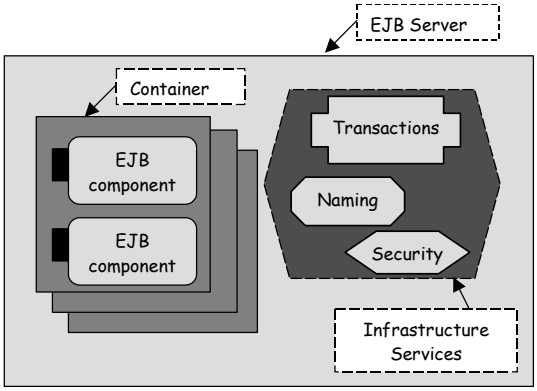


Figure 3-8 Structure of a typical EJB server

An *EJB server* is the run-time environment of one or more containers and provides services like a raw execution environment, multiprocessing, load-balancing and device access. It also provides the infrastructure services that are configured by a container during the deployment of an EJB component.

The *EJB containers* act as the interface between an EJB component and the outside world. An EJB client never accesses an EJB component directly. An EJB component is accessed through container-generated methods that in turn invoke the components' methods. The two types of containers are session containers that may contain transient, non-persistent EJBs whose states are not saved and entity containers that contain persistent EJBs whose states are saved between invocations.

EJB clients are the users EJB components. They find the EJB container that contains the EJB component through the Java Naming and Directory Interface (JNDI). EJB clients make use of the EJB container to invoke EJB component methods.

Standardisation

The Java Community Process (JCP) program has been initiated by Sun Microsystems. It is a process that Sun has formalised in 1998 to develop and revise Java technology specifications in close collaboration with the international Java community. This community has over 300 companies and individuals as members.

The Process Management Office is the group within Sun that overlooks and manages the daily operations of the program. The actual development of the specification occurs within the Expert Groups. Members can begin a Java technology specification by issuing a Java Specification Request (JSR). The Executive Committee (EC) is the group of members that decides on the life cycle of a JSR and has the authority to give the final approval to specifications. Part of the approval process is the availability of a "proof of concept", or reference implementation of a specification.

3.2.5 Evaluation

In the previous sections, the origin and features of three of today's most widespread used object middleware platforms have been outlined. This section evaluates the features and origins of CORBA, SOAP and J2EE and how this impacts the choices for the parts of a DPE.

Support for basic facilities

CORBA and the other OMG standards based on the ORB have boosted the development of object middleware. A CORBA platform supports all the basic facilities identified in section 3.2.1. But still, a considerable amount of research effort is put into improving ORB implementations, the specifications that accompany it and enriching the ORB with functionality to support components through container technology. EJB technology is on the forefront of the development of next-generation object middleware that supports containers and component technology. As such, EJB technology is a forerunner in the development of CORBA component specifications.

IIOP is the standard protocol that an ORB uses to package and transport messages. However, IIOP has shown some limitations when used in conjunction with current firewall technology and as such is not suitable for cross-organisational message exchange. SOAP does not have this limitation, since it uses HTTP as the underlying transport protocol and most firewalls allow unlimited traversal of HTTP traffic. Therefore, the emergence of SOAP is expected to have an impact on the way messages between distributed objects are packaged and transported.

SOAP has gained a lot of attention and support from large software companies such as Microsoft and IBM. Despite some of the simplifications made to SOAP, such as lack of language mappings and object activation

schemes, wide use is expected since many middleware vendors include SOAP support in their products. In particular when information must be exchanged across the boundaries of organizations, SOAP has the advantage that it can be carried over the existing Internet infrastructure. Consequently, no changes have to be made to the security settings of firewalls in order to exchange SOAP messages between organizations. Inter-organisational message exchange with other object middleware solutions requires a large amount of organizational and engineering resources. Since SOAP builds on existing Internet protocols, organizations are more likely to choose SOAP over other object middleware solutions for cross-organisational information exchange.

From a research perspective, SOAP introduces a number of challenges. SOAP not only uses more network bandwidth, but also parsing and generating of SOAP messages requires more processing compared to CORBA or Java RMI. Efficient solutions for parsing and generating SOAP messages should be investigated. In order to reduce bandwidth consumption, SOAP engines could, for example, compress messages on the fly. One could even image SOAP engines that make a dynamic trade-off between the added processing needed for compression and the reduced bandwidth consumption.

The *J2EE* specification is one of the first specifications for component middleware. It leads the way for object middleware based applications from an arbitrary set of distributed objects to a set of distributed components. The main advantage of EJBs over an arbitrary set of distributed objects is that EJBs require a strict separation between application logic and the deployment configuration of the EJB component. Application programmers do not have to consider deployment aspects such as naming, security and transactional properties.

EJB containers offer a standardised execution environment for EJB components, which separates the deployment configuration from the functional behaviour. The EJB specification is taken as an input to the development of other component execution environments. For example, the CORBA component model (CCM) is an OMG specification under development that is a strict super set of the EJB specification. The CCM enables the deployment of components written not only in the Java programming language, but also written in other languages supported by the CORBA language mappings, such as C++ and Smalltalk.

From a research perspective it is interesting to consider the QoS offered by a component as a deployment property. This requires that component containers expose the proper interfaces for configuring the QoS offered by a component. Clients should have a way to discover the offered QoS and establish some agreement with a container about the QoS they can expect

from a component. In addition, containers should have the mechanisms for enforcing the QoS agreements with clients.

Impact of standardisation efforts

The OMG coordinates the development of standards for object middleware and component based software design. Through its liaison with other consortia and standardisation bodies, some of its key specifications have become internationally accepted, beyond the OMG. For example, the OMG IDL and MOF specifications are accepted as ISO standards.

The ISO/IEC standards of most interest for this thesis are ones that define the Reference Model for Open Distributed Processing. The role of ISO is to establish dejure standards for open distributed systems, which include standards for the middleware, the computing systems and the transport network. ISO standards do not require a reference implementation.

Although the OMG does not produce implementations of its standards either, the standardisation process is designed in such a way that commercial implementations must be available before a specification is considered as a formal standard. In addition, a growing number of research effort is directed towards building (open source) implementations of OMG specifications [ORBacus, JacORB].

CORBA, the CORBA services and CORBA facilities have had an enormous impact on the advancements of object middleware architectures for the past decade. The OMG has managed to create a process that attracts many experts to devote their time and energy to the development of object middleware. As a result, a set of mature and widely supported specifications have been produced. The many commercial and open source implementations prove that these specifications lead to feasible implementations. Some research groups are specialised in finding bottlenecks for CORBA implementations [TuBu01] and have proposed optimisations [SGHP97]. This has lead to scalable and high-performance implementations. Other groups focus on testing and comparing the performance of several commercial and research implementations [CORBAComparisonProject].

However, despite the efforts on the continuous improvements of the specifications and the availability of many ORB implementations, CORBA may not become the ubiquitous middleware solution as some OMG members would have expected or hoped. Several middleware solutions are already in use for existing applications and companies find no reason to migrate their legacy to CORBA. Furthermore, the commercial interests of large companies such as Sun Microsystems and Microsoft are to keep or expand their portion of the middleware market and therefore promote their products aggressively. Finally, the growing interest in web technologies

such as XML and HTTP offers an alternative to CORBA technologies. XML is considered a very flexible way to represent information and HTTP is one of the most widely used protocols for conveying information. It can even be argued that executives are more familiar with web technologies and therefore XML and HTTP are more easily accepted than OMG IDL and CORBA.

The W3C has recently gained a lot of attention with the eXtensible Markup Language (XML) standard. XML is broadly accepted as a standard for structuring text documents. A growing number of application components use XML to exchange information. The SOAP standard defines how messages are packaged as an XML structure and how these messages are transported over the network. The W3C produces recommendations that advance XML, SOAP and related standards. That makes W3C also a player that impacts the structure of a DPE.

The SUN-JCP program concentrates on Java technology specifications and does not consider other programming languages or run-time environments, therefore, it can move relatively fast compared to the OMG process. The OMG also considers programming languages and run-time environments other than Java and delivers specifications that are programming language neutral.

The Java Specification Requests cover a wide range of topics, including specifications of Java interfaces to XML documents, SOAP and a Java specific version of the Meta Object Facility (MOF). As such, the JCP has an impact on the advancement of Java-based middleware. Through close collaboration with the OMG and participation of JCP members in the OMG process, some of the specifications developed in the JCP program have been leveraged to a programming language-neutral OMG specification. For example, the Enterprise Java Beans (EJB) specification has formed an important starting point for developing the CORBA Component Model specification.

3.3 Network technologies

The transport network forms an important part of the Distributed Resource Platform. A network consists of connected network nodes (e.g., routers and switches). This section discusses the developments in network technology, with a focus on the mechanisms and protocols implemented in the network nodes that can be employed for the control of network level QoS of Internet Protocol (IP) networks.

As the processing capacities of end nodes (e.g., PCs and workstations) rapidly improve and the amount of data flowing through the network is ever increasing, it becomes important that the utilisation of network resources

can be managed and controlled. With QoS awareness in packet-based networks we mean that the network nodes have the ability to influence the performance of the network. A QoS aware network consists of nodes that implement suitable QoS mechanisms. Network performance is defined as a set of QoS characteristics offered by the network and is expressed in terms of bandwidth, delay, jitter and/or throughput.

We first describe the basic QoS mechanisms and then give an overview of current research activities on the main standards and research efforts.

3.3.1 Basic mechanisms for QoS in packet networks

QoS in packet networks is mainly driven by the need to provide QoS differentiation to various users and/or applications. For example, a video distribution application may have stringent requirements on delay variance (i.e., jitter) of the packets that are transported between a video source and sink, but may permit a certain percentage of packet loss. However, a file transfer application may require that no packets are lost, but allows for significant delays and jitter. These are two examples of applications that use the same network, but have different QoS requirements. To create this QoS differentiation, the network nodes need to manage a number of resources. For the management of resources a number of mechanisms are available.

The most important resources that routers in packet networks need to manage for service differentiation are buffers and bandwidth. Corresponding mechanisms are buffer management schemes and scheduling mechanisms, respectively [GuPe99, WOS00]. Buffer management schemes decide which incoming packets are queued for transmission and which packets are discarded, while scheduling mechanisms decide when outgoing packets are transmitted.

Figure 3-9 depicts a model of a router. Packets arrive at the incoming network interface and leave the router at the outgoing network interface. Internally the packets are stored in one of the queues according to some buffer management scheme. A scheduler, servicing each queue, determines when packets are forwarded to the outgoing network interface according to some scheduling mechanism.

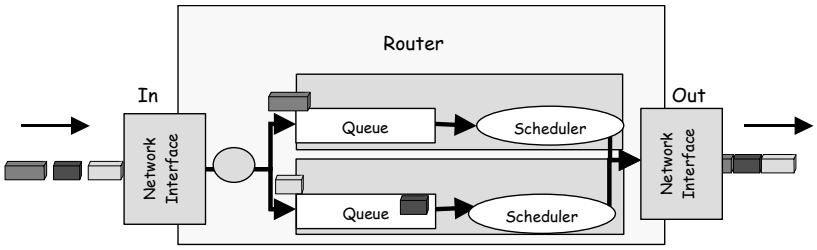


Figure 3-9 A model of a router

The combination of the scheduling mechanism and the buffer management scheme determine the QoS perceived by a particular dataflow. For example, a more frequent scheduling of packets from a flow increases the bandwidth for that flow, whereas the buffer size and buffer management determine the delay and jitter (i.e., delay variance). The research community has proposed several buffer management schemes, such as Class Based Queuing (CBQ) [FlJa95] and Weighted Fairness Queueing [DKS90, Pa92]. For each management scheme a trade-off exists between fairness (i.e., all flows get a fair share of the resources), efficiency (i.e., storage capacity requirements are within reasonable limits) and complexity (i.e., determination where a packet is stored scales well with the number of flows).

At the network layer, we distinguish between a control plane and a data transfer plane (see Figure 3-10). The data transfer plane is sometimes referred to as the *data path*, or the *fast path*. Packets travelling through the network can thus be classified as signalling packets (i.e., control packets) or as application packets (i.e. data packets). If network nodes assign application packets to dedicated buffers and associated schedulers, the network can distinguish between premium traffic and best-effort traffic. The signalling messages are used to exchange resource reservations or traffic policies between network nodes. The network node maps the resource reservations or traffic policies to the appropriate buffer management schemes and schedulers. Figure 3-10 shows a network node processing signalling, premium and best-effort traffic. The control plane processes and forwards signalling packets, while the data transfer plane processes two classes of application packets: premium and best-effort packets.

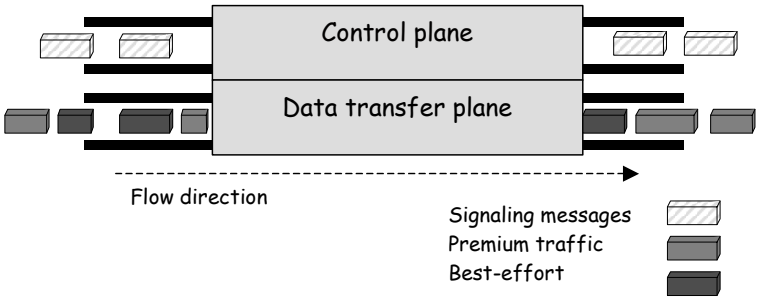


Figure 3-10 Control and data transfer plane packets at a network node

Examples of control plane protocols are RSVP and Boomerang. These protocols are described in the next two sections.

3.3.2 RSVP

The Resource reSerVation Protocol (RSVP) is a network-control protocol that enables applications to obtain an agreed QoS for their data flows. RSVP is not a routing protocol; instead, it works in conjunction with routing protocols and installs the equivalent of dynamic access control lists along the routes that routing protocols calculate. These access control lists determine which packets are treated as premium traffic and which packets are treated as best-effort traffic.

In RSVP, a data flow is a sequence of packets that have the same source, destination (one or more), and quality of service. QoS requirements, such as average and peek packet arrival rates, are communicated through a network using *flow specifications*. A flow specification is carried through the network as the payload of a signalling packet.

RSVP data flows are generally characterized by *session*s, over which data packets flow. A session is a set of data flows with the same unicast or multicast destination, and RSVP treats each session independently. RSVP supports both unicast and multicast sessions (where a session is some number of senders talking to some number of receivers), whereas a flow always originates from a single sender. Data packets in a particular session are directed to the same IP destination address or a generalized destination port. The IP destination address can be the group address for multicast delivery or the unicast address of a single receiver.

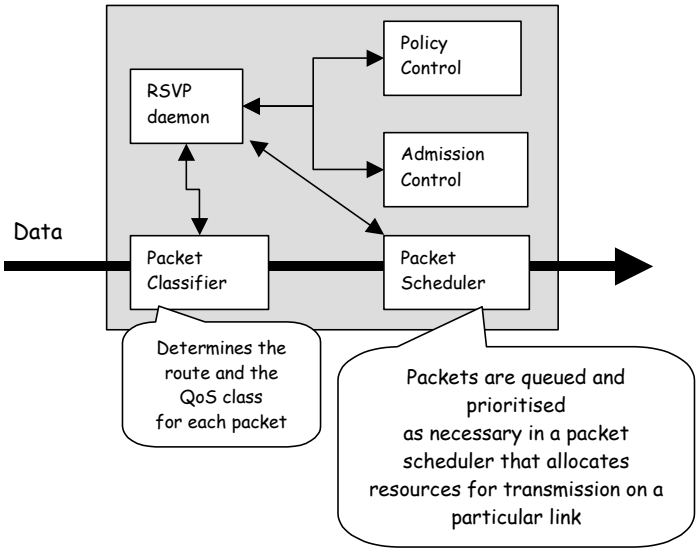


Figure 3-11 RSVP functions

RSVP in operation

Figure 3-11 depicts the key functions of RSVP. The RSVP resource-reservation process begins when an RSVP daemon consults the local routing protocol(s) to obtain routes. Each router that is capable of participating in resource reservation passes incoming data packets to a packet classifier and then queues them as necessary in a packet scheduler. The RSVP packet classifier determines the route and QoS class for each packet. The RSVP scheduler allocates resources for transmission on the data path.

3.3.3 Boomerang

Boomerang [Fe99] is a recent development from Telia Research and Budapest University of Technology. Boomerang is a lightweight signalling protocol for IP networks that can be used to signal per micro-flow requirements to the network and to reserve resources end-to-end.

The boomerang protocol offers similar network control features as RSVP, however it aims to overcome the following limitations:

1. RSVP relies on per micro-flow state that results in a scalability problem in terms of memory, capacity and processing time.
2. RSVP is complex to implement both in nodes and hosts due to separation of reservation and path finding messages and receiver diversity.
3. RSVP requires modification in the far end host (i.e., the destination host of a micro-flow).
4. RSVP spreads the signalling processing over the network. Each node along a reserved path contains a flow state and a signalling state. Therefore, each reservation session increases the load on network nodes.
5. RSVP requires multiple interactions between sender and receiver for a successful reservation setup.

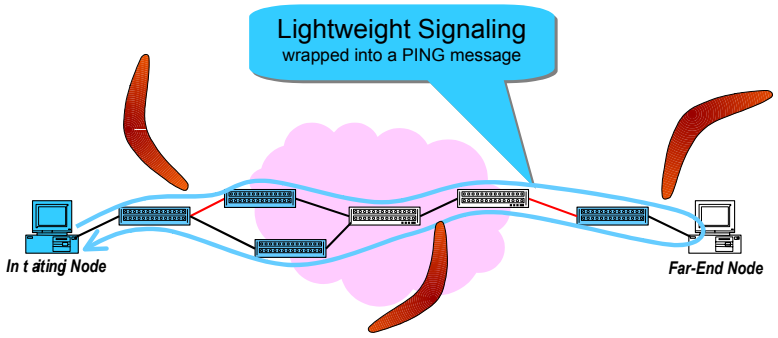
Boomerang uses a single signalling message to set-up a bidirectional softstate reservation of resources in the network. The signalling messages are wrapped inside an ICMP ECHO message. This design decision eliminates the need for modification of the far-end host and makes it possible to reserve resources in both directions in a single signalling loop.

Boomerang in operation

The Boomerang resource-reservation process begins when an initiating node sends a boomerang message to a far-end node, containing the requested upstream and downstream bit rates. The far-end node echoes the message back to the initiating node. This is depicted in Figure 3-12.The

boomerang message allocates resources along the route, which is determined by standard routing protocols.

Figure 3-12 Boomerang in operation



The initiating node is responsible for handling the flow-state of an established reservation. Therefore refresh messages are sent out periodically to keep the reservation alive and possibly adapt to route changes.

3.3.4 DiffServ

RSVP and Boomerang are examples of protocols for network performance control, which belong to the framework of Integrated Services (IntServ).

The main issue with IntServ is the need to maintain state information at every node for every flow. In addition, the classifier of an IntServ router needs to inspect multiple fields of the header of each data packet. To reduce the state and to simplify the classification function, the Differentiated Services (DiffServ) framework uses the technique of packet marking. Each packet is marked with a flag indicating how to treat it. This field is called the Differentiated Service Code Point (DSCP).

The DSCP is used to select the per-hop behaviour (PHB) that a packet experiences at each DiffServ router along a route. A PHB determines how a packet is forwarded, such as the relative weight for sharing bandwidth or a relative priority for dropping. The mapping of a DSCP to a PHB at each router is not fixed.

When a packet enters a domain of DiffServ routers its DSCP field is marked according to the service quality a packet is entitled to receive. Within the domain of DiffServ routers each router only needs to look at the DSCP to decide how to treat a packet. Routers do not have to maintain per flow state or use a complex classification function. The complexity of deciding what DSCP to assign to a packet is pushed to the edge routers of a DiffServ domain.

Interworking between DiffServ and IntServ

The separation of functions performed at the edge of the network from the functions performed by core routers, is vital for the scalability of DiffServ. On the other hand, DiffServ does not allow applications to specify the end-to-end QoS of a particular network flow. However, for a DPE the end systems could benefit from the ability to specify the expected network QoS on a per-flow basis. Some have suggested the integration of DiffServ and IntServ solutions [RCV98, De+99]. The main idea of this integration is that DiffServ technology is used in the backbone, where as IntServ technology is used to access the DiffServ domain. Figure 3-13 shows how end systems access the DiffServ domain through an IntServ network. Key to the design of such integration is the mapping of IntServ reservations to marking of packets with DSCP values at the edge routers.

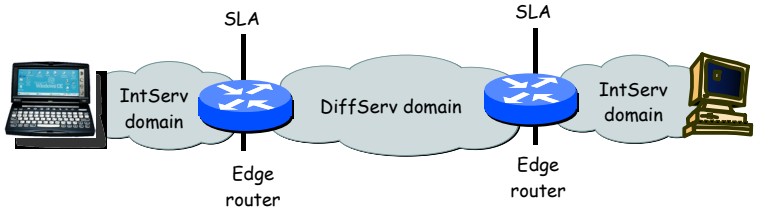


Figure 3-13 IntServ access to a DiffServ domain

3.3.5 Evaluation

The network performance protocols and standards discussed in the previous sections can be used to realise performance support at the middleware layer. They provide a means for the middleware to control the performance characteristics offered by the network.

The RSVP protocol was designed to reserve network bandwidth from a single source to multiple receivers, but can also be used to establish point-point reservations. However, in an object middleware context, two point-point reservations must be made to support a single client-server association in order to carry both request and reply messages over a reserved communication channel. RSVP has considerable overhead in terms of control messages needed to create a network reservation.

Boomerang was designed to reduce the number and size of control messages compared to RSVP. In addition, it can create a bi-directional reservation from source to sink in a single reservation request. This reservation can be asymmetric in order to facilitate different upstream and downstream reservations.

Boomerang seems more suitable for application in an object middleware context, however it is still in a research phase and it is not as widely supported in routers and computers as RSVP.

RSVP and Boomerang require each node along the path of a flow to maintain per-flow state information. This is typical for any IntServ approach to network QoS. The IntServ approach is complemented with the DiffServ approach. Most of the scalability drawbacks of IntServ have been resolved in the DiffServ approach. A combined solution of IntServ and DiffServ seems a feasible way to realise end-to-end QoS in a large scale packet network.

3.4 QoS architectures

The notion of QoS is broad and is applied to many areas, such as end-user quality perception, ergonomic quality of user interfaces, network performance, system performance. This section gives an overview of the terminology used in this thesis to express the QoS aspects of a DPE. Other QoS aspects such as user needs, customer satisfaction or price/quality ratios are not considered in this thesis.

A number of QoS definitions are presented, and then the concepts of a QoS framework derived from an ISO/IEC standard are described.

3.4.1 QoS definitions

Several definitions of QoS can be found in standards and literature. The following table quotes some of these definitions:

Table 3-1 Examples of QoS definitions

Origin	Definition
ISO/IEC (X.641) / ITU/T 13236	QoS is a set of qualities related to the collective behaviour of one or more objects [ISO X.641].
ISO 8402	Quality: the totality of features and characteristics of a product or services that bear on its ability to satisfy stated or implied needs [ISO8402].
QOSMIC	QoS is a set of user-perceivable attributes, which describe a service the way it is perceived. It is expressed in a user-understandable language and manifests itself as a number of parameters, all of which have either subjective or objective values. Objective values are defined and measured in terms of parameters appropriate to the particular service concerned, and which are customer-verifiable. Subjective values are defined and estimated by the provider in terms of the opinion of the customers of the service, collected by means of user surveys [Me91, Me92].

ISO/IEC (X.902) / ITU/T 10746-2	The notion QoS is a system or object property, and consists of "a set of quality requirements on the collective behaviour of one or more objects... QoS is concerned with such characteristics as the rate of information transfer, the latency, the probability of a communication being disrupted, the probability of system failure, the probability of storage failure, etc." [ODP2]
Tutorial "The Enterprise of QoS"	The notion *quality of service* is defined by its purpose (objective), scope and policies applied. So, the "purpose of QoS" is to guarantee contracted quality throughout the use of a service to a community of agents or objects. A QoS contract defines agreement by specifying requirements and obligations for a community that is involved in the service application. A QoS contract also specifies the policies to keep track about QoS during all phases of application. These policies constrain the activities about QoS that are undertaken by the community objects, i.e., the enterprise, to achieve the system objectives [MeHa98]

3.4.2 QoS terminology

The terminology with respect to QoS used in this thesis is based on concepts of ISO/ITU QoS framework [X.641]. The QoS concepts and definitions that are introduced comprise a framework for modelling the QoS aspects of an open distributed system.

The framework considers a service user and a service provider who describe QoS aspects by QoS requirements, characteristics, management functions, categories, mechanisms, activities and phases. For example, user requirements are conveyed as parameters to the service provider. The service provider is able to determine management functions according to the requested characteristics or categories of QoS. The management functions comprise components of appropriate mechanisms. The application of mechanisms occurs as activities in a behavioural specification and will be controlled during pre-defined QoS phases.

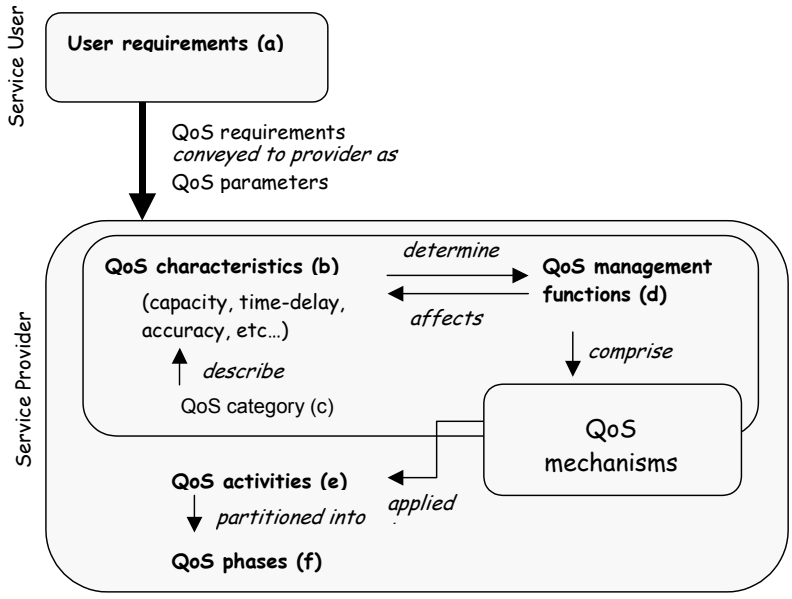
*Figure 3-14*shows a graphical representation of the relationship between user requirements, QoS characteristics, QoS categories, QoS management functions, and QoS mechanisms for the service user and the service provider to support QoS. In Chapter 5 we elaborate on these definitions.

The main purpose of Figure 3-14 is to show the concepts (a, b, c, d, e and f) concerning the QoS provisioning process:

a) A service user expresses its *User requirements* independent of the way a service provider realises these requirements. User requirements are conveyed to the provider as QoS parameters.

b) The QoS parameters are expressed in terms of the *QoS characteristics* that a service provider supports.
c) A *QoS category* describes one or more QoS characteristics.
d) The *QoS management functions* of a service provider affect the QoS characteristics that a service provider supports. Conversely, the QoS characteristics determine which QoS management functions a service provider employs to provide QoS support. A QoS management function is comprised of one or more *QoS mechanisms.*
e) A QoS mechanism applies one or more *QoS activities*
f) QoS activities are partitioned into QoS phases.

Figure 3-14
Concepts of the QoS framework



3.4.3 Utilisation

The QoS user-provider relationship is applied three times in this thesis. First a QoS user-provider relation is identified to model the relation between an application component (as user) and the DPE (as provider), and then a QoS user-provider relation is identified between the middleware (as user) and the DRP (as provider). The third QoS user-provider relation is identified between a computational client object (as user) and a computational server object (as provider).

Chapter 5 further applies the QoS user-provider relationship to the construction of QoS models for open distributed systems.

3.5 Software engineering technologies

The QoS aware DPE offers support for interactions between computational objects that are implemented for a possibly heterogeneous set of distributed resources. Ideally, the structure and behaviour of these components should be modelled in an implementation language independent way. This eases the portability of collaborating computational objects in a heterogeneous distributed resource platform. Therefore, several software companies have specified meta-models [OMG-CWM, EDOC] that are used to develop models of software.

The various meta-models that are used to develop models that specify (parts of) the DPE has led to the need for a generic and standardised framework for the management, manipulation and exchange of these models. The OMG has addressed this need with the specification of the Meta-Object-Facility (MOF) [MOF]).

This section presents a brief overview of the current status of Unified Modelling Language (UML) [UML], which is a modelling language that is used to develop models of software. The widely adopted UML has enabled the standardisation of the MOF, which is also described in this section.

3.5.1 UML overview

The UML is a graphic language for specifying, visualising and constructing artefacts of a software system [Ko99]. The first version of the language was published in 1996 when the modelling languages found in the Booch, OOSE/Jacobson and OMT methods where combined. The combination of the three modelling languages into a single language resulted in UML 0.9. Since 1997 the further development of the UML has become subject to the OMG process. Already several new versions of UML have been standardised and more revisions are expected to pass through the OMG process. Each revision extends or refines the syntax and semantics of the UML.

The basic building blocks of the UML are model elements, relationships and diagrams. The model elements include e.g., classes, interfaces, components and use-cases. Examples of relationships are associations, generalisations and dependencies. The diagrams of the UML are used to express various views on a software system. Diagrams can be class diagrams, use case diagrams, interaction diagrams and others. The building blocks of

the UML can be used to construct large, complex structures that describe the blueprint of a software system.

The UML specification defines syntax and the semantics of the UML. The syntax definition includes the *UML notation guide*, which also defines the graphical notation for UML building blocks. The UML can be extended through the definition of a *UML Profile*. A profile does not introduce new basic concepts, but provides a way to specialise the UML for a particular environment or domain. A UML profile associates specific semantics with the UML basic building blocks.

3.5.2 The Meta-Object Facility

The MOF is a generic framework for describing and representing meta-data. Meta-data in this context denotes any data that in some sense describes other data. Although the MOF supports any kind of meta-data it is particularly suited for handling data that represents a model. A model in the MOF context refers to a collection of meta-data that describes a collection of related data. In the context of MOF, the model of a collection of related data is regarded as the meta-data of this collection of data. As a result, a MOF (meta-) model is an abstract language that can express this collection of data.

Modelling data recursively as meta-data leads to a potential infinite number of meta-levels. The architecture of MOF defines four layers of meta-modelling, that are labelled M0, M1, M2 and M3:

- Layer M0 - the *instances*: information (data) that describes a concrete system at a certain point in time. This layer consists of instances of elements of the M1-layer.
- Layer M1- the *model*: definition of the structure and behaviour of a system using a well defined set of general concepts. An M1-model consists of M2-layer instances.
- Layer M2 - the *meta-model*: The definition of the elements and the structure of a modelling language. An M2-layer model consists of instances of the M3-layer.
- Layer M3 - the *meta-meta-model*: The definition of the elements and the structure for the description of a meta-model.

The meta-meta-model, or M3-layer model, is standardised in the OMG MOF specification. The M3-layer model is also referred to as the *MOF model* and forms the fixed point that unifies MOF compliant models. The 4-layer MOF structure is depicted in Figure 3-15.

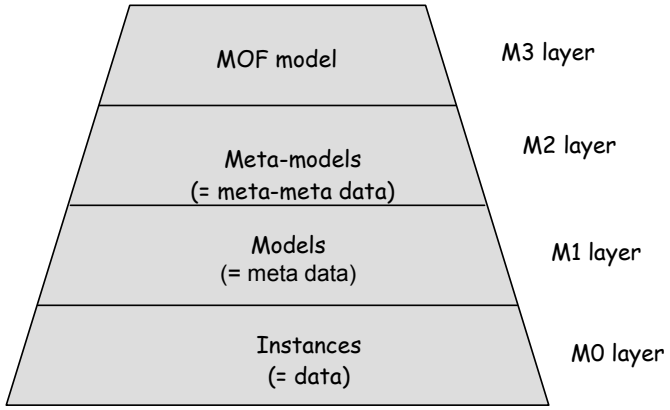


Figure 3-15 The MOF layers

Elements and structure of the MOF model are directly derived from the object-oriented formalism. The MOF-model consists of the following concepts for the definition of meta-models [HKB01]:

- *Classes*: Classes are first-class modelling constructs. Instances of classes (at M1-layer) have identity, state and behaviour. The structural features of classes are attributes, operations and references. Classes can be organized in a specialisation/generalisation hierarchy.
- *Associations*: Associations reflect binary relationships between classes. Instances of associations at the M1-layer are links between class instances and do not have state or identity. Properties of association ends may be used to specify the name, the multiplicity or the type of the association end. MOF distinguishes between aggregate (composite) and non-aggregate associations.
- *Data types*: Data types are used to specify types whose values have no identity. Currently MOF comprises the CORBA data types, i.e., integers and string, and OMG IDL interface types.
- *Packages*: The purpose of packages is to organize (modularise, partition and package) meta-models. Packages may be nested, inherit from other packages or import components from other packages.

The MOF standard defines a representation of the MOF model (i.e., the M3 layer) in OMG-IDL. This representation consists of the OMG-IDL module *Model* (meta-model specific interfaces) and *Reflective* (generic interfaces). All interfaces in Model directly or indirectly inherit from interfaces defined in Reflective. The MOF interfaces, as defined in the Model and Reflective, allow to:

- Stepwise create a new meta-model in the MOF by creating new objects,
- Change an existing meta-model in the MOF,

– Extract information from a meta-model using query functions and traversal functions,
– Request a validation of the meta-model.

To produce an external representation of a meta-model (externalise) or to create a meta-model from an external representation (internalise), the mapping to an external format must be defined. Currently two specific mappings from MOF to external formats have been standardised:
– MOF-IDL-mapping: This mapping generates the IDL-specification for a meta-data service from a MOF-meta-model specification. This service (e.g., repository) is used to store or manipulate models (M1-layer), which are conforming to an M2 model. An example for such a service is the UML CORBAfacility [UML-F] that is derived from the UML-meta-model.
– XMI (XML based model interchange): This mapping defines rules to derive an XML Document Type Definition (DTD) [Bo98] from a meta-model in MOF and to represent an M1-model as an XML document structured according to that DTD.

The MOF-IDL-mapping enables the automated generation of a meta-data repository that allows CORBA applications to access meta-data about application objects at run-time. This meta-data could for example be (a part of) the design constructed by an application designer. The XMI specification offers a standardised way to represent meta-data as an XML document and ensures that MOF-based meta-data can be deployed in a scope wider than pure CORBA applications. The main purpose of the XMI mapping is to exchange designs between design tools.

3.5.3 Evaluation

The convergence of modelling languages into the UML has led to the availability of a set of widely accepted software modelling building blocks. These building blocks enable a software designer to create a well-supported description of a software system, which can easily be exchanged with another designer or integrated other with UML based software designs. The UML provides a language for describing application components that use the DPE as an execution environment. In addition, the UML can be used to specify the design of middleware components.

The MOF further extends the capabilities of the UML. The main advantage of the MOF-approach is to make the definition of (meta)-models independent of a concrete domain, and to provide a concise and unique set of concepts for the definition of meta-models.

MOF can be employed to define multiple meta-models that are used to develop models of system. The MOF model is a suitable meta-meta-model that unifies the meta-models of the modelling concept space defined in Chapter 2. MOF compliant meta-models and models, specifying various viewpoints and views of a DPE respectively, are easily manipulated, managed and exchanged by MOF compliant tools.

MOF enables an approach to the design of a QoS aware DPE where one or more meta-models are used to define the functional characteristics of a DPE and one or more other meta-models are used to specify the qualitative aspects of a DPE.

3.6 Related work

The research area presented in this chapter comprises several subjects as shown in section 3.1. For each subject several advances and ongoing research can be reported. However, in this section we limit ourselves to only discuss research activities that are related to the development of a QoS aware DPE. The activities discussed are: QML, QuO and Quartz.

3.6.1 QML

The Quality of service Modelling Language (QML) [FrKo98] is a language for defining QoS specifications for distributed objects. QML originates from HP Laboratories, Palo Alto. QML is 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*. This specifies the QoS category, such as reliability or performance. 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.
- *Contract*. This defines the constraints on the dimensions of the contract type.
- *Profile*. QML uses profiles to associate contracts with interface entities.

QML effectively treats contracts as abstract data types; they can be defined and reused by name. This allows QML to support inheritance between contract types. This behaviour is known as refinement. One type of a refinement on a particular contract consists of the specification of properties that were not present in the original contract. It is also possible to create a new contract that inherits the properties of another contract, but with altered properties.

QML also supports conformance between contracts. This allows two things: a contract P can be said to be stronger or weaker than a contract Q, and allows a specification that provides P to satisfy one that requires Q, provided that P is stronger than 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.

QML also has a QoS fabric, called QRR, which makes it possible to manipulate QoS specifications at runtime. It also allows creating new QoS specifications at runtime. QML however does not prescribe how these specifications should be enforced by the middleware.

QML offers a basic framework for specifying QoS contracts and contract types. However, QML does not prescribe any specific QoS categories, QoS dimensions or QoS contract types. The issue of QoS specification in a way that reflects the actual requirements of an application domain still remains. In addition, QoS specifications should be defined in such a way that they can be supported by the middleware and the appropriate QoS mechanisms are present to realise a QoS contract.

3.6.2 QuO

Quality Objects (QuO) is a framework for providing quality of service (QoS) in network-centric distributed applications [PLS+00]. QuO supports the specification of QoS contracts between clients and service providers, runtime monitoring of contracts, and adaptation to changing system conditions. It is developed by BBN Technologies [VZL+98].

The QuO application not only consists of the client program, ORB, and (target) object, it also has the following components, shown in Figure 3-16, provided to an application developer:

- A local delegate of the remote object. The delegate provides a functional interface identical to the remote object, but can trigger contract evaluation upon each method call and return.
- A QoS contract between the client and (remote) object. This specifies the level of service desired by the client, the level of service the object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
- System condition objects (SysCond) interface between the contract and resources, mechanisms, objects, and ORBs in the system. These are used to measure and control QoS.

When a client calls a remote method, the call is passed on to the object's local delegate. The local delegate then passes 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 fulfil the requirements.

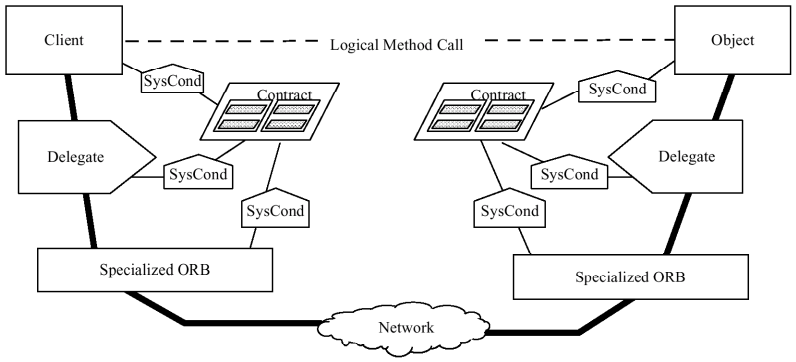


Figure 3-16 A remote method call in a QuO application

At runtime, client and server interact about the level of QoS they can provide. Callbacks into both the client and server are used to signify that a change in imminent. The QuO architecture provides objects that define these callbacks; the developer is responsible for implementing their behaviour. The delegates, depicted in the figure above, are free to modify their own behaviour in any way desirable to maintain the systems current QoS. For example, if the network throughput degrades to such a level that they may degrade below the QoS specifications, the delegates may choose to compress the data and thus to trade CPU cycles for throughput.

3.6.3 Quartz

The Quartz QoS architecture aims to solve the lack of flexibility and expressiveness in QoS specification that can be found in other QoS architectures. Quartz originates from Trinity College Dublin from the hand of Frank Siqueira [SiCa00]. Quartz was designed with the following requirements in mind:

- Users can express QoS according to the notion of quality that is appropriate at application level.
- Transparency of the characteristics of reservation mechanisms and platforms present at lower levels.
- Adequacy to open systems, in which different protocols and hardware co-exist
- Support for dynamic resource adaptation to be performed by the system without loss of service consistency at application level

The main component of Quartz is a QoS agent. The agent is responsible for harmonising the capabilities of the lower level protocols and platforms, with

the application level QoS requirements. Several case studies have been conducted to successfully validate the approach.

3.7 Conclusions and further directions

A DPE is an open distributed system that is constructed from hardware and software components that are obtained from various vendors and open source communities. Several forces have an impact on the construction of a QoS aware DPE. This chapter discusses four research subjects that influence the technological advancement of a QoS aware DPE. These research subjects are object middleware architectures, QoS architectures, network technologies and software engineering technologies.

Standardisation of object middleware platforms is impacted by several standardisation organisations. This chapter shows the mutual interests and dependencies of these organisations and how this impacts the standardisation of object middleware. At least three competing object middleware standards are evolved and improved by various organisations. It is not likely that these standards will converge into a unified object middleware standard, due to (technological) differences in these standards and the commercial interests of companies that have implemented these standards.

Vendors that build products based on object middleware standards offer very similar but nevertheless non-interoperable system parts. As a result, the market for DPE products is segmented. At this point in time only assumptions can be made about the products and standards that will be used by the majority in the long run. Therefore, a design of a QoS aware DPE has to be generic in the sense that it is applicable to DPE products in different market segments.

EJB component technology is a leading example of an object middleware technology that enforces the separation of the functional behaviour of computational objects from component deployment. It emphasises the distinction between the role of application designer and deployment designer. Ideally, this distinction should be further enforced by a QoS aware object middleware platform. Such a platform should enable a deployment designer to configure QoS characteristics of software components at deployment time.

The QoS offered by a middleware platform depends on the QoS offered by the DRP. To support QoS, object middleware platforms must therefore control the QoS offered by the DRP. This chapter focuses on the QoS support offered by a packet network.

Two competing approaches to network QoS provisioning, i.e., IntServ and DiffServ have been identified. The IntServ approach allows for fine-

grained QoS control, but requires each network node to maintain per-flow state information along the path of a flow. This is not required by the DiffServ approach at the expense that is allows for course-grained QoS control.

Analogous to the developments in the area of object middleware architectures, the developments in the area of network technologies for QoS support indicate that it is not likely that there will be a unified solution for the control of network QoS.

New protocols and mechanisms for the control of QoS in packet-based networks are expected to emerge. Consequently, the QoS control interfaces and the quality delivered by future networks will change over time. Therefore, the approach to QoS provisioning should be service driven, i.e., QoS support at the DPE level should not reveal the protocols, interfaces and mechanisms used by the DRP to control the QoS. QoS support for a DPE should be offered as a generic service that abstracts from underlying QoS enforcement functions. A service driven approach to QoS provisioning must be extensible, in the sense that new mechanisms for QoS control can be incorporated when such mechanisms become available.

In the area of software engineering technologies, the UML represents the convergence of software modelling languages. The UML supports the use of multiple viewpoints and associated meta-models, which is in line with design concepts presented in Chapter 2.

The MOF model is a generic meta-meta-model that suits our need to construct multiple meta-models. These meta-models constitute a modelling concept space, which can then be used to develop models of a QoS aware DPE. In this thesis we use the MOF model in Chapter 5 to develop a QoS meta-model.

Chapter 4

An object middleware reference model

The purpose of this chapter is to construct an object middleware reference model. Many similarities can be discovered in the structure of different object middleware platforms if specific implementation choices are ignored. In the next chapters the object middleware reference model is used to introduce QoS awareness into object middleware. Defining support for QoS in object middleware based on the common structures of a reference model instead of some specific architecture makes the proposed solutions more widely applicable.

Our approach to the construction of an object middleware reference model starts in section 4.1 with a discussion of the supporting role of object middleware in the design of distributed applications. Our experiences with the supporting role of object middleware have been reported at several international conferences and workshops [HNSW99, HTW98, KHSW00, NiHa99, OlHa98]

To identify the common structures in object middleware platform that have remained invariant over the past decades, section 4.2 discusses a number of early middleware platforms and the support these platforms provide to an application designer. The results of this discussion, the distribution transparencies identified in Chapter 2 and a review of the functions and layers of contemporary object middleware platforms, provide the basis for a list of features that should be supported by middleware that complies with our reference model. This list of features is presented in section 4.3 after the layers and functions of contemporary object middleware platforms are reviewed.

A generic object middleware model is then constructed that complies with the major object middleware systems. Sections 4.4 to 4.6 present the object communication middleware, general purpose services and the

component execution environment, respectively. These are the main parts of our object middleware reference model.

Finally, section 4.7 assesses the compliance of our object middleware reference model with current object middleware technologies and presents the conclusions of this chapter.

4.1 Object middleware as a supporting infrastructure

This section discusses the role of object middleware in the design of distributed applications. The discussion starts with a generic approach to distributed system design, using generic design principles such as refinement and abstraction. This approach is then applied to the design of a distributed system, without consideration of object middleware, i.e., considering the design of a distributed system from scratch. Then we consider the use of object middleware and how distributed system design benefits from object middleware.

The objective of this section is to investigate the need for a supporting generic infrastructure, independent of a specific distributed application. Current object middleware platforms are examples of such supporting generic infrastructures.

4.1.1 Structured distributed system design

Chapter 2 introduces the notions of abstraction, refinement and decomposition. Abstraction and refinement are there positioned as opposite design steps. Decomposition is defined as a special case of refinement, which concerns the refinement of parts of a design into subparts. This section applies these structured design principles to the design of a distributed system using objects as basic modelling entities.

Figure 4-1 shows two approaches to distributed system design. In a strict top-down design process, a distributed system is refined in a sequence of decomposition steps. Decomposition enables a designer to model a system part at a more fine-grained level. The decomposition stops when an object in the design is available in software or hardware.

In a strict bottom-up process, a distributed system is composed in a sequence of abstraction steps. Abstraction enables a designer to model a system part at a more coarse-grained level that hides the implementation of that part. The composition stops when one or more coarse-grained objects, which represent the system in an integrated way, model the distributed system.

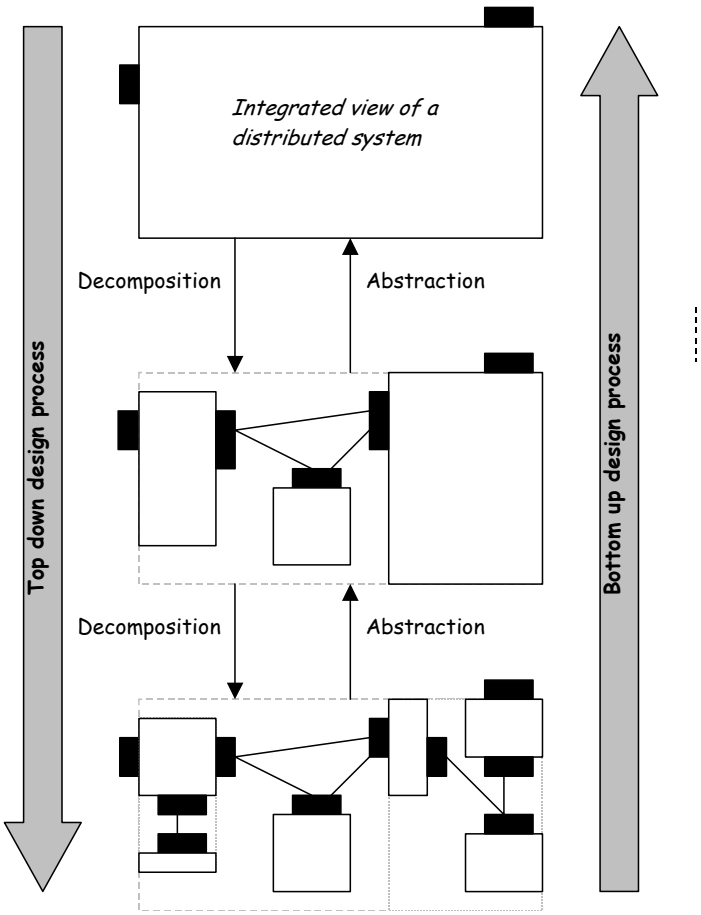


Figure 4-1
Decomposition and abstraction applied to distributed system design

In practise, a distributed system is designed using a combination of top-down steps and bottom-up steps. A designer aims to balance the design process by constructing a model from objects that represent entities that are available in the implementation concept space. Availability of an implementation of an object in the implementation concept space is a reason to stop the further decomposition of this object. Another reason to stop the decomposition of an object is that the implementation of that object is automated using transformation rules. Figure 4-2 shows how a model and its implementation are related.

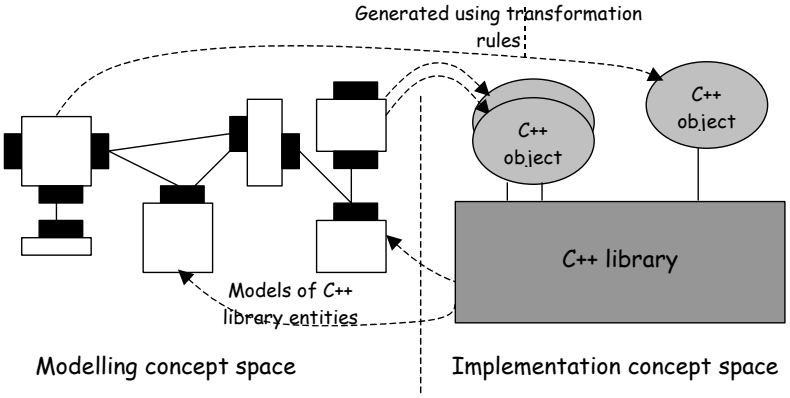


Figure 4-2 Applying transformation rules to relate a model with its implementation

The observation is that decomposition by a designer should stop when either the parts of a model are readily available in the implementation concept space, or when the parts of a model can be generated using a set of transformation rules.

For example, a designer of a shared whiteboard application models a communication library as a set of objects that can establish bi-directional reliable connections between two end-points. A suitable design pattern that represents this functionality of the communication library model is the Acceptor/Connector pattern [Sc97]. During the top-down design of the whiteboard application, the designer ensures that at some level of decomposition Acceptor and Connector objects appear. The transformation of the Acceptor/Connector objects to the implementation concept space becomes trivial; therefore no further decomposition of these objects is necessary.

4.1.2 Distributed system design without object middleware

Now consider the design of a distributed system, in case there is no object middleware available and a distributed system must be designed from scratch. As already identified in chapter 2, several roles can be distinguished that are involved in the design of the distributed system. The application designer designs a distributed application under the assumption that there is an infrastructure that provides supporting functions to the application objects. An application designer models this infrastructure in at a high level of abstraction, leaving the refinement of the infrastructure to the infrastructure designer.

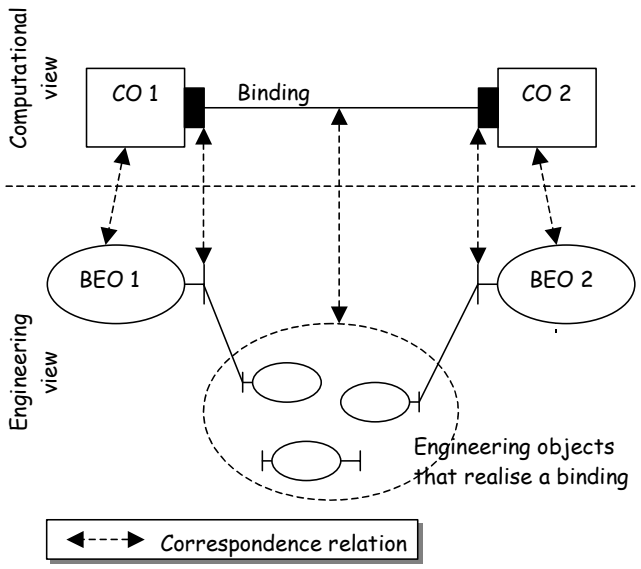
The application designer uses the computational viewpoint to construct a specification of a distributed application and provides the infrastructure designer with a set of requirements that the supporting infrastructure

should meet. The infrastructure designer then takes this set of requirements and specifies an infrastructure that meets these requirements.

An infrastructure designer uses the computational viewpoint, the engineering viewpoint or both, to create the infrastructure design. The suitability of a viewpoint for the design of the infrastructure depends on which part of the infrastructure is designed and what requirements are met by that part.

The modelling entities of a computational design and an engineering design are related through correspondence relations. For example, a computational object corresponds to a BEO, an interface of a BEO corresponds to an interface of a computational object and the binding between two computational objects corresponds to a set of engineering objects. Figure 4-3 shows an example of two related views.

Figure 4-3 Related computational and engineering designs



The correspondence relation gives an infrastructure designer the choice to model parts of the infrastructure as computational objects and to map this specification to an engineering specification, according to the correspondence relations.

The deployment designer then takes the specifications of the application and infrastructure designers and packages the classes found in these specifications into components. This results in two types of components: infrastructure components and application components.

Infrastructure and application components are mapped to the implementation concept space using transformation rules.

4.1.3 Distributed system design with object middleware

In practice, the implementation of infrastructure components can be obtained from a vendor. Deployment of these infrastructure components on a set of nodes provides a distributed application with an object middleware layer.

According to the deployment view, a distributed system consists of application components that are deployed on a run-time environment. Internally, the run-time environment consists of Native Computing and Communication Environment (NCCE) and an infrastructure component. A node hosts the run-time environment and one or more application components. Nodes are interconnected through a transport network. Figure 4-4 shows a distributed system that consists of node A and B, which are connected through a transport network. Each node hosts a run-time environment and one application component.

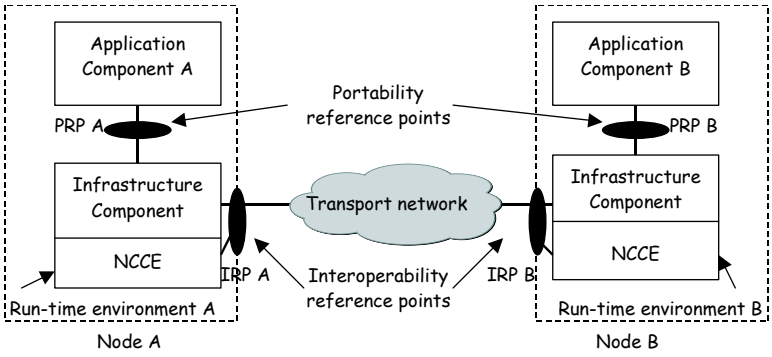


Figure 4-4 Deployment view of a distributed system

Since run-time environments can be obtained from different vendors, collaboration between run-time components and application components must be standardised. Therefore, a run-time environment has two reference points to which an implementation of the run-time environment should conform to offer portability and interoperability. The first reference point concerns the portability of application components; the second reference point concerns the interoperability of application components. The interoperability and portability reference points in Figure 4-4 are labelled IRP and PRP respectively.

Portability means that the run-time environment can be adapted to a variety of configurations, while a single application component can still be deployed on these various configurations. For example, a run-time

environment may vary over vendor, internal design or implementation language. Compliance to a portability reference point means that application components are agnostic to these variations.

Interoperability is the ability of two or more run-time environments to communicate and co-operate despite a variety of configurations. The variation in the configuration of a run-time environment concerns variations over internal design, vendor or implementation language.

4.2 Influences from early middleware platforms

This section presents some early middleware systems that have influenced the middleware systems of today. Some of these systems may not have been considered middleware at the time they were designed. However, today we categorise them as middleware systems as they provide a supporting infrastructure to an application designer. We discuss remote procedure calls (RPC), the V distributed system, ANSAware and OSF DCE. In fact, experience and knowledge gained from the design of these systems has been incorporated into current object middleware systems.

4.2.1 Remote procedure calls

The basic idea of an RPC is to extend the use of a procedure call to a distributed environment [BiNe84]. Most RPC systems aim to make the semantics of an RPC as close as possible to a local procedure call. RPC systems are concerned with binding, heterogeneity, call semantics and concurrency.

A local function call must by bound to a remote function. Binding can take place at design time, compile time or at run-time. The RPC system must ensure that the binding between a local function and a remote function is type safe to guarantee the integrity of the function call.

The RPC system is responsible to shield applications from heterogeneity aspects such as the use of multiple programming languages for procedure implementation, differences in byte-order due to different processor architectures and the use of various network protocols. A technique that RPC systems use to deal with heterogeneity is the use of stubs. A stub is a local program module that represents the remote procedure and shields an application from the mechanisms needed to use a remote procedure.

An RPC offers the semantics nearly identical to a local call. The only difference between the semantics of a local call and an RPC is the presence of network failures. An RPC system gives some degree of failure protection, by transparently re-issuing an RPC in case of a network failure. In case

re-issuing leads to multiple calls at the remote site, the RPC system suppresses duplicate calls and thus guarantees zero-or-one semantics. This is also referred to as at-most-once semantics. In addition, the RPC system maintains the call-by-value or call-by-reference semantics of the parameters of an RPC.

Finally, the RPC system manages concurrency aspects of an RPC. An RPC may be initiated concurrently from multiple threads. The order of RPC handling cannot be guaranteed. However, a calling thread is blocked until the RPC has completed.

4.2.2 The V distributed system

The V distributed system [Ch88] is a distributed operating system, designed for a cluster of workstations connected by a network. The system is structured as a small distributed kernel, a set of service modules, various run-time libraries and a set of commands. The kernel is distributed, i.e., each workstation executes a separate instance, but these kernel instances collaborate to offer a single abstraction of processes and associated address spaces.

The kernel provides a software backplane, to plugin software modules that can use the communication facilities provided by the kernel. The V distributed system is defined in terms of protocols and not in terms of predefined software specifications. Any network node that implements the system protocols can participate, independent of the internal software architecture.

High performance communication is considered the most critical facility of the V distributed system. A significant research effort has been spent on finding optimisations for interprocess communication (IPC). This has resulted in four contributions to an efficient interprocess communication system:

1. The kernel handles sending a request message and receiving a response message in a single send primitive. An application issues a request and then waits for the response to return, before it can continue processing. This reduces scheduling overhead and simplifies buffering.
2. Messages have a fixed size of 32 bytes with an optional and variable size data segment. Most messages fit into the fixed part of the message structure. The kernel interface, kernel buffering and network transmission have been optimised for this message size.
3. The VMTP transport protocol is used, which is optimised for exchange of request and response messages. The protocol has no explicit connection setup and teardown. Communication state for a client is established upon receiving the first request from that client. Duplicate messages are suppressed. The header of a VMTP message includes a

short fixed-sized message, thus supports efficient handling of small messages.

4. Every process descriptor contains a template VMTP header. Using this header, the overhead of creating a message as part of a send primitive is significantly reduced.

On top of the IPC system, a number of kernel services for time, process, memory, name and device management have been realised. These kernel services provide the basic framework for the realisation of various non-kernel services. These services include a pipe-server (implementing Unix-like pipes), an Internet server (implementing TCP/IP), a file server and a display server.

One of the lessons learned from the V distributed system, is that research should first focus on the design of protocols and interfaces of the parts of a distributed system. After the design meets the performance, reliability, security and functional requirements, the design should be converted to high-quality software [Ch88].

4.2.3 ANSAware

The Advanced Networked Systems Architecture (ANSA) advocates a common approach to distributed system design. The ANSA project has developed a set of common design principles. These design principles are categorised into concepts for describing distributed systems, design rules and implementation concepts.

Fundamental to the ANSA approach is the use of viewpoints. The enterprise, information, computational, engineering and technology viewpoints, as found today in the RM-ODP standards were developed in conjunction with the ANSA project.

ANSAware is the distributed application-programming environment produced by the ANSA project. It offers an abstract machine for the execution of the computational concepts of ANSA. These computational concepts do not require a new programming language, but are simply a set of constraints on a program, necessary to enable distribution. ANSAware programs are written using a standard programming language with embedded statements for (remote) interactions with other programs. These embedded statements are written in the Distributed Programming Language (DPL).

ANSAware has resolved a number of problems and simplifies the design of a distributed system. The issues resolved by ANSAware are (1) a language for the specification of interfaces, (2) automatic target language mapping, (3) late binding between client and server programs, (4) separation of

object and interface and (5) additional infrastructure services. Each issue is discussed in the sequel.

The ANSA Interface Definition Language (ANSA IDL) enables the designer of a distributed application to define the permitted types of interactions and the type of the data that can be included in these interactions. An ANSA IDL specification is similar to the specification of a set of operations for an interface of an object. ANSA IDL uses exceptions to deal with the failures due to distribution, such as network or remote host failure.

The mapping of ANSA programs to a target language is automated. Stubs are generated from an ANSA IDL specification. A pre-processor scans the source code to find DPL statements and converts these statements to code that calls the stub functions. Automated target language mapping enables distributed applications to be written in various programming languages.

The binding between a client program and a server program is established at run-time. A client program has no static reference to a server program and can obtain a reference just before it calls the server. This late binding enables a flexible deployment of ANSAware programs.

Objects are strictly separated from interfaces. An interface is considered to be a unit of structuring, whereas an object is a unit of distribution. An object can have multiple interfaces.

The run-time support offered to ANSAware programs is augmented with the trading service. The trading service is an infrastructure service, designed and implemented as an ANSAware application. The service enables server programs to export a reference to an interface to the trader. Client programs can then discover this interface reference based on some properties.

4.2.4 OSF DCE

The Distributed Computing Environment (DCE) is a standard from the Open Software Foundation (OSF). The OSF is an independent group for the support of the IT industry, with the goal to make open systems available to the industry. DCE exists since 1989 and has been developed for several years.

The technology comprises software services that reside on top of the operating system; DCE is a middleware that employs lower-level operating system and network resources. DCE enables organizations to distribute processing and data across the enterprise.

DCE was one of the first software solutions available from a vendor-neutral source that enables the development, usage and maintenance of

distributed applications in heterogeneous systems. DCE is available for many types of computing systems and operating systems.

Communication between DCE applications is based on RPC. DCE uses a language for describing interface definitions and has tools that automate the mapping to programming languages. In addition, DCE provides a number of services:

- Security Service -- authenticates the identities of users, authorizes access to resources on a distributed network, and provides user and server account management.
- Directory Service -- provides a single naming model throughout the distributed environment.
- Time Service -- synchronizes the system clocks of the computing systems in the distributed system.
- Threads Service -- provides multiple threads of execution capability.
- Distributed File Service -- provides access to files across a network.

The DCE RPC is an optional means for interaction between CORBA objects. However, DCE RPC is not widely used in CORBA systems, since the OMG has developed its own protocol for object interactions.

4.2.5 Observed concerns

Despite the many differences that can be found between the early RPC, the V distributed system, ANSAware and OSF DCE, these systems have a common structure for supporting interactions between software components. The four systems discussed before can be seen as generations of distributed systems that have contributed to a consolidated structure of the object middleware systems of today. The systems discussed have a number of concerns in common.

In all four systems, there is a clear *separation between interface and implementation*. The early RPC and V distributed system do not have an explicit definition of an interface, but do mention the need to separate the protocol definitions from the software implementation.

With ANSAware and later OSF DCE a language (*IDL*) for describing an interface is introduced. This language expresses the permitted interactions and the allowed types of interaction data. IDL definitions enable early validation of the (syntactic) compatibility of software components.

The mapping of remote calls to the run-time infrastructure is automated. The systems discussed have tools that can map IDL definitions to programming languages and network representations. These tools generate a *stub* to shield an application from the underlying mechanisms for (remote) interactions.

The systems discussed provide a means to map to various transport protocols. Applications are shielded from choosing a suitable transport protocol and managing connections.

Exceptions are the generalised way to deal with failures due to failures of the network or the remote host. If an interaction fails after a configurable number of retry attempts, the application receives an exception from the run-time system.

The V distributed system shows that performance benefits can be achieved by a fixed header size, with an optional and variable size data portion. The RPC mechanisms found in ANSAware and OSF DCE adopt this principle.

The RPC is the key mechanism for interaction between distributed software components found in all four systems. In ANSAware and DCE the RPC system is augmented with general purpose infrastructure services, such as the trader, time services, security services, etc. These services are general purpose in the sense that they can be reused in many application domains.

4.3 Support provided by contemporary object middleware

The design of a distributed application is simplified when an application designer can assume the availability of an infrastructure that supports possibly distributed application objects. Object middleware offers the supporting infrastructure to computational objects. The support offered by object middleware follows from the support that computational objects require. In any case, object middleware provides the functionality to implement the relative abstract notion of computational object binding and it provides the functionality to implement one or more of the distribution transparencies of the computational model

This section results in a set of features that must at least be supported by object middleware that complies with our reference model.

To arrive at this set of features, first the layers and functions found in CORBA and J2EE are reviewed. Then, based on these observations and the common concerns identified in the previous section and the distribution transparencies listed in Chapter 2, a set of object middleware features is listed.

4.3.1 CORBA layers and functions

A high-level overview of CORBA, its internal parts and the organisational processes that regulate the standardisation of CORBA have been discussed

in chapter 3. This section focuses on the layers and functions of the internal parts of CORBA, the CORBA services and the CORBA component model.

CORBA

The ORB is responsible for all of the mechanisms required to find the server object for the request, to prepare the server object to receive the request, and to communicate the data making up the request. The interface the client object sees is completely independent of where the server object is located, what programming language it is implemented in, or any other aspect that is not reflected in the object's interface [CORBA].

An ORB provides object invocation support, location transparency, access transparency and object lifecycle management.

The General Inter-Orb Protocol (GIOP) specifies a standard transfer syntax (low-level data representation) and a set of message formats for communications between ORBs. GIOP is designed to work directly over any connection oriented protocol.

The Internet Inter-ORB Protocol (IIOP) specifies how GIOP messages are exchanged using TCP/IP. The IIOP specifies how GIOP establishes and tears down TCP/IP connections and how TCP/IP connections are used to transport GIOP messages.

To exchange GIOP messages by means of another protocol than TCP/IP, the OMG has defined the extensible transport framework [ETS02]. This framework provides a set of interfaces that give an infrastructure designer the ability to create and insert a new transport protocol underneath an existing message distribution layer.

On the server side, CORBA defines the Portable Object Adapter (POA) that manages server objects. A POA maintains the relation between a server object and the actual code and data that implement the object. In addition, the POA manages the lifecycle of a server object and creates an object reference when a server object is instantiated.

A stub on the client side and a skeleton on the server side perform marshalling and demarshalling of remote invocations. CORBA also enables an application object to create a request at run-time, through the Dynamic Invocation Interface (DII). A server object can demarshall a request at run-time through the Dynamic Skeleton Interface (DSI).

CORBA services

The CORBA services specifications [OES01, ONaS02, OLS01, ONoS02, OTrS00] are the part of the Object Management Architecture (OMA) that define general purpose services for CORBA applications.

CORBA component model

The CORBA component model (CCM) [CCM01] defines a *component* as a basic meta-type, which is an extension of a CORBA object meta-type. A component is defined using CORBA IDL. Components are addressed by their component reference, which is in fact a specialised object reference.

The CCM component is a unit of instantiation, and a *component package* is a unit of deployment. So, the CCM component corresponds to a computational object, whereas a CCM component package corresponds to a component.

A CCM component package maintains one or more implementations of a component (section 69.1 – [CCM01]). It is represented by a software package descriptor and a set of files. The software package descriptor defines the properties of a CCM component package using an XML formatted structure. The software package descriptor consists of a generic part and a CCM specific part.

The CCM container provides the runtime environment of a CCM component. The container uses the POA, ORB and a set of CORBA services as supporting services and shields the application component from the use of these services. A set of configuration values, i.e., name-value pairs, configures the CCM container when it is created. The specification does not exhaustively define what these configuration values are.

At deployment time, the ComponentInstallation interface is used to install a component package. A deployment application calls the ComponentInstallation interface with a reference to the component package as a parameter. As a result, several objects related to the parts defined in the component packages are created. One of these objects is the CCM container.

4.3.2 J2EE layers and functions

High-level overviews of the Java 2 Enterprise Edition (J2EE), its internal parts and the organisational processes that regulate the standardisation of J2EE have been discussed in chapter 3. This section focuses on the layers and functions of the internal parts of Java RMI and J2EE.

Java RMI

The Java Remote Method Invocation (RMI) specification [RMI02] defines how an invocation of a Java server object that resides in another virtual machines is supported. RMI provides the mechanism by which client and server objects communicate. Details of communication between remote objects are handled by RMI, to the application developer remote communication looks just like standard invocations.

A stub on the client side and a skeleton on the server side perform marshalling and demarshalling of remote invocations. RMI server functions are provided by instances of RemoteObject and its subclasses. RMI provides lifecycle management by means of activatable objects. An Activatable object is a Java server object that is registered by an activation description at an activator object. Such objects are activated on an as-needed basis, thus saving resources on the server host.

RMI allows for native RMI and IIOP as alternative messaging protocols. Using IIOP as a messaging protocol enables CORBA objects to invoke Java objects and vice versa.

Native RMI conveys messages by the RMI transport protocol or by HTTP. Message transport through HTTP has been added to the RMI to let object invocations traverse through firewalls. An invocation supported by native RMI is conveyed using standard TCP/IP communication. However, if a firewall prevents TCP/IP connection establishment, the invocation is transparently conveyed by HTTP.

RMI defines an interface, called RMISocketFactory, which provides provides hooks for customisation of the socket object that RMI uses. Through this interface, customised sockets can be provided that enable alternative transports to be plugged into RMI.

J2EE general purpose object services

J2EE uses a number of general purpose services to support distributed applications. The Java Messaging Service (JMS) [JMS02] provides decoupled interactions and one-to-many interactions. The Java Naming and Directory Interface (JNDI) [JNDI01] is an interface specification for naming of objects. JNDI provides an interface with directory and naming functionality to Java applications.

Enterprise Java Beans

The Enterprise Java Beans (EJB) specification defines an architecture for distributed object computing. An EJB component corresponds to a computational object. EJBs are packaged in an EJB ARchive (EAR).

An EAR contains the binary representation of one or more EJBs and a deployment descriptor. The deployment descriptor provides structural information of the EJBs, such as supported interfaces and external dependencies, and it provides assembly information, such as how client and server interfaces of EJBs should be bound. An EAR is a unit of deployment and therefore corresponds to a component.

The run-time environment for an EJB is the EJB container. The run-time environment for an EAR is an application server. The application server uses general purpose services, such as JNDI and JMS as supporting services and shields the EJB component from the use of these services.

4.3.3 Observed concerns

CORBA and J2EE are contemporary object middleware platforms that reveal some common concerns that are needed to support distributed object applications.

Just as with the early middleware platforms, both platforms clearly separate between interface specification and implementation. In CORBA an application designer uses OMG IDL to specify an interface, in J2EE the Java keyword *interface* is used.

An application designer that uses these object middleware platforms does not have to design the means to support interactions between possibly remote objects. In CORBA the ORB supports object interactions, whereas Java RMI provides this support in J2EE.

Object interactions are conveyed as messages between peer entities. CORBA defines the GIOP protocol and J2EE uses native RMI or HTTP to convey messages. Application designers are shielded from managing the connections needed by these message protocols. CORBA allows the transport protocol that GIOP uses to be replaced. The default transport protocol is TCP/IP. The combination of GIOP and TCP/IP is called IIOP. IIOP defines interoperability rules between ORB implementation of different vendors. Both platforms allow an infrastructure designer to replace the default messaging functionality with other messaging functions.

Both platforms shield application designers from differences in the representation of application data that may result from differences in hardware architectures of computing nodes. Consequently, access transparency is provided.

Support for object lifecycle management is provided. CORBA defines the POA as a standard object lifecycle manager whereas an activator object provides this functionality for J2EE.

Naming and directory services are defined for both platforms as general purpose infrastructure services. These services offer increased support for location transparency. To locate a server object based on a name is supported by a naming service in CORBA and the JNDI in J2EE.

Decoupled communications, one-to-one and one-to-many communications, is also supported as a general purpose service. This support is provided by the event or notification service in CORBA and the Java Messaging Service in J2EE.

Each platform defines a component model, which provides support for deployment. The environment where components are deployed is called the container for both J2EE and CORBA.

4.3.4 Supported features

The following list of features must at least be supported by object middleware that complies with our reference model:

- Object invocation support: a client object must be able to invoke a server object so the object middleware must ensure that invocations are delivered;
- Access transparency support: objects can be instantiated on computing nodes constructed from heterogeneous hardware and the networks that connect these computing nodes may also be heterogeneous, so the object middleware is responsible to deliver invocations despite this heterogeneity.
- Location transparency support: objects can be geographically distributed so the object middleware is responsible to deliver invocations to the proper location while hiding the location of a server object from a client object;
- Decoupled interaction support: objects must be able to interact with each other in a decoupled way, i.e. without waiting for a reply, so the object middleware must support decoupled interactions;
- One-to-many interaction support: an object must be able interact with many objects in a single action, so the object middleware must ensure that interactions are delivered to multiple objects;
- Object lifecycle management support: objects must be created, activated, deactivated and destroyed during their lifetime, so the object middleware is responsible for maintaining the lifecycle of an object;
- Deployment support: classes from which objects are instantiated are packaged into a component that must be deployed in a configurable run-time environment that enables parameterisation of component through a deployment descriptor, so object middleware should support configuration and enable parameterisation of a component.

The object middleware reference model constructed in this chapter must be sufficiently powerful to support these features. Later on we show that current object middleware systems that correspond to our model generally comply with these features.

Our model does not consider other distribution transparencies, such as failure, migration, relocation, replication and persistence transparency. Support for these transparencies is only found in dedicated object middleware systems and therefore these transparencies are not taken into account as features supported by our reference model. Supporting these transparencies would make our model less generic.

4.3.5 Feature sets

Historical developments have influenced the internal structure of an infrastructure component. The portability reference point of an infrastructure component changes over time as a new generation of an object middleware system becomes available that supports additional features. Older generations of object middleware systems may support only a subset of the features identified in section 4.3.4.

To relate our object middleware reference model with current and older generation object middleware platforms, we identify three parts. Each part meets a subset of the features identified in section 4.3.4. The subsets of features are labelled CFS, SFS, EFS.

The three subparts are *object communication middleware*, *general purpose object services* and the *component execution environment*. The object communication middleware provides support for object invocations, location transparency, access transparency and limited support for object lifecycles (CFS). The general purpose object services provide support for decoupled interactions, one-to-many interactions and increased location transparency (SFS). The component execution environment offers extended support for object lifecycle management and deployment (EFS).

Some of these subparts depend on others. The general purpose object services depend on the object communication middleware. The component execution environment depends on the general purpose object services and on the object communication middleware.

The structuring of object middleware related to feature sets is depicted in Figure 4-5

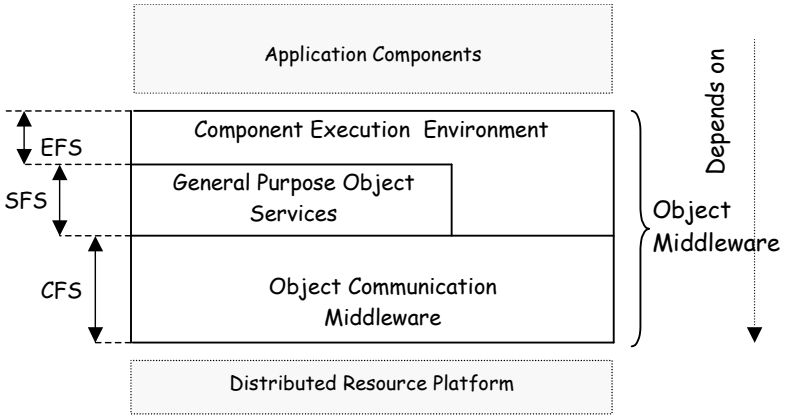


Figure 4-5 Structure of object middleware related to feature sets

How each of these subparts offers support in accordance with their related feature set is discussed in the sequel.

4.4 Object communication middleware

The *object communication middleware* relates to software that enables objects to communicate with each other, irrespective of their location, the computing environment they are deployed on, or the network that is used for data transport. The object communication middleware layer hides all the heterogeneity aspects with respect to remote object interactions. This includes the common concerns such as representation of interaction data in a common format, type checking of interaction data, the binding of programming language specific method invocations to a remote function, exception delivery in case of failures and mapping to the transport facilities of the underlying distributed resources. The permitted interactions and allowed interaction data types supported by the object communication middleware layer must be described in an interface definition language (IDL).

4.4.1 Engineering view

According to the engineering view on a distributed system, the system parts are modelled as engineering objects. A stepwise refinement of the engineering specification results in the structure of our object communication middleware model. The engineering model constructed in this section complies with the feature set CFS as identified in the previous section.

The high-level engineering specification of the object communication middleware shows this middleware as a *channel* that supports the interactions between basic engineering objects (BEO). Figure 4-6 shows this channel. This figure also shows that the interfaces between the BEO and the object communication middleware correspond to the portability reference points as discussed in section 4.1.3.

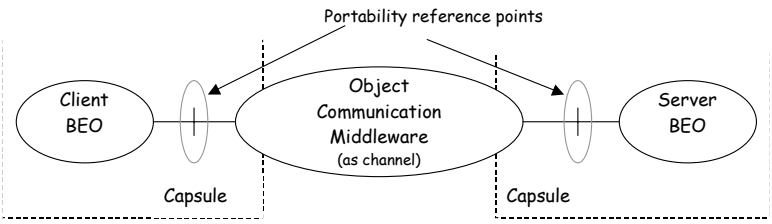


Figure 4-6 High-level engineering view

An interaction between a client BEO and a server BEO is composed of a request and a reply interaction. As discussed in chapter 2, a request interaction consists of a submit action at the client interface and a deliver action at the server interface. The reply interaction consists of a submit action at the server interface and a deliver action at the client interface. To support an interaction, the object communication middleware conveys interaction data as a message. These observations lead to a refined engineering view.

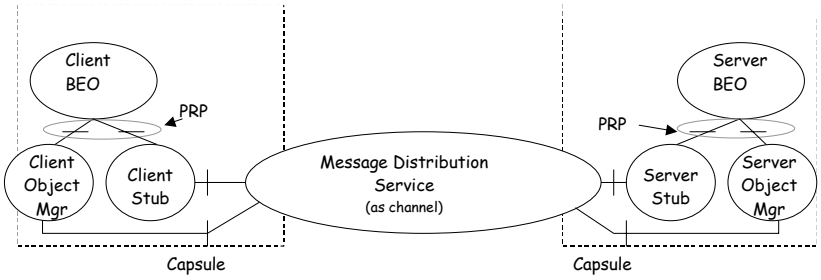
In a refined engineering view, the object communication middleware is refined into *stub objects*, *object managers* and a lower level channel that supports *message distribution*.

A client stub object when invoked by a client BEO converts a request in a message containing the request and its parameters. The message distribution service is responsible to transparently convey the message to the remote stub.

The object manager manages the lifecycle of the BEOs. It ensures that a client BEO can address a server BEO by means of an object reference. We define an object reference as a pointer to a server BEO that can be used anywhere in the distributed system to address this BEO. The object reference is created on the server side when an object is created and a client BEO uses it to address the server BEO.

The portability reference points are now covered by the interfaces of the stub and the object manager. Figure 4-7 shows these reference points and how the object manager, stub and message distribution service are related.

Figure 4-7 Refined engineering view



A client stub on the sending side of the message distribution service is closely related to a server stub on the receiving side. The marshalling of invocation parameters on the sending side must be reversed on the receiving side. Therefore, the stubs have a common set of marshalling and demarshalling rules.

The object managers on the client and server side are also closely related. The object reference created on the server side must be interpreted correctly by the object manager on the client side to ensure that the message distribution service sends messages to the capsule in which the server BEO resides.

A further refined engineering view reveals the internal structure of the message distribution service. This service sends a request message from the client capsule to the server capsule. After the server BEO has produced a reply and the stub has marshalled the parameters of that reply, the message is returned to the client capsule. It is the responsibility of the message distribution service to maintain the relationship between request and reply messages.

Messages are transported by a *transport service* that offers a connection oriented reliable transport service. Connections of the transport service are established and teared down by a *messaging object*.

On the client side a client messaging object receives a request message from a stub. The client messaging object determines the destination address of the message. The destination address is obtained from the object manager, which knows how to extract address information from an object reference. The messaging object also adds an identifier to the message to enable the remote messaging object to associate a reply message with its request message.

The messaging object creates a connection using the transport service. Connections may be reused for multiple invocations between the same client and server BEO. Connections are teared down by the messaging object on the client side, on the server side or by the transport service itself. In case the messaging object aims to reuse connections, it will reuse a connection that already exists as the result of a previous request.

A message received at the server side is forwarded to the object manager. The object manager knows how a destination address is related to an instance of a server BEO. From this relationship the stub that is associated to a server BEO is derived. The messaging object removes the information added on the client side from the message and offers the request message to the stub for demarshalling.

Figure 4-8 shows these how the messaging objects and the transport service are related.

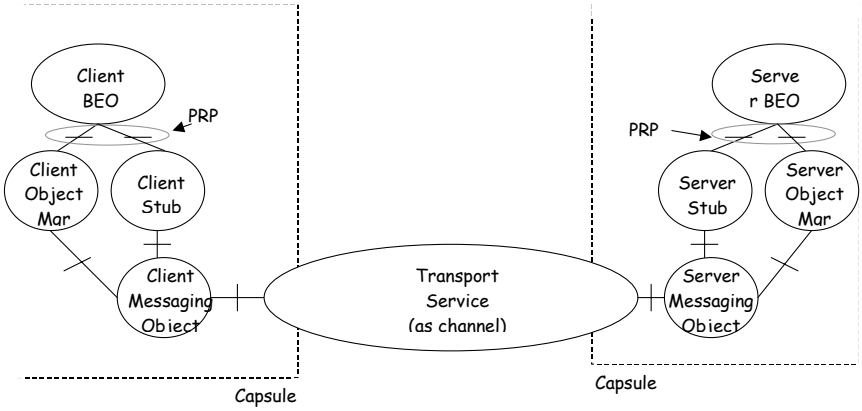


Figure 4-8 A further refined engineering view

Yet another refined view reveals the internal structure of the transport service. The reliable connection oriented service of the transport service may not be offered in all cases by the transport network. Therefore a *transport adaptor object* is responsible to leverage the service offered by the transport network.

Figure 4-9 shows how the messaging, transport adaptor objects and the transport network are related. This figure also reveals the interoperability reference points.

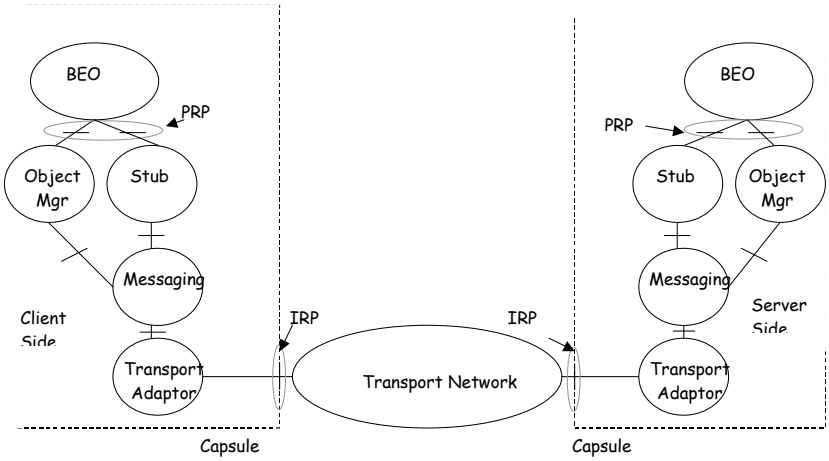


Figure 4-9 Most refined engineering view

Each of the object manager, stub, messaging and transport adaptor objects in the client capsule has a relation with their peer objects in the server capsule. For example, the stub objects are peers in the sense that the encoding rules on the client side must be understood by the stub on the server side to decode request parameters. In the same way the messaging objects are peers because the formatting of a message created on the client side must be understood on the server side.

The peer transport adaptor objects together offer a transport service to the messaging objects. The peer messaging objects together offer a messaging service to the stub objects. The peer stub objects and peer object manager objects offer object interaction services to the BEO objects.

4.4.2 Object communication middleware layers

The stacking of peer objects shown in Figure 4-9 results in a layered structure for the object communication middleware. The horizontal layering of the object communication middleware results in three layers: the object interaction layer, a message distribution layer and a transport adaptation layer. Figure 4-10 shows how the engineering objects are related to the internal layers of the object communication middleware.

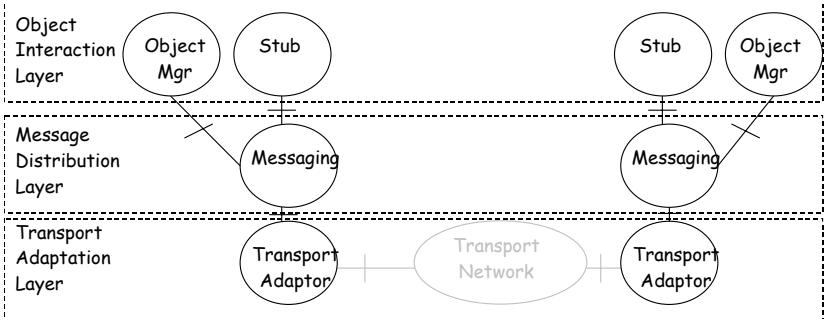


Figure 4-10 Object communication middleware layers

Object interaction layer

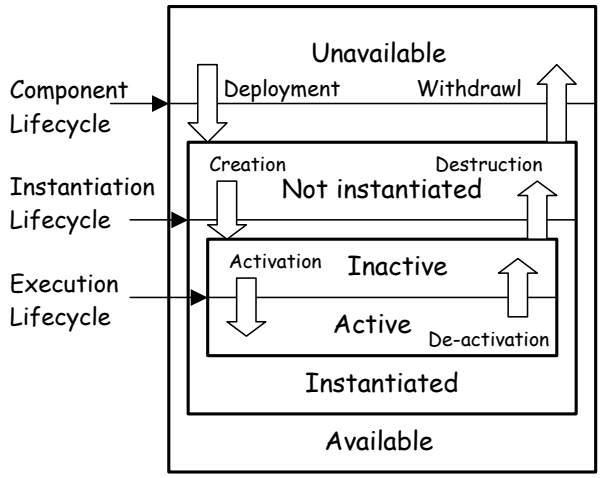
The object interaction layer offers distributed objects the services needed to interact with each other despite differences induced by the heterogeneous distributed resource platform. The object interaction layer uses stubs to offer a local interface to a possibly remote object. The stub is specific for a particular programming language and is usually generated from an IDL specification. The stub hides the marshalling and unmarshalling of parameters from a client object. Marshalling concerns the placement of interaction data in a message such that it can be conveyed across the

network. Differences in byte-order due to different CPU architectures on the client and server side are resolved

In addition, the object interaction layer is responsible for the management of object references. The object manager on the server side creates an object reference. An object reference consists of one or more transport addresses that are understood by the transport adaptor and an identifier that uniquely identifies a server object within its capsule. An object reference is opaque to the client BEO. A client side object manager knows how to interpret an object reference.

An object manager creates an object reference when a server BEO is bound to the object interaction layer, i.e., a server BEO makes its interface remotely accessible. Resources for processing, storage and communication are scarce. An object requires resources in order to execute, but objects bound to the object interaction layer do not require these resources all the time, so scarce resources can be shared between multiple server BEOs. Consequently, lifecycle management functions are needed to assign resources to an object. Lifecycle management functions allow objects to be registered, instantiated, activated and deactivated. The life cycle of an object consists of two nested cycles: the instantiation and the execution life cycle. The first life cycle relates to publishing an object reference and the second life cylce relates to object activation. Figure 4-11 shows the nesting of these life cycles.

Figure 4-11 Nested life cycles



A component clusters related classes into a more coarse grained unit of deployment. An object is instantiated from its class that is contained within a component. Until a component is deployed, the object is unavailable. When a component is deployed in its run-time environment, the object

becomes available and may then be created using its class. However, an object that is created exists as a virtual entity, i.e., no resources have been assigned to it and the object is inactive. At this stage, the object communication layer only maintains an object reference to the newly created object. When the object is activated, as part of its execution lifecycle, resources for communication, storage and processing are assigned to the object. Conversely, an object can be de-activated while its object reference is still valid. An object reference becomes invalid when an object is destructed, i.e., at the end of its instantiation lifecycle.

Distinguishing the execution life cycle from the instantiation lifecycle enables the object interaction layer to efficiently assign resources to objects. A huge number of objects can be created, while only a limited subset of these objects actually requires storage, processing and communication resources at some point in time.

Message distribution layer

The *message distribution layer* supports the object interaction layer with a message distribution service. Object interactions between remote objects require the transport of request and reply messages. The message distribution layer is responsible to locate the transport endpoints that are used on the server side to receive request messages. A server object may have multiple transport endpoints, in which case the message distribution layer chooses a suitable endpoint. This choice may be directed by policies given by the application.

Following the choice of a suitable transport endpoint, a transport association between a client and a server object is established. Once this has been established, messages are exchanged between that client and server using that transport association. The message distribution layer ensures that the relation between a request and a reply message is maintained so that a client object receives a reply that belongs to a request issued earlier.

An object interaction may carry a large amount of data, which can be more efficiently transported when fragmented into a set of smaller messages that are reassembled at the receiving side. To support this a message distribution layer may perform fragmentation and reassembly of large messages.

Transport adaptation layer

The *transport adaptation layer* adapts the systems specific services of the distributed resources to the needs of the layer above. Since the distributed resources are potentially heterogeneous, the transport network may offer different levels of service. The transport service can for example be connection oriented, connectionless, reliable or unreliable. The transport

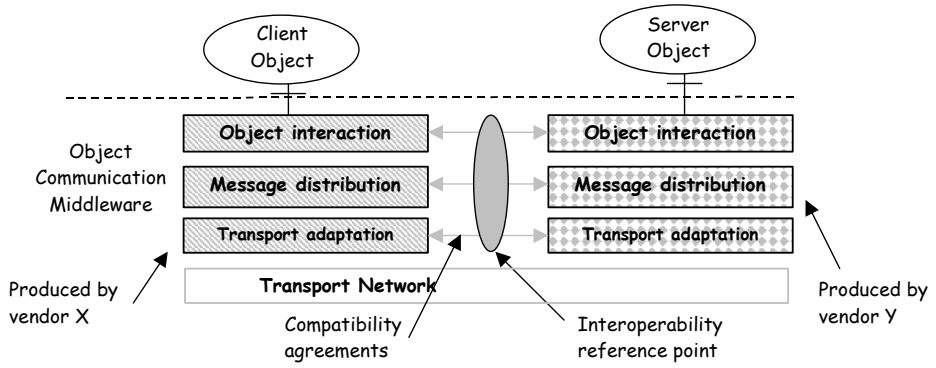
adaptation layer leverages the system specific transport services to a transport service that reliably transfers data.

4.4.3 Interoperability concerns

One of the prime purposes of object communication middleware is interoperability between software components developed for different hardware platforms and with different implementation languages. Ideally, the object middleware implementation would be obtained from various vendors and components deployed on these implementations should be able to collaborate. Interoperability of two object middleware implementations concerns agreement on the rules for collaboration between these implementations. Each of the layers of the object communication middleware contributes to the interoperability rules that define the interoperability reference point.

Figure 4-12 shows how an interoperability reference point is used in a situation where multiple vendors provide an object communication middleware implementation. The interoperability reference point is defined between the three layers of the object communication middleware. This enables a client object instantiated on an implementation produced by vendor X to interact with a server object instantiated on an implementation produced by vendor Y.

Figure 4-12 An interoperability reference point



In case a third party produces one or more of the layers of the communication middleware, the set of interoperability rules can be relaxed. This situation is depicted in Figure 4-13, where producer Z delivers the transport adaptation layer. There is no need for interoperability rules between different transport adaptation layer implementations as the implementation of this layer is within the realm of producer Z. However, a portability reference point must be defined between two layers.

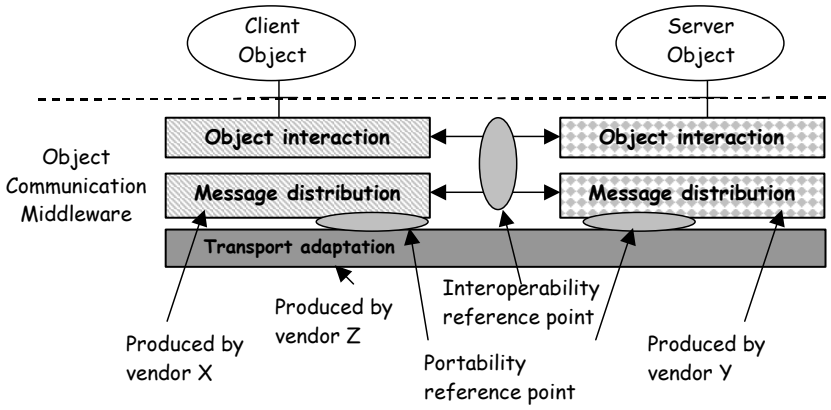


Figure 4-13 A portability reference point

The introduction of an additional portability reference point between layers within the communication middleware reduces the set of interoperability rules. A simplified interoperability reference point allows extension of the communication middleware with a specialised transport adaptation layer. With these extensions the object middleware can benefit from specific characteristics of the network, such as QoS features, without compromising the features that the object communication middleware must support.

Enabling QoS features of a transport network may dictate that a single vendor produces a layer. Therefore portability reference points between communication middleware layers should be standardised.

4.4.4 Offered support

The three layers that constitute the object communication middleware together provide support for a subset of the features, i.e. CFS, identified in section 4.3.4.

The object interaction layer offers support for *object invocations*. This layer ensures that a client invocation is delivered to a server object and that the response of the server is delivered to the client. A reply message is related to its associated request message by the message distribution layer. The object interaction layer produces an opaque object reference that hides the location of a server object and thus offers *location transparency*. The message distribution supports the object interaction layer in providing location transparency by establishing a transport connection between the location of the client and the location of the server. However, an object reference must contain addressing information that the message distribution layer extracts to establish a transport connection. As a result

location transparency is limited, since the transport address reveals the physical location of a server object.

Marshalling and unmarshalling functions ensure that invocation parameters conveyed between a client and server are independent of the programming language in which client and server objects are developed. The transport adaptation layer ensures that heterogeneity of underlying transport networks is shielded from the message distribution layer. As a result the collective services of the message distribution layer and transport adaptation layer provide *access transparency* support.

The object interaction layer also offers support for *object life cycles*, through the management of object references and the decoupling of the instantiation life cycle from the execution lifecycle.

The message distribution layer offers support for *decoupled interactions* in case it supports store and forward of messages. If this is supported then a client object can invoke an object without waiting for a reply and obtain the result of the invocation later. The message distribution layer can store a message at the server side or at the client side.

4.5 General purpose object services

One or more distributed objects that enhance the service offered by the object communication middleware, are offering an *object service*. In case these distributed objects offer a service that useful for a large set of distributed applications and an application designer considers that service fundamental to the design of a distributed application, such a service is called a general purpose object service.

Examples of general purpose object services are event service, licensing service, persistent state services, property service, time service, naming service. Table 4-1 shows a brief description of each of these services.

Table 4-1 Examples of general purpose services

Service Name	Description
Event Service	Defines two roles for objects: supplier and consumer. Suppliers produce event data, and consumer process event data.
Licensing Service	Provides support for the licensing of software artefacts.
Naming Service	Provides the mechanism to locate objects based on a logical, location independent name
Persistent State Service	Provides support for persistent storage of the state of an application object.
Property Service	Provides the ability to associate named values with application objects.

Time Service	Provides a general clock interface to obtain the local system time in a standard format.

This section discusses those general purpose object services that leverage the support of the object middleware to include the feature set SFS. An object middleware that corresponds to the model presented in the previous section and the model of the general purpose services discussed in this section complies with the feature sets CFS and SFS.

General purpose object services use the object communication middleware as a supporting infrastructure for (remote) interactions. The computational viewpoint is the most appropriate view to explain the general purpose object services, as this viewpoint suits an application designer and it simplifies the integration with application specific objects. In the remainder of this section the computational view of the naming service and the event service are discussed.

4.5.1 Computational view of the Naming service

The Naming Service allows a human readable name to be associated or *bound* to an object. The reference to that object can subsequently be found by *resolving* that name within the Naming Service. Using the Naming Service a name is bound to an object relative to a naming context. Different names can be bound to an object in the same or different contexts at the same time, this is called a name binding. A naming context is an object that contains a set of name bindings in which each name is unique. In file management terms, a naming context resembles basically a directory structure for objects. A name is always resolved relative to a context; there are no absolute names. To resolve a name is to determine the object associated with the name in a given context. To bind to a name is to create a name binding in a given context.

Because a context is an object like any other object, it can also be bound to a name in a naming context, thus creating a naming graph. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a compound name) defines a path in the naming graph that directs the resolution process. The naming service provides the principal mechanism through which most client objects locate server objects that they intend to use. Given an initial naming context, client objects navigate naming contexts by retrieving lists of names bound to that context.

A server registers an object reference with the Naming Service by binding the object reference to a naming context. This name can then be used by other components in the system to find the registered object.

The design of a naming service as a set of naming context objects, allows these naming context objects to be distributed over several nodes. The benefits of such a distribution, such as resilience against partial failures and load sharing, have been shown for a pan-European object middleware platform [HTW98].

4.5.2 Computational view of the Event service

The event services enables an object to send an event to many other objects. An object that sends an event is called a producer and an object that receives an event is called a consumer. Event production is decoupled from event production through an event channel. A consumer can consume an event at a later stage than when the event has been produced, even when the producer has already been deactivated. Event consumption is not acknowledged to a producer.

The initiative for event production can be at the producer object or at the event channel. In case the initiative lays with the producer, the event is pushed to the event channel, otherwise the event is pulled from the consumer. On the consumer side events can also be pushed by the event channel or pulled by the consumer. The computational objects that constitute an event service are depicted in Figure 4-14.

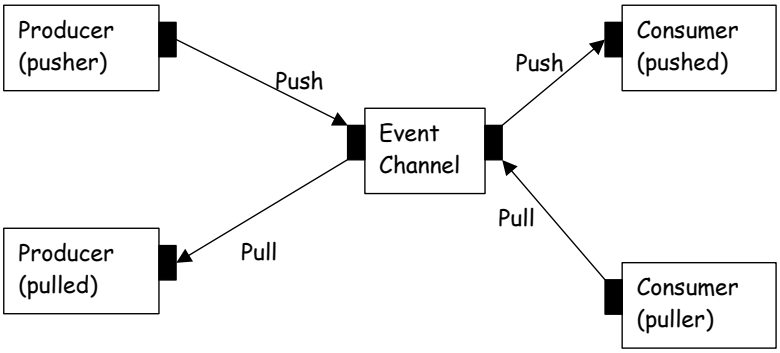


Figure 4-14 Computational description of an event service.

The role of a consumer and a producer with respect to an event channel can be asymmetric. For example, in case a producer pushes events to an event channel these events can be pushed to some consumers while other consumers pull the event.

The event service decouples producer objects from consumer objects and enables one to many communication of events.

4.5.3 Offered support

The general purpose object services offer additional support to distributed applications. Services that are reusable for many applications such as Naming, Persistent State or Time services are examples of general purpose object services. In some cases general purpose object services offer a standardised interface to a complex distributed service whose implementation requires the knowledge of a specialist. An example of such a service is the Persistent State service, which coordinates persistent storage of the internal state of application objects. An application designer uses general purpose object services as proven building blocks that simplify the design of distributed applications.

Support for *decoupled interactions* is offered through the Event service. Although decoupled interactions can also be offered by the object communication layer, in some cases this support is not available at this level. The Event service enables decoupled interactions in case the object communication middleware does not support this. It is a design choice of an application designer to determine which solution is most suitable.

The Event services also offers support for *one-to-many* interactions. This enables one application object to produce events for many interested consumers of these events.

The support for *location transparency* is further extended by the Naming service. This service enables an application object to lookup an object based on a name or discover an object based on a set of properties of that object. This hides the location of a server object completely from a client object.

4.6 Component Execution Environment

From the perspective of an application designer, the object communication middleware in conjunction with the general purpose object services offers sufficient support for distributed object applications and meets all but one of the features of section 4.3.4. However, from the perspective of a deployment designer, additional support for the deployment of components is needed. This support concerns the configuration and parameterisation of a component at deployment time.

Configuration and parameterisation of an application component should be left to the deployment designer, however in practice an application designer can easily mix these concerns with the computational design. We review three examples that together demonstrate how these concerns can be mixed:

– The first example concerns a computational object that acts as a factory for other computational objects. As a policy, the factory immediately

activates each object that it instantiates. This policy is part of the behaviour of the factory object as defined by the application designer and is therefore embedded within the factory object. As a result, lifecycle management of objects is statically configured within the computational design and thus the application designer implicitly assigns resources to objects. In this case, it is not possible for the deployment designer to control the object lifecycle policy once the application classes have been packaged into a component.

- The second example concerns the configuration and use of an event channel. To use an event channel it is required that the channel is created and that producer and consumer objects connect to the channel. The lifecycle of an event channel and the associated producers and consumers of that channel must be controlled. Channel configuration from a computational object makes the use of the channel and its producers and consumers application specific. Future use of this computational object in the context of other computational objects is limited as the channel establishment and connecting producers and consumers is embedded in the computational design. In this case, it is not possible for the deployment designer to control channel configuration once the application classes have been packaged into a component.
- The third example concerns the use of the naming service. Consider a computational object that registers itself with a naming service and the name it uses for registration is embedded within that object. Again, we see configuration concerns mixed with application concerns. The computational server object has a hard-coded name that client objects must use to resolve the server object reference from the naming service. In this case, it is not possible for a deployment designer to externally control the name that the server object uses to register itself with the naming service. Even if such a control interface is provided, it is application specific and not for general use at deployment time.

These examples show that computational objects can easily be designed with embedded configuration actions. As a result, application components constructed from the classes of these application objects can only be deployed as parts of a dedicated distributed application. Reuse of such components in future distributed applications becomes restricted. Separating application logic from deployment configuration requires object middleware to offer standard interfaces for the deployment configuration of a component.

The *Component Execution Environment* offers a run-time environment for the deployment of components and configures components according to a deployment descriptor. The component execution environment separates

deployment and configuration from application logic, by shielding components from the communication middleware and general purpose object services. It offers interfaces for the deployment designer to configure a component according to the needs of a distributed application. In addition, it manages the lifecycle of objects (i.e., activation and deactivation of objects), the processing and storage resources (including storage and retrieval of object state to non-volatile storage), and possibly the transaction and security context of object interactions.

A component execution environment is closely associated with a particular component model. This means that it offers an environment for the execution of components constructed according to that component model. A component model prescribes standard interfaces for the registration and manipulation of components.

4.6.1 Engineering view of a component container

The component execution environment (CEE) offers a deployment designer the environment for the deployment of components. The CEE consists of *component containers* that are the run-time environment for one or more components. A container offers a set of interfaces that simplify the deployment and configuration of a component. A container shields a component from the underlying communication middleware and from a set of general purpose object services. Functions of a container typically include:

- Automatic lifecycle management of objects to preserve limited system resources, such as main memory.
- Adaptation of a set of general-purpose object services. A container typically provides an adaptation layer to services that provide transaction, security and event notification. This adaptation layer frees the application designer from locating, initializing and configuring these services.
- Adaptation of interesting events from the communication middleware and the general purpose object services for use by a component. For example, a container can manage the events associated with a transaction. This frees a component from handling these events and enables a component to be involved in a transaction while the component has no application code for transaction management. The container can be configured to report only transaction failures to a component as a standard exception.

A container uses a deployment descriptor to configure the run-time properties of a component. An infrastructure designer is responsible for the internal structure of a container. A deployment designer uses the interfaces

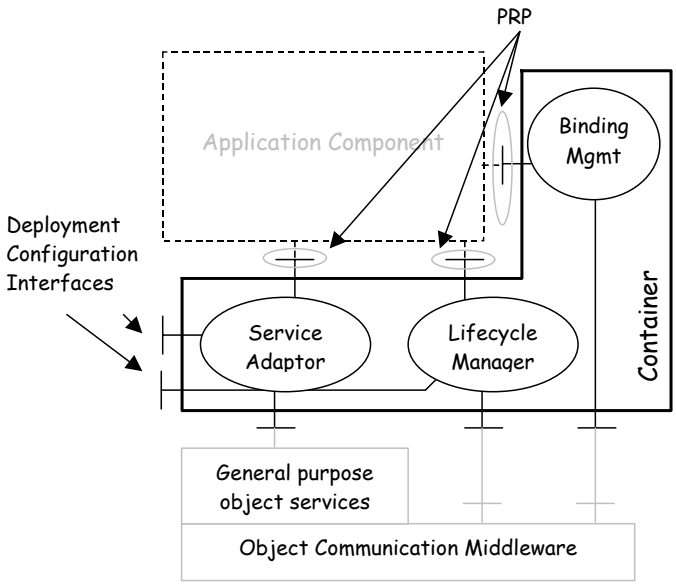
that an infrastructure designer prescribes as a means to realise the desired run-time properties of a distributed application.

A container exposes a number of interfaces to an application component. These interfaces constitute the portability reference point for a component execution environment. The container adapts these interfaces to the services offered by the object communication middleware and the general purpose object services. In addition, a container exposes a set of interfaces for deployment configuration.

Internally, a container typically has a service adaptor object, a life cycle manager and a binding management object. The service adaptor object adapts the general purpose object services to the needs of an application component and according to the configuration settings that are received through its deployment configuration interface. The lifecycle manager manages the instantiation and execution lifecycle of application objects. The binding management objects establishes bindings between client and server objects using the object communication middleware.

Figure 4-15 shows which interfaces a container exposes and how a container relates to object communication middleware and general purpose services.

Figure 4-15
Engineering view of a container



The CEE completes the object middleware reference model. The object middleware reference model reveals several layers and subparts that constitute object middleware. Each of these layers and their subparts are

needed to satisfy the requirements of application and deployment designers.

4.7 Evaluation and conclusion

This section assesses the correspondence of our generic object middleware reference model with CORBA, J2EE and Web services. Conclusions regarding our reference model are drawn.

4.7.1 Correspondence with CORBA, J2EE

CORBA

An ORB corresponds to an infrastructure component. A set of ORBs connected through a transport network offer an object communication middleware. GIOP corresponds to the message distribution layer in our reference model. The set of GIOP messages are a part of the interoperability reference points between ORBs.

The adaptation of GIOP connection management to TCP/IP connection management corresponds to the transport adaptation layer. The extensible transport framework corresponds to an interoperability reference point between the message distribution layer and the transport adaptation layer.

A POA corresponds to a server object manager in our reference model. The CORBA stub and DII correspond to the client stub, whereas the skeleton and DSI correspond to the server stub in our reference model. The DII, DSI, stub, skeleton, POA and ORB interface define the portability reference point for CORBA 2.x implementations. These entities collectively correspond to the object interaction layer in our reference model.

The CCM container constitutes the portability reference point as defined for the component execution environment of our model.

J2EE

An implementation of the RMI specification corresponds to the object communication middleware in our model.

This corresponds to the nested lifecycles defined in our reference model. The interfaces provided by the stub, skeleton, Activatable object and RemoteObject correspond to the object interaction layer in our reference model.

The internal structure of RMI contains a messaging layer that corresponds to the message distribution layer our reference model. The IIOP specification defines an interoperability reference point between Java RMI and CORBA. RMI defines an interface, called RMISocketFactory,

which provides a portability reference point for the transport adaptation layer of RMI.

JMS and JNDI are examples of services that correspond to the general purposes services in our reference model.

An application server corresponds to a container in our reference model. The EJB container interfaces constitute the portability reference point as defined for the component execution environment of our model.

4.7.2 Correspondence with Web services

The term "web services" is used loosely to denote a collection of (related) technologies. These include:

- SOAP (Simple Object Access Protocol) [SOAP01] – "an emerging distributed middleware technology that uses a lightweight and simple XML-based protocol to allow applications to exchange structured and typed information across the Web"
- WSDL (Web Services Description Language) [WSDL01] – an XML-based language to describe web services "interfaces"
- UDDI (Universal Description, Discovery and Integration) [UDDI] / WS-Inspection (Web Services Inspection Language) [NaBa01] – Service description and discovery mechanisms.

The use of these technologies for the realisation of a distributed system ensures interoperability between services offered over the web. A web service corresponds to a computational object in our model.

Correspondence to the object communication middleware

The SOAP specification defines how a client of a web service invokes this service. A client stub in our reference model corresponds to a service proxy and a server stub in our model corresponds to a service implementation template. SOAP messages are formatted as XML data structures, which are structured according to the interface definitions described in WSDL. SOAP corresponds to the object interaction layer in our reference model. However, SOAP lacks the notion of an object manager and does not provide the mechanisms for managing the life cycle of a web service.

Message distribution for web services is mostly based on HTTP, although other message distribution layers such as SMTP are also allowed. The message distribution layer takes the XML formatted messages and conveys these messages from client to server side and vice versa.

Since SOAP depends on XML as means to structure SOAP messages and the interfaces to manipulate an XML structure are standardised, in the DOM [HHW+00] and SAX [SAX98] specifications, it can be argued that

DOM and SAX define alternative portability reference points for web services.

Correspondence to the general purpose object services
UDDI is a specification for a general-purpose service, which corresponds to a trading service in our reference model.

Currently, the web services specifications do not define any entities that provide decoupled interactions, such as one-to-many and many-to-many interactions. No corresponding entities to an event service can be found.

Correspondence to the component execution environment
Support for deployment, such as the specification of an entity that corresponds to a container in our reference model, is not (yet) provided by the web services specifications.

4.7.3 Summary

Table 4-2 summarises the correspondence between notions in our object middleware reference model and CORBA, J2EE and Web services.

Table 4-2 Correspondence between notions in our reference model and contemporary object middleware platforms

Reference model notions	CORBA	J2EE	Web services
Communication middleware	ORB	Java RMI	
Object interaction layer	Stub, skeleton, POA	Stub, skeleton, Activatable object, Remote Object	SOAP (although without support for life cycle management)
Message distribution layer	GIOP	Native RMI, HTTP or IIOP	HTTP or SMTP
Transport adaptation layer	IIOP	RMISocketFactory (for native RMI only)	- (integrated with message distribution layer)
General purpose services			
Naming service	Naming service	JNDI	UDDI
Event service	Event or Notification Service	JMS	-
Component execution environment	Component server	EJB server	-
Container	CCM container	EJB container	-

4.7.4 Conclusion

This chapter shows that stepwise refinement, as used for the structured design of a distributed system, stops when either the parts of a design are readily available in the implementation concept space, or when the parts of a design can be generated using transformation rules.

An object middleware platform is a supporting generic infrastructure, which is independent of a specific distributed application. Distributed system design benefits from the use of object middleware.

An application designer that refines a computational design of a distributed application and that directs this design towards object middleware, benefits from the readily available functionality of object middleware platforms in the implementation concept space. Object middleware provides functionality for the relative abstract notion of object binding and it provides the functionality to implement one or more of the distribution transparencies of the computational model. In any case, object middleware provides the mechanisms needed to overcome problems of distribution.

A few specialists can focus on the design of the mechanisms needed to overcome problems caused by distribution. This is cost-effective as these mechanisms can be reused in multiple cases of distributed application design. The tasks of middleware specialists coincide with the tasks of the infrastructure designer identified in Chapter 2.

CORBA, J2EE and Web services are examples of contemporary object middleware platforms that have resulted from advances in distributed systems. Early middleware platforms have contributed to the functions and layers found in these middleware platforms.

A number of common concerns of early and contemporary object middleware platforms have been identified in this chapter. Based on these observations an object middleware reference model has been constructed. This reference model defines object communication middleware, general purpose services and the component execution environment as sub parts.

Some or all of the layers and functions of our object middleware reference model are found in contemporary object middleware platforms, such as CORBA, J2EE and Web services.

In case multiple vendors implement an object middleware design, interoperability of these implementations is directed by rules that define an interoperability reference point. Some of the rules of the interoperability reference point can be relaxed, when a portability reference point is defined. This enables specialised mechanisms to be plugged-in to the communication middleware without compromising interoperability.

Chapter 5

Models for QoS aware middleware

This chapter provides the concepts to model QoS aspects of an open distributed system. This chapter refines the intuitive notion of QoS provided in Chapter 2 and introduces a set of modelling concepts to incorporate QoS aspects into the engineering and computational viewpoints. The concepts discussed in this chapter are the building blocks for the design presented in Chapter 6.

Chapter 2 introduces the modelling concepts and principles that are relevant to the design of an open distributed system. The intuitive notion of QoS introduced in chapter 2 is refined in this chapter. The modelling concept space, as discussed in Chapter 2, is expanded with meta-modelling concepts that are used to develop computational QoS designs. A computational QoS design is concerned with the design of QoS aspects of computational objects, such as $Q_{offered,}$ $Q_{required}$ and Q_{agreed}.

An object middleware that provides QoS support to components of a distributed application incorporates QoS functions. These QoS functions must build on the QoS functions that the underlying resources, such as resources for computing and communication, provide. Chapter 3 concludes that new protocols and mechanisms for the control of QoS in packet-based networks are expected to emerge. Therefore, middleware QoS functions must be able to adapt to these evolutionary changes of QoS functions. The modelling concepts in this chapter enable this kind of adaptation.

Chapter 4 constructs an object middleware reference model. In this chapter we relate the QoS design concepts to the reference model presented in Chapter 4. This includes the definition of a correspondence relation between computational QoS concepts and engineering QoS concepts.

This chapter is structured as follows. Section 5.1 discusses the design concerns of a QoS aware open distributed system, from the computational and engineering viewpoints. This discussion results in the identification of

QoS relations between computational design concepts and QoS relations between engineering design concepts. Section 5.2 provides a more in-depth discussion of these QoS relations. Section 5.3 reviews the design principles used to design QoS aware networks and applies these principles to the design of QoS aware middleware. Section 5.4 defines requirements on the QoS design concepts. These QoS design concepts are then defined in section 5.5 and presented as a meta-model in section 0. Section 5.7 evaluates our models and concludes this chapter.

5.1 Design concerns of QoS aware distributed systems

This section discusses the concerns that a designer of a QoS aware open distributed system faces. The design concerns are regarded from the computational viewpoint and the engineering viewpoint, respectively. A correspondence relation between the design concerns of both viewpoints is discussed.

5.1.1 Computational viewpoint concerns

The computational viewpoint abstracts from the functions and mechanisms that are needed to deal with the inherent problems that arise from the distribution of resources. In a similar way, a computational viewpoint design abstracts from the functions and mechanisms needed to deal with QoS aspects of an open distributed system. To simplify the design of a QoS aware open distributed system, an application designer is supported by an infrastructure that shields the application designer from the functions and mechanisms needed to enforce, establish or maintain QoS agreements.

An application designer ideally expresses QoS aspects of an application, independent of underlying mechanisms that establish and maintain QoS agreements.

The functions needed to establish and maintain a QoS agreement, support the computational design of a distributed application with additional distribution transparencies to the distribution transparencies identified in Chapter 2. We distinguish between functions and mechanisms that only provide for the establishment of a QoS agreement and functions that establish and maintain a QoS agreement. This leads to the definition of a *QoS enforcement transparency* and a *QoS control transparency*, respectively.

Definition 10 QoS enforcement transparency

The QoS enforcement transparency is a distribution transparency that hides the functions and mechanisms needed to establish a QoS agreement, with the purpose to simplify the design of a QoS aware distributed application.

Establishment of a QoS agreement requires functions and mechanisms in the middleware that configure the parameters of the distributed resource platform in such a way that the QoS agreement is met. In other words, the DRP is enforced into a configuration that ensures that the QoS agreement is met. However, a middleware that provides QoS enforcement transparency does not deal with changes in the DRP that result in the violation of the QoS agreement. Maintaining a QoS agreement is an additional concern covered by the QoS control transparency.

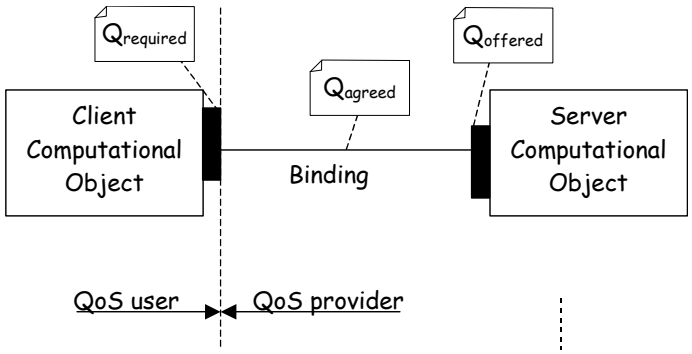
Definition 11 QoS control transparency

The QoS control transparency is a distribution transparency that in addition to hiding the functions and mechanisms needed to realise a QoS enforcement transparency, also hides the functions and mechanisms to maintain a QoS agreement

A server computational object together with the binding is a provider of QoS. A client computational object is a user of QoS (provided by the server object and the binding).

Figure 5-1 shows a client object that is bound to a server object. The QoS requirements of the client are labelled as $Q_{required}$. The QoS offered by the server are labelled as $Q_{offered}$. The QoS agreement of the established binding is labelled Q_{agreed}. The client object is in this case the QoS user, whereas the binding and server object collectively act as a QoS provider.

Figure 5-1 QoS user-provider relation in the computational viewpoint



The discussion above shows that to introduce QoS awareness into a computational design, an application designer needs meta-model concepts to design computational QoS aspects.

5.1.2 Engineering viewpoint concerns

The engineering viewpoint reveals the functions and mechanisms needed to enforce and maintain QoS agreements. These functions and mechanisms are called QoS enforcement functions and QoS enforcement mechanisms,

respectively. It is the responsibility of the infrastructure designer to design the QoS enforcement functions and mechanisms.

The QoS provided by the middleware depends on the QoS provided by the underlying resource platform, i.e., the DRP is a QoS provider to the object middleware. As the result of the late binding principle, the QoS support that the object middleware can provide to application components can only be determined at run-time. The late binding principle dictates that application components are bound to, i.e., deployed on, infrastructure components at deployment time and that the design of application components cannot incorporate deployment assumptions.

The infrastructure designer of a QoS aware middleware faces two additional challenges: first to provide DRP independent interface to BEOs for specification of QoS requirements, and second to integrate and use existing QoS functions available from the DRP into the middleware.

In the engineering viewpoint, the transport network is the QoS provider for the object middleware. The object middleware is a QoS user of the transport network and it is the QoS provider for the client or server BEO objects.

Figure 5-2 shows client and server BEO objects that are supported by object middleware. Multiple QoS user provider relations are shown, i.e., between the BEO objects and the object middleware and between the object middleware and the transport network. Inside the object middleware the three layers of the object communication middleware that are identified in Chapter 4 are revealed, therefore the object middleware in this figure corresponds to the object *communication* middleware. Substitution of the object communication middleware with a component execution environment, would still lead to the same QoS user provider relations. However, in that case the component execution environment has the role of QoS provider.

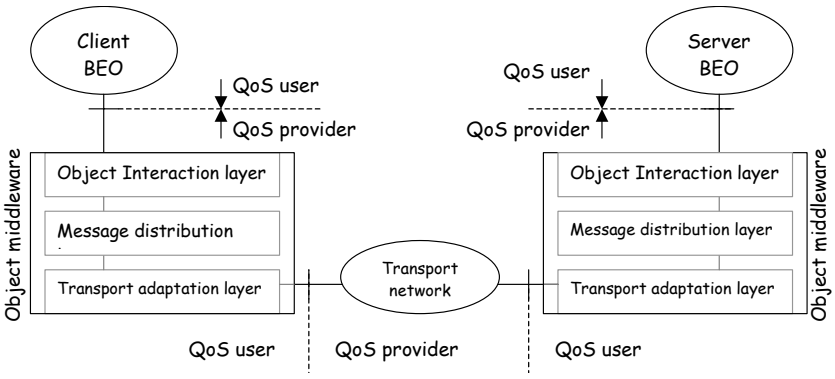


Figure 5-2 QoS user-provider relations in the engineering viewpoint

The discussion above shows that to introduce QoS awareness into an engineering design, an infrastructure designer needs to map QoS provided by the transport network to QoS provided by the object middleware. In addition, an infrastructure designer needs concepts to express the QoS aspects supported by the middleware in order to convey the middleware QoS capabilities to the application designer.

5.1.3 Correspondence

To relate the QoS aspects of the computational and engineering designs, we define a correspondence relation between the QoS aspects in these viewpoints.

The required QoS ($Q_{required}$) of a client computational object corresponds to the $Q_{required}$ of a client BEO that acts as a QoS user of the object middleware, in case the computational object corresponds to the BEO. The offered QoS ($Q_{offered}$) of a computational server object corresponds to the $Q_{offered}$ a server BEO, in case the computational object corresponds to the BEO.

Figure 5-3 shows the correspondence between computational and engineering QoS aspects. Only the correspondence between QoS aspects is shown, the correspondence between computational and engineering objects has been omitted to simplify the figure.

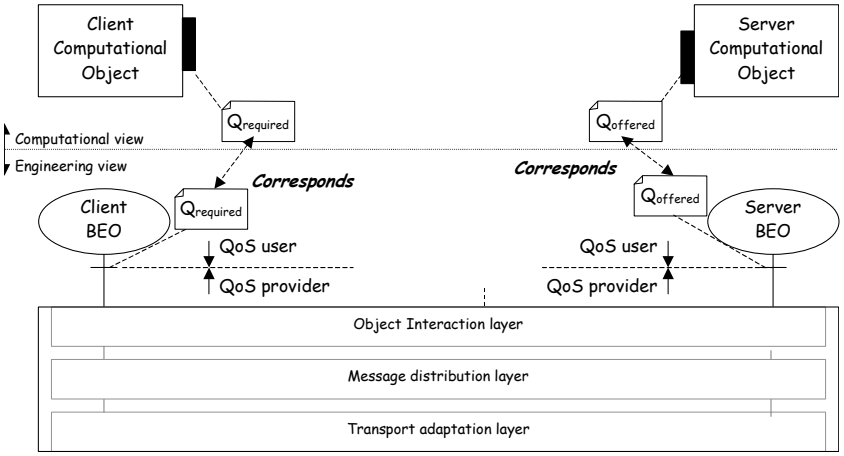


Figure 5-3 Correspondence between computational and engineering QoS aspects

From the discussion above, it follows that modelling QoS aspects for the computational and engineering viewpoint, requires appropriate modelling concepts and that these modelling concepts are closely related through a correspondence relation.

5.2 QoS relations

From the design concerns for a QoS aware open distributed system, several relations between QoS aspects of such a system have been identified. This section discusses the user-provider QoS relations, QoS mapping and QoS negotiation schemes.

5.2.1 User-provider QoS relations

The QoS agreements (Q_{agreed}) that are established between a computational client and server object are governed by the user-provider principle. This principle dictates that two entities are involved in a user-provider relationship when one entity (the user) makes use of the services providers by other entity (the provider), and the latter does not depend on the former [AA97]. A provider may offer its services to multiple user entities, in which case a separate user-provider relation occurs between each user and the provider. A user entity may be in a in a user-user relation with one or more other user entities. Such a user-user relation requires a provider to act as an intermediary.

An example of a user-user relationship is found in the engineering viewpoint between a client and server BEO. In this case each BEO has a separate user-provider relation with the DPE. The DPE acts as an intermediary to establish a user-user QoS relation between the client and server BEO. This user-user relation corresponds to a user-provider relation between a client and server object in the computational viewpoint. Figure 5-4 shows the various user-provider relations and the user-user relation in two related engineering and computational designs.

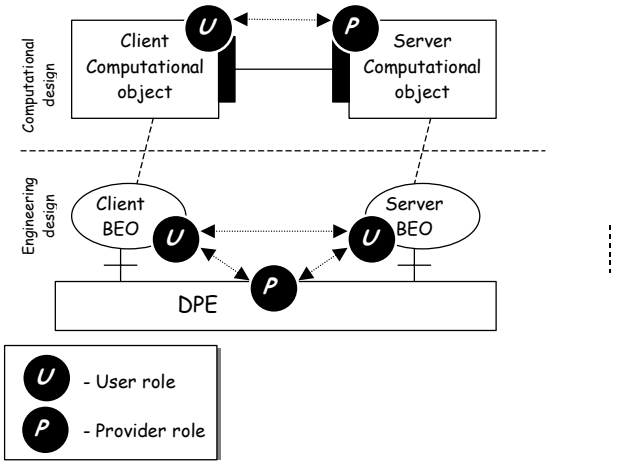


Figure 5-4 User-provider relations in computational and engineering viewpoint

The user-provider relation can be applied recursively to a provider. That is, a provider may be decomposed into one or more lower level user entities and a lower level provider entity. The lower level user entities implement the services of the higher-level provider using the services of the lower level provider. To this end, the lower level user entities interact with their higher-level neighbours (user-provider relation), with peer user entities (user-user relation) and with the lower level provider (user-provider relation). The recursive process of applying the user-provider relationship is a stepwise refinement of the provider.

In the context of a QoS aware open distributed system, two or more objects require to interact with each other at a certain quality level. These objects rely on the services provided by a QoS aware distributed processing environment. The QoS aware distributed processing environment acts as a QoS provider that establishes user-user QoS relations between client and server objects.

Recursive application of the user-provider principle results in a QoS provider that consists of a set of lower level entities that use a lower level QoS provider. In the case of a distributed processing environment, the lower level QoS provider is the distributed resource platform.

5.2.2 QoS mapping

The notions of QoS at a higher and at the lower level user-provider boundaries are different because a higher-level service considers QoS in other terms than the lower level service. For example, a higher-level service may consider QoS in terms of number of object interactions per second, whereas the lower level service may express the QoS in terms of bandwidth. This demonstrates the need for a function that maps higher-level QoS terms to lower level QoS terms. Figure 5-5 shows the relations between a higher and lower level QoS user and provider applied to open distributed systems.

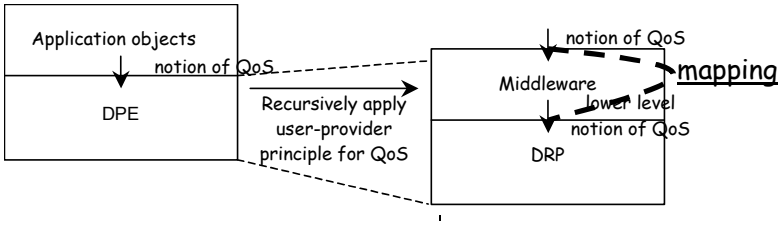


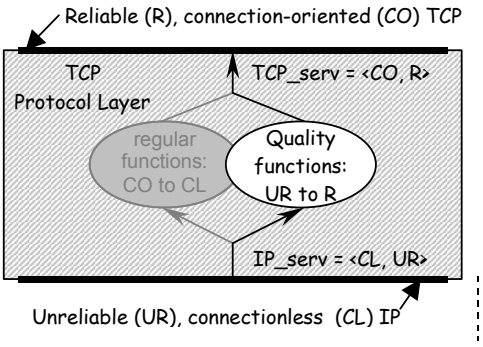
Figure 5-5 QoS user and QoS provider relations

In addition to functions that map a higher-level notion of QoS onto a lower level notion of QoS, there is also a need for functions that maintain an

established QoS level. For example, delivering a reliable higher-level service over an unreliable lower level service requires functions that are able to detect and correct errors introduced by the lower level service. Such functions may for instance use a Forward Error Correction (FEC) mechanism in which senders add FEC information to data units before they transmit them. Receivers then use the FEC information to detect and correct any errors that were introduced during transmission of the data unit. Another possibility would be to use error correction and detection functions that use sequence numbering and retransmission. The TCP protocol, for instance, uses the latter approach to provide a reliable service on top of the unreliable IP service.

The above examples show that there are functions that primarily deal with bridging the quality gap that exists between a higher-level service and a lower level service. These functions complement the functions that mainly deal with bridging the functional gap between the higher and lower level services. For example, TCP provides functions that allow TCP service users to establish and release connections over the connectionless IP service (functional gap). TCP complements these functions with functions that provide a reliable service on top of the unreliable IP service (quality gap). The division of functions suggests that a service provider offers functional support and an associated quality support, where each support is realised by a dedicated set of functions. Figure 5-6 shows how the functional and quality gap categorises the functions of a service provider.

Figure 5-6 Functionality-quality relations for TCP



In case of an open distributed system, the functional gap and the quality gap between application objects and the distributed resource platform must be bridged by a QoS aware object middleware. An infrastructure designer is responsible to design the mapping and quality functions that are part of a QoS aware object middleware.

5.2.3 QoS negotiation schemes

A QoS agreement (Q_{agreed}) is established at run-time as the result of interactions between the application objects and the distributed processing environment. This requires a negotiation of a QoS that is acceptable to the application objects (= user) and the distributed processing environment (= provider).

QoS negotiation is initiated by application objects or by the distributed processing environment (DPE).

An application object that requires the establishment of a QoS agreement typically captures these requirements in the form of a specification. The specification may for instance indicate the bandwidth that the object requires, the maximum latency that it wants method invocations to be subject to, and so forth. As part of the negotiation process, the application object may convey the specification to one or more other application objects, to the distributed processing environment or to a combination thereof.

When an application object conveys a QoS specification to a DPE as part of a request to that DPE to establish the specified QoS, the object's specification needs to be in line with the capabilities of the DPE. That is, the object must specify a QoS that the DPE is able to deliver. For this purpose, the DPE typically publishes the classes of QoS that it can handle. For instance, a provider may publish the fact that it supports a 'Performance' class, which gives objects the opportunity to request delay constraints on remote method invocations. If an application object is unaware of the DPE capabilities, it may first query the DPE to figure out which classes of QoS are supported. The object can then select the class that best meets the QoS it requires.

Alternatively, the DPE may initiate the establishment of a certain QoS. This may for instance occur when the DPE includes a mobile network and a roaming application object just got into range of that network. The DPE may then take the initiative and advertise the different classes of QoS that it supports as well as the cost associated with them. If the QoS classes that become available are a better match to the application object requirements, the object may decide to renegotiate a QoS agreement between the DPE and one or more other application objects.

Alternative negotiation schemes

The scenarios above assume that the negotiation of a suitable QoS is modelled as a single interaction between two or more application objects and a DPE (multiparty negotiation). The result of the interaction is an agreed QoS. Figure 5-7 shows an example of such a negotiation interaction.

In this example, the interaction involves application objects O_1, O_2 and O_3, and a DPE.

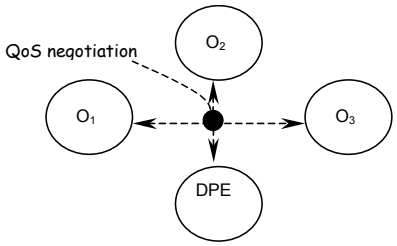


Figure 5-7 The negotiation of a suitable QoS as a single interaction

At a lower level of abstraction, the negotiation interaction can be refined in many ways in terms of combinations of user-user and user-provider interactions. Figure 5-8 shows two ways of negotiating a QoS. O_1 is assumed to be the initiator in both cases.

Figure 5-8a shows a form of negotiation in which the application objects first negotiate a suitable QoS and then involve the DPE in the process. The figure assumes that object O_1 is responsible for interacting with the DPE on behalf of O_2 and O_3. The user-to-user interactions require a provider to act as an intermediary (DPE' in Figure 5-8a), but that this provider does not need to be the provider that will eventually establish the QoS (DPE in Figure 5-8a). Such an 'out-of-band' provider simply conveys QoS related information between objects without interpreting it. Note that DPE and DPE' may be the same.

When the objects and the DPE have agreed upon a suitable QoS, the DPE needs to reserve and initialise resources to actually establish the QoS. The objects may subsequently commence interacting with each other, while the DPE ensures that the agreed QoS is sustained. The QoS may be established for each application object individually. Also note that the configuration actions the DPE makes to its resources (e.g. to reserve and initialise them) are implicit to the application objects (transparency principle).

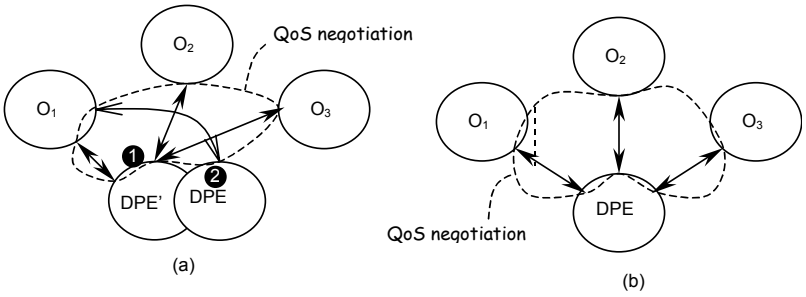


Figure 5-8 Two ways of negotiating a QoS (refinements of previous figure).

Figure 5-8b shows a form of negotiation in which the provider is involved right from the start. When the DPE has decided that it can support a QoS that is in line with O_1's request and with O_2's and O_3's responses, it reserves and initialises the necessary resources and initialises the quality functions that are required to maintain the agreed QoS. The DPE subsequently informs the application objects of the agreed QoS. If the DPE discovers it cannot provide a QoS that meets O_1, O_2 and O_3's requirements, it may start another round of negotiation or let the negotiation terminate unsuccessfully.

5.3 Scope of QoS functions

This section reviews the design principles used to design QoS aware networks and applies these principles to the design of QoS aware middleware.

Functions that realise QoS support in a distributed processing environment are called QoS provisioning functions. The design of QoS provisioning functions and how these functions are positioned in an open distributed system is guided by the separation and the integration principles [ACH98].

5.3.1 Design principles

The *separation principle* dictates that transfer, control, and management of data are three functionally distinct activities [La92]. The *integration principle* states that QoS must be configurable, predictable and maintainable over all architectural layers to meet end-to-end quality of service [CCG+93].) Both principles originate from broadband and multimedia networks, but are applicable as guiding principles to the design of QoS support in open distributed processing environments.

5.3.2 Separation principle applied

Application of the separation principle for QoS provisioning functions means that the transfer of data, the control of QoS and the management of QoS are three functionally distinct activities, which should be kept separate. In an open distributed system, the transfer of data occurs when a client object invokes a server object. Therefore, when we apply the separation principle to the DPE, we substitute the notion of data transfer with object interaction. The QoS services of a QoS aware distributed system may thus be structured into object interaction services, QoS control services and QoS management services.

The three services have a "control" or "govern" relationship with each other. That is, the QoS control services govern the behaviour of the object interaction service, while the QoS management services govern the behaviour of the QoS control services (and, indirectly, that of the object interaction services). The QoS control services typically affect the prediction, establishment and maintenance of QoS for individual bindings, whereas QoS management performs the same task on a time-scale that transcends the lifetime of individual bindings. Figure 5-9 depicts the separation principle applied to QoS functions.

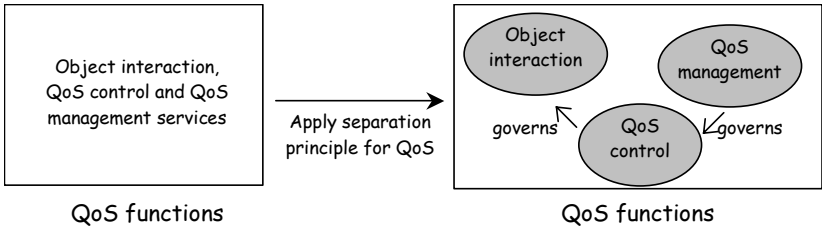


Figure 5-9 Separation principle applied QoS functions

The QoS control and QoS management functions may be distributed. The functions that provide support for QoS negotiation, for example, perform local activities as well as distributed activities. Another example concerns the mapping of middleware level QoS notions to DRP level QoS notions through a database lookup. The mapping tables offered by the database may be stored in a central location. QoS control functions distributed throughout the DPE will perform distributed activities to access the remote database.

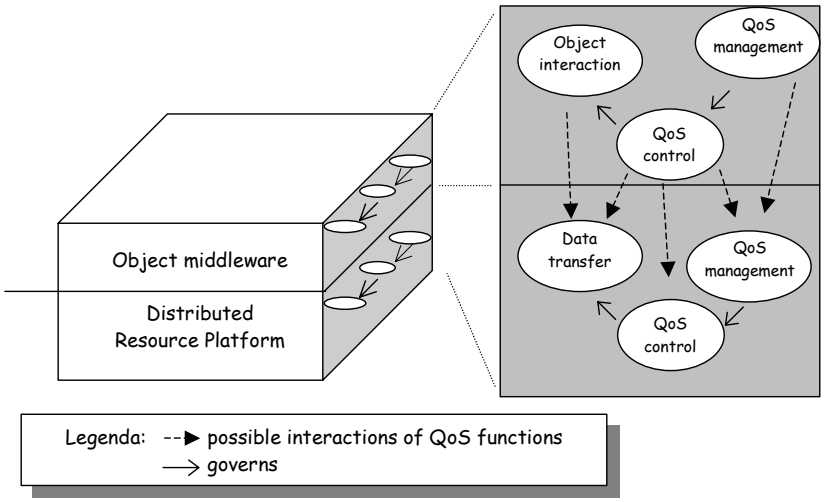
5.3.3 Integration principle applied

A QoS aware distributed processing environment predicts, establishes and maintains QoS agreements for applications objects. From the integration

principle it follows that all the middleware parts of a QoS aware DPE (i.e., object communication middleware, general purpose object services, component execution environment) must be QoS aware. That is, every architectural component should ideally be able to predict, establish and maintain QoS so that the overall system is able to provide QoS for application objects.

Integration of QoS functions at the object middleware layer and QoS functions at the distributed resource layer introduces several choices to an infrastructure designer. At the DRP layer, QoS functions are classified in a similar way as at the object middleware layer. Data transfer functions are distinguished from control functions and from management functions. The separation of QoS functions in these three categories is also determined by the time-scale at which these functions operate. However, the time-scale of each of these three sets of QoS functions is not necessarily the same as the time-scale of the QoS functions found at the object middleware layer. Therefore, an infrastructure designer must carefully choose how middleware layer QoS functions employ DRP layer QoS functions to achieve integrated QoS support. As a general principle we regards QoS functions with the scope of a single binding QoS control functions, whereas QoS functions with the scope of a set of bindings are considered to be QoS management functions. Figure 5-10 shows the positioning of QoS functions at the middleware layer and DRP layer. QoS functions at the middleware layer use QoS functions at the DRP layer in various combinations.

Figure 5-10 Integration of QoS functions at different layers



The QoS awareness of a communication network of a DRP may concern individual data flows, aggregates of data flows, or to both. In case the communication network is aware of the QoS of individual flows it can exercise a fine grained level of QoS control for each flow, but will not scale up to large numbers of data flows. Scalability is limited because each node of the DRP has to maintain QoS related information about each flow. For large numbers of flows, this implies a large memory footprint for storage and consumes too much processing power for information look-up and manipulation. The overall performance of the DRP is affected when too many resources are allocated to QoS functions.

On the other hand, in case a the communication network of a DRP is only aware of the QoS of data flow aggregates, it can exercise a coarse level of QoS control only, but generally scales better to large numbers of flows. After all, the DRP only needs to maintain QoS related information about aggregates of flows, which takes substantially less storage and processing resources for large numbers of flows.

As identified in Chapter 3, experts expect that large-scale QoS aware communication networks will use a combination of per-flow and flow aggregate QoS awareness. In particular, per-flow QoS awareness will be used close to the network boundaries (e.g. in access routers and in computing nodes attached to the network) where the number of flows is relatively small. Per flow aggregate QoS awareness will then be used in the core of the transport network, in particular on backbone networks.

A consequence of the integration principle is that a DPE can only offer the same granularity of QoS control as is supported by the DRP. The DPE depends on which QoS functions are available from the DRP and the granularity of QoS control that these functions provide.

In practice, an open distributed system is constructed from parts manufactured by different vendors and these parts are purchased and owned by different organisational domains. As a result, the available QoS support provided by a DRP depends on how it is deployed and which hardware and software components are used to construct a DRP. An infrastructure designer, responsible for the design of QoS functions at the object middleware layer, must therefore find a means to deal with the unavailability of QoS functions.

5.3.4 Determining the scope

The QoS control and management functions are actively involved in the establishment and maintenance of a QoS agreement. The QoS support found in object interaction functions is primarily concerned with the transmission, receipt, processing and forwarding of request/reply messages.

These functions are not concerned with the establishment and maintenance of a QoS agreement.

Examples of QoS control functions are functions for (re)negotiation of an agreed QoS, functions for adaptation to variations in QoS, functions for mapping between different notions of QoS (cf. Figure 5-5), tests for resource availability functions, resource reservation and configuration functions, and resource monitoring and reconfiguration functions. The object interaction functions may also contain elements that influence QoS. However, unlike the activities of the QoS control functions, the influencing activities of the object interaction functions are limited by the policies of an already established binding. The activities of the QoS control functions, on the other hand, control QoS by manipulating the policies of a binding or can involve establishment of a new binding object.

The span of control, or scope, of a QoS function is determined by the time-scale at which the function operates. Some QoS functions are active for each method invocation, whereas others are only active during the binding establishment. QoS functions that operate on an even larger time-scale get active after a sequence of bindings has been established.

Object interaction functions are tailored to an efficient transfer of request and reply messages and efficient processing of these messages on the client and server side. The QoS actions of the object interaction functions must take place on the same time-scale as a method invocation. The QoS control functions operate at the time-scale of binding establishment. QoS management functions operate at the time-scale of multiple binding life cycles.

An infrastructure designer needs to determine the scope of a QoS function as this determines how efficient the QoS function must be realised. Obviously, the smaller the time-scale at which a QoS function operates the more efficient, i.e. with as little overhead as possible, the function must be engineered.

5.4 Requirements on QoS design concepts

The design principles, presented in the previous sections, guide the design of QoS functions. These principles must be augmented with a refined set of concepts that facilitate the exchange of designs between application designers and infrastructure designers. QoS providers use these concepts to advertise and express their QoS capabilities to potential users. We need design concepts to be able to specify the required, offered and agreed QoS and formulate the following requirements on these design concepts.

5.4.1 Extensible

As the result of the integration principle, the QoS support that a DPE can offer depends on the QoS support offered by the communication network and computing systems. The network and computing systems may be owned by different organisational domains and may vary in the level and granularity of QoS support. In addition, new QoS functions can become available as organisations upgrade their network, their computing systems or both.

To cope with these differences in a particular deployment of an open distributed system, the design concepts that model QoS aspects must be extensible. This allows for the design of new QoS capabilities as they become available to the distributed resource platform. An extensible set of QoS design concepts caters for the design of future QoS capabilities.

5.4.2 Composable

When new QoS capabilities become available from the DRP to the infrastructure designer, existing models of QoS capabilities of the object middleware do not have to become obsolete, but should rather be incorporated into a new model of the middleware QoS capabilities. Consequently, QoS designs must be composable, i.e. new QoS designs can be constructed from existing designs.

5.4.3 Verifiable

The concepts that an application designer uses to create required and offered QoS designs must be checked against the QoS concepts created by the infrastructure designer. An incomplete or unsupported QoS design is invalid and should be detected by the DPE. To let a DPE detect and possibly reject an invalid application layer QoS design, a QoS design must be verifiable.

A valid QoS design means that it is positively verified against some QoS type and that this QoS type is supported by the DPE. This ensures that the DPE can support the requested application layer QoS design to at least some degree. A QoS type is a predicate on some QoS design.

5.4.4 Suitable run-time representation

QoS design concepts, like any other design concepts, are conceptual models manipulated in the mind of a designer. For specification purposes the QoS design concepts must be represented in a way that suits infrastructure designers and application designers. In addition, a QoS design also needs a suitable run-time representation. The run-time representation is managed by the DPE and is used to exchange QoS specifications between application

objects and the DPE. The suitability of a run-time representation of a QoS design is determined by several factors.

A run-time representation must be efficient in terms of memory usage and processing complexity. This is especially important when a QoS specification is accessed by QoS functions that operate at the time-scale of an object interaction. Access to a QoS specification must not introduce a lot of overhead.

A run-time representation should be compliant with existing software engineering practices and standards where possible. Use of existing standards enables an infrastructure designer to re-use existing design patterns for the implementation of the run-time representation. In addition, application designers may use existing tools to create QoS specifications.

5.5 QoS design concepts

The set of QoS design concepts presented in this section meets the requirements presented in the previous section. The approach is to define a meta-model, which is derived from the OMG MOF model, from which QoS designs can be developed.

The definitions found in the ISO/ITU QoS standard [ISO X.641] are adopted as a starting point to derive the meta-model. These concepts have been introduced in Chapter 3. Some of these definitions are interpreted and lead to our own set of basic QoS concepts.

5.5.1 ISO QoS concepts

The definitions of user requirement, QoS category, QoS characteristic and QoS requirement are adopted.

Definition 12 User requirement

A user requirement is a quantifiable quality aspect of the interactions between user entities that are needed by (one of) these entities, to enable them to achieve their interaction objective, and including the required quantity of the aspect.

User requirements are defined in the scope of a user entity. By using these requirements, a user entity can identify and express the quality needs for their interactions, in a way independent from the means that realize these interactions. This means that the user requirements are of concern to the user-user relation. It also means that these requirements are independent of the QoS capabilities of a provider.

However, to enable provisioning of QoS, these user requirements have to be formulated in terms of provider-oriented concepts, i.e. QoS characteristics and QoS requirements, which will be defined a little later.

User requirements are aggregated in the following definition:

Definition 13 QoS category

A QoS category is a group of user requirements that leads to the selection of a set of QoS requirements..

The QoS category is the user perspective on a set of requirements for QoS support, which are independent of the QoS capabilities of the provider. When these requirements are expressed in terms of the QoS capabilities supported by a provider, they are referred to as QoS requirements. A QoS requirement is expressed in terms of a QoS characteristic.

Definition 14 QoS characteristic

A QoS characteristic is a quantifiable aspect of QoS, which is defined independently of the means by which it is represented or controlled.

A QoS characteristic specifies a particular quality aspect of the capability of a provider, independently of the means by which it is represented or controlled. A QoS characteristic is also quantifiable. The latter enables the matching of an offer to a demand using computational means. It also enables the mappings of QoS specifications. QoS characteristics at the lower interface (i.e. the DRP boundary) are for instance delay, bit-rate and probabilistic transmission error rate. Characteristics at the upper interface (i.e. the middleware to application boundary) are for instance rate, accuracy, freshness and urgency.

Definition 15 QoS requirement

A QoS requirement is QoS information that expresses part or all of a requirement to manage one or more QoS characteristics, e.g. a maximum value, a target, or a threshold

A QoS requirement is always associated with one or more QoS characteristics. It represents the needed value or range of values of the associated characteristic, and also the *qualifiers* of these values, e.g. the upper-/lower-bounds or the probabilistic properties of the corresponding characteristic.

QoS Classes

From the provider perspective, it is useful to group the supported QoS characteristics. This grouping allows association of QoS requirements with a set of characteristics that a provider regards as a logical unit. As was mentioned earlier, a provider may advertise its QoS capabilities. In analogy

to the way many network or transport service providers make their QoS capability known and available, we define the concept QoS class.

In ATM networks, a service class (also called service category) represents the QoS capability of this network. Examples are the classes CBR (Constant Bit Rate), VBR/NRT (Variable Bit Rate - Non Real Time), and ABR (Available Bit Rate). These classes are defined in terms of QoS characteristics like CLR (Cell Loss Ratio), CTD (Cell Transfer Delay), and MTD (Minimum Cell Rate).

The previously given examples motivate the definition of the concept QoS class.

Definition 16 QoS class

A QoS class is the QoS capability of a service or a set of services, defined in terms of QoS characteristics and the corresponding ranges or values that can be supported by the provider.

A QoS class defines the dimensions of QoS aspects and includes the domain of values that are supported by a service provider. Figure 5-11 shows the geometrical representation of a class as a region in a space spanned by the QoS characteristics

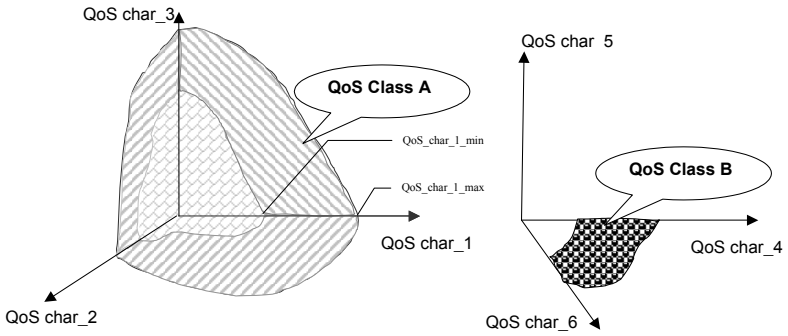


Figure 5-11 Examples of QoS classes, associated characteristics and supported ranges or values

The QoS characteristics do not need to form an orthogonal basis, i.e. QoS characteristics may have some interdependency. The ranges or values of the characteristics that are supported by the provider determine the region in this space. The dimension and shape of the region of a QoS class expose essential properties of the providers QoS capability.

As opposed to QoS categories, QoS classes are provider-oriented, in the sense that they represent the capabilities of a provider and are meant to accommodate the user oriented QoS categories.

5.5.2 Additional basic concepts

The ISO/QoS definitions offer a generic set of concepts for the design of QoS aspects. Additional concepts are needed that relate the QoS design concepts to object oriented concepts. The concepts presented below are adopted from the QoS Modeling Language (QML) [FrKo98]. Notions defined in QML are:

- QoS dimension;
- Direction of a QoS dimension;
- QoS category;
- QoS contract;
- QoS contract type.

For each of these notions we discuss the relation with the definition presented in the previous section

The term QoS *dimension* is closely related to a QoS characteristic, as it also specifies a quantifiable aspect of QoS. The main difference is that a QoS dimension has a number of concrete attributes such as a name, a domain of values (e.g. non-negative numbers) and the units of the value (e.g. milliseconds, kb/sec, or per hour). Examples of QoS dimensions are throughput, rate, delay, failure rate and integrity level. For each of these dimensions, the domain of values, the unit of a value and the *direction* must be specified.

The direction of a QoS dimension can be increasing or decreasing. An increasing QoS dimension means that higher values are better. An example of an increasing QoS dimension is rate, expressed as number of invocations per second. For a decreasing QoS dimension a lower value is considered better. Delay is an example of a decreasing QoS dimension.

QoS dimensions that are grouped define a *QoS category*. Examples of QoS categories are performance, security or availability. Each QoS category consists of one or more QoS dimensions. A QoS category resembles an ISO QoS class, as it is defined in terms of QoS dimensions and the corresponding ranges or values that can be supported by the provider.

A provider expresses its QoS capabilities in terms of one or more QoS categories that it supports. A QoS category concerns not the individual agreements that have been made, e.g., QoS agreements between a set of application objects and the DPE. A QoS category defines the potential space for the establishment of QoS agreements. Actual QoS agreements that are established depend upon the availability of resources from the provider. Such an agreement is also referred to as a *QoS contract* between a user and a provider.

A QoS category can be regarded as a predicate, which must hold for all QoS contracts that a provider makes. From this perspective the QoS

category defines a *QoS contract type*. Figure 5-12 shows an example of a Performance contract type in QML. This contract type supports delay and throughput as dimensions.

Figure 5-12 A QML performance contract type

```
type Performance = contract {
    delay: decreasing numeric msec;
    throughput: increasing numeric mb/sec;
    };
```

A contract type specification is a way for a provider to express its capabilities. A contract type identifies and defines a QoS class. A QoS contract is a constraint for a given QoS class, i.e., it imposes constraints on the dimensions defined in the contract type. If a contract type is considered a template for the construction of valid contracts, a contract is then an instantiation and parameterisation of a contract type. For an example of a contract see Figure 5-13.

Figure 5-13 A QML performance contract

```
somePerformance = Performance contract {
    delay < 180;
    throughput > 2;
    };
```

The contract above specifies that the delay dimension should be less than 180 ms and the throughput should be larger than 2 mb/sec. The contract does not specify if this is a required, offered or an agreed QoS.

5.6 Meta-model concepts

This section discusses the meta-model concepts that an infrastructure designer and application designer use to develop QoS aware middleware and QoS aware applications, respectively. The notions of QoS contract types and QoS contracts are discussed. The QoS contract types are designed by an infrastructure designer and the QoS contract by an application designer.

5.6.1 QoS contract types

The QoS design concepts presented before are captured in a model that consists of a set of classes and associations between these classes. This model is a meta-model in the sense that QoS designs can be instantiated from it. Our meta-model is an instantiation of a standardised meta-meta-model, which is known as the OMG MOF model.

The meta-model for QoS designs consists of two parts: a part that defines QoS contract types and a part that defines QoS contracts. The latter part also relates the contract to its contract type. This section discusses the first part that focuses on the QoS contract types.

A contract type contains zero or more dimensions. The contract type and the dimension are defined as a class. A contract type has a *name*, a *major version* and a *minor version* number as attributes. The name of the contract type makes it easier to refer to the contract type and the version numbers enable a provider to offers multiple versions of the contract type.

The attributes of a dimension are a *direction*, *dimension type* and a *description of the unit*. The direction of a dimension is increasing or decreasing. The dimension type defines the domain of values that apply to a dimension. The dimension type is defined as a TypeCode, which allows the reuse of the types defined in the CORBA::TypeCode system. A dimension may allow certain constraints or disallow them. A boolean attribute *allowMultiConstraint* indicates which constraints are supported for that dimension. The purpose of a multi-constraint is discussed in the part that defines QoS contracts.

The relationship between a contract type and it dimensions is modelled using the container-contained pattern. The container-contained pattern is also found in the CORBA Interface Repository specification and is a variation on the Composite design pattern [Ga+95]. Essentially, the container-contained pattern provides a design solution in case multiple objects must be grouped together by a single object and all objects, whether single or grouped, must be treated in the same way. The solution consists of an abstract container object that contains a set of abstract contained objects. The container itself can also be contained in another container. This allows for a potentially infinite level of containment.

The container-contained pattern is used to model the relationship between a contract type and a dimension. As a result, a dimension is a contained object and a contract type is a container object. The benefit of the pattern is that multiple contract types can be contained in a larger contract type. This enables the composition of contract types from contract types that already exists.

Models instantiated from the meta-model are constrained so the validity of these models can be verified. The meta-model requires that a dimension can only be contained in a contract type. This constraint is required as the container and contained classes are also used in the part of the model concerning the contracts.

The meta-model classes and their relations are represented as a UML class diagram. Figure 5-14 shows a UML specification of the meta-model for QoS contract types.

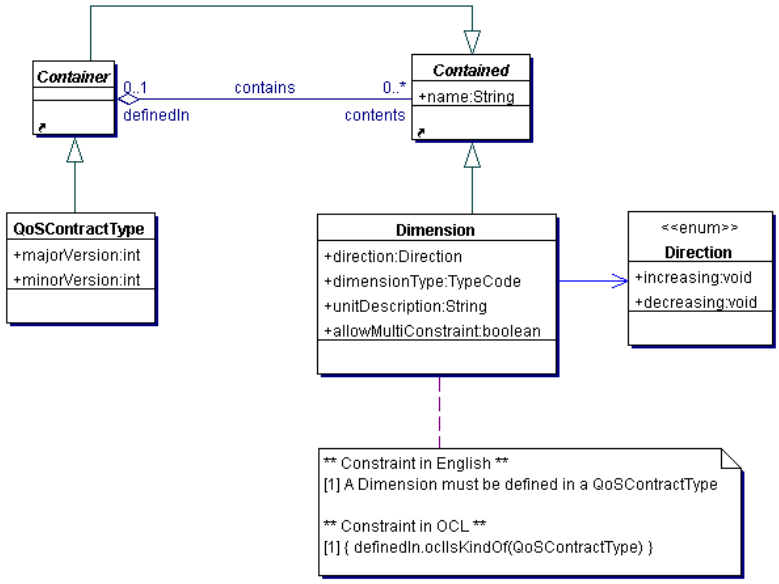


Figure 5-14 Meta-model for QoS contract types

The QoSContractType meta-class and associated meta-classes are used to validate a QoS contract.

5.6.2 QoS contracts

This section discusses the part of the meta-model that concerns the QoS contracts. A QoS contract consists of a set of constraints on QoS dimensions. A contract must be associated with a contract type. A contract is only valid when it defines constraints for dimensions that have been defined for its associated contract type.

A constraint on a dimension is defined in terms of an *operator* and a *parameter*. The constraining operators are taken from the set {=, <, >, =<, >=}. The parameter of a constraint must match the dimension type attribute as found in its corresponding dimension. An example of a constraint on a dimension is "delay < 120", assuming there is a dimension with name delay and with a numeric dimension type code.

In some situations the set of five constraining operators is not sufficient. The constraint governs all interactions that are subject to the QoS contract and are therefore considered hard constraints. In the example above the constraint requires that every interaction has a delay less then 120. It may be that the user does not want to set such hard requirements, or that a provider does not support hard guarantees. In this case a statistical

constraint is used. An example of a statistical constraint is "delay < 120 for at least 80% of the cases".

A statistical constraint is a special case of a single constraint. It is common to require multiple statistical constraints for a single dimension. For example, a constraint on the dimension delay could be "delay <120 for at least 80% of the cases and average delay < 200". With this constraint an occasional delay may be longer than 120, but on all invocations the average delay should be less than 200. Such a constraint is captured by a multi-constraint. A multi-constraint has a *statistical operator*, which is taken from the set {percentile, frequency, mean, variance} and a *statistical parameter*. Statistical constraints can only be contained in a multi-constraint.

The container-contained pattern is again used to model the relationship between a contract and a single constraint and in a similar way the relation between a multi-constraint and a statistical constraint. The meta-model constraints are formulated in such a way that a contract may contain single constraints and multi-constraints, while a multi-constraint may contain statistical constraints.

The meta-model classes and their relations are represented as a UML class diagram. Figure 5-15 shows a UML specification of the meta-model for QoS contracts.

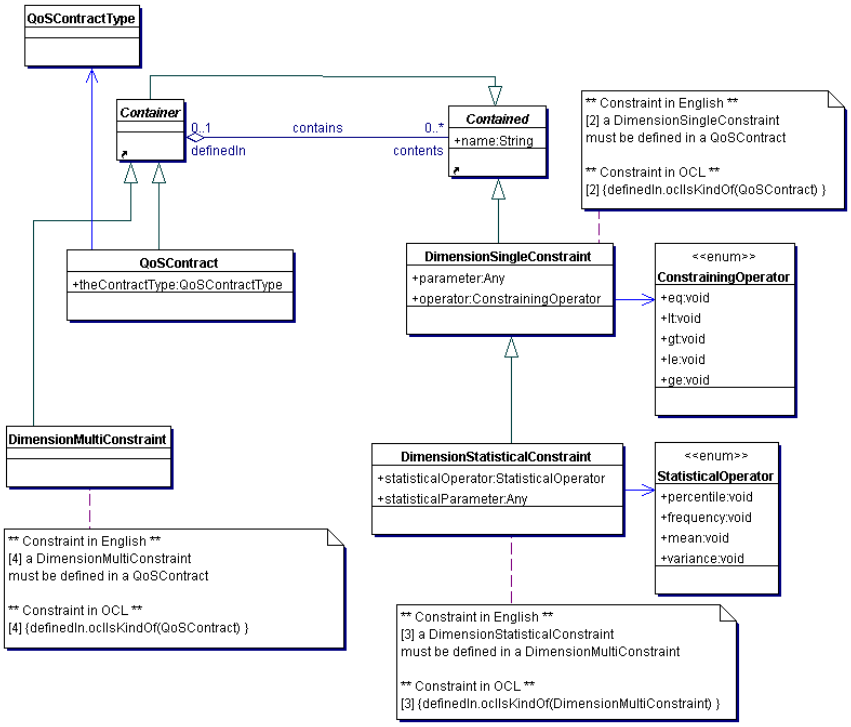


Figure 5-15 Meta-model for QoS contracts

5.7 Evaluation and conclusion

The meta-model discussed in the previous sections offers a solution to the requirements on the QoS design concepts. The requirements presented in section 5.4 are evaluated.

5.7.1 Extensible

The extensibility requirement states that when new QoS capabilities become available from a provider, the design concepts must be able to capture this.

The extensibility requirement is met through the meta-model approach. The meta-model enables the construction of contract types that represent the QoS capabilities of a provider. A QoS contract type represents a potential space for the establishment of QoS agreements. The meta-model offers the designer the freedom to choose this space according to the

capabilities of a provider. Construction of new contract types enables the extension of this space.

5.7.2 Composable

The composability requirement states that existing QoS designs should be reused in new designs.

The composability requirement is met through the container-contained pattern. Through this pattern a contract type can be a container for QoS dimensions and also a container for other contract types. This offers a designer the freedom to compose new contract types using existing contract types.

5.7.3 Verifiable

The verifiability requirement states that a QoS specification must be checked for validity. A QoS specification is valid if it is positively verified against some QoS type and that the provider supports this QoS type.

The verifiability requirement is partially met by the meta-model. The validity of a QoS contract can be verified by checking it against its contract type. This means that for all the dimension constraints found in the QoS contract a dimension must be defined in the contract type. In addition, the type code of a dimension must match the parameter type of a constraint.

Verification of the support of a QoS type by a provider is not facilitated by the meta-model, but is determined by a provider at run-time.

5.7.4 Suitable run-time representation

The suitable run-time representation requirements states that a run-time representation must be efficient in terms of memory usage and processing complexity and that the run-time representation should be compliant with existing software engineering practices and standards.

Again, the meta-model offers a solution to this requirement. The meta-model does not prescribe any run-time representation. The OMG has standardised two mappings from a meta-model to IDL and from a meta-model to XML. These standardised mappings enable the representation of a QoS design in a manner that is compliant with existing design tools, thus enabling a designer to manipulate a QoS design in a tool of choice.

Efficiency of the run-time representation is an implementation issue for the infrastructure designer and can be optimised where needed. For example, when a QoS contract is represented as an XML tree, an efficient DOM tree implementation must be chosen to minimise the overhead of DimensionConstraint parameter lookups.

Chapter 6

Design of a QoS provisioning service

This chapter presents the design of a general purpose service that provides QoS support. This service is called the QoS Provisioning Service (QPS) [FaHa01, Ha00, HFG01, FHLS02]. QPS enables application objects to specify a QoS contract and associate client and server interfaces with these contracts. A binding between a client and server object that have a QoS contract is subject to the establishment of a QoS agreement. QPS acts as a broker between the application level QoS requirements and the available QoS mechanisms of the distributed resource platform.

QPS supports the QoS aspects of a design of a distributed application, according to the QoS modelling concepts discussed in chapter 5. QPS ensures that QoS agreements are established and maintained during the life-time of the binding [BHP+00].

This chapter is organised as follows. Section 6.1 presents an overview of the features of QPS and the services it provides. Section 6.2 describes QPS from an engineering viewpoint. Section 6.3 shows how the generic design of QPS has been transformed to a specific implementation for a CORBA context. Section 6.4 discusses some of the design decisions that were made for the CORBA implementation of QPS. Section 6.5 introduces QIOP, which is our protocol that ensures performance contracts between a CORBA server object and its clients. Section 6.6 describes an experiment with QIOP implementation and demonstrates the performance benefits of QIOP over the standard CORBA protocol. Section 6.7 presents conclusions and identifies issues for further investigation.

6.1 Overview of QPS

This section presents an overview of the QoS provisioning service (QPS). It describes the service that QPS offers to computational objects and the realisation of this service, discussing the QPS lifecycle, a framework for QoS negotiation and a framework for QoS control.

6.1.1 Service description

QPS is a general purpose object service that manages the life cycle of a QoS aware binding. In the computational viewpoint, QPS is modelled as a computational object that client and server computational objects use to establish a QoS aware binding that supports the QoS requirements for their interactions.

QPS provides application level support for QoS contracts. QPS shields application objects from QoS contract negotiation and the mapping of QoS agreements to internal actions on the QoS functions and mechanisms of the object middleware or the DRP. QPS also shields application objects from the functions and mechanisms needed to maintain a QoS agreement.

QPS has been designed so that it can be extended. When new functions and mechanisms for QoS enforcement and QoS control become available, these functions and mechanisms can be added to QPS. New contract types can be made available to the application developer. Establishment and maintenance of QoS contracts derived from these new contract types are the concern of QPS and not of the application developer.

QPS adapts to the dynamic availability of processing, storage and communication resources. QPS monitors actual QoS levels achieved by the middleware and DRP and takes counter actions in an attempt to maintain a QoS agreement. However, if too many resources of the DRP are consumed by components that are outside the control of QPS, a QoS agreement can be violated and the associated binding is released. In this case, QPS will notify the application objects.

QPS is configurable through policies. The infrastructure designer is responsible for defining the mapping of QoS agreements to the QoS functions and mechanisms provided by the DRP. Negotiation of QoS agreements can also be configured through policies. The infrastructure designer provides a number of negotiation strategies. A deployment designer decides which strategy is used for a specific deployment of QPS. Finally, the adaptation policies of QPS are configurable. These policies are concerned with the strategies that QPS employs to control the QoS levels achieved by the middleware and DRP.

6.1.2 QoS aware binding lifecycle

The purpose of the QoS provisioning Service (QPS) is to control the resources of the distributed resource platform (DRP) in such a way that some agreed QoS (Q_{agreed}) is established and maintained for the lifetime of the binding. This agreed QoS is the result of a matchmaking process between the offered QoS ($Q_{offered}$) of the server object and the required QoS of the client ($Q_{required}$).

An application object expresses its offered or required QoS as a QoS contract according to the concepts defined in chapter 5. QPS takes the QoS contract of a client and a server object as input to establish an agreed QoS.

A QoS agreement is valid for a single binding between a client and a server object. A binding that is subject to a QoS agreement is a QoS aware binding. A QoS aware binding consumes resources from the distributed resource platform. Therefore, a QoS aware binding is only created upon demand and discarded when the client does not need it anymore or when influences outside the control of QPS force the release of a binding.

The scope of QPS is a single binding between a client and a server application object. Figure 6-1 shows the five life-cycle phases of QPS of a client-server binding. The lifecycle phases are inform, negotiate, establish, operate and release.

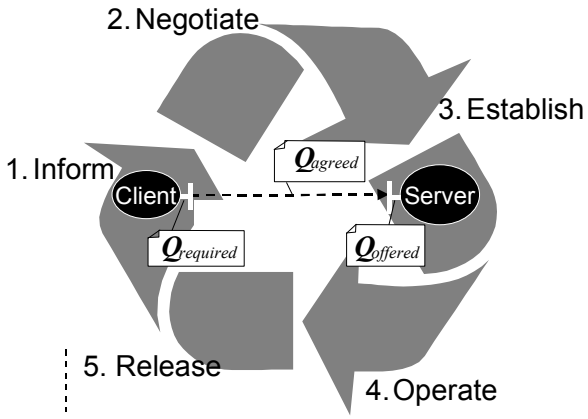


Figure 6-1 QoS support lifecycle in QPS

In the inform phase the client specifies its $Q_{required}$ and the server specifies its $Q_{offered}$. During the negotiate phase, QPS initiates a three-party negotiation between the client, the server and the DRP to reach an agreement. A successful negotiation results in a Q_{agreed} which is then associated with the binding. During the establish phase, QPS commits the resources that have been allocated during the previous phase. These can be communication, storage and processing resources.

Once sufficient resources have been allocated to the binding, Q_{agreed} must be maintained, to support QoS control transparency. Which means that QPS corrects drifting quality levels, for example, by re-allocating system resources. This is the operate phase. Finally, when the client does not further need the binding or when radical changes in the DRP make it impossible to sustain Q_{agreed}, the system resources are released.

In the remaining sections we focus on phase 2,3 and 4. For the realisation of phase 1 QPS uses the meta-model concepts discussed in Chapter 5. Realisation of phase 5 is a matter of proper administration of the resources allocated to a binding to be able to release them.

6.1.3 Framework for QoS negotiation

The design of QPS for phase 2 is constrained by two conflicting forces: a) the design has to be flexible enough such that we can incorporate various negotiation strategies, b) the design has to be stable enough to ensure robustness and portability.

Bond and Gasser [BoGa88] regard a negotiation as a process by which conflicts (with respect to resource allocations) may be resolved. However, we share the view presented by Dillenbourg and Baker [DiBa96] that the existence of a 'conflict' is not essential to the definition of negotiation. All that is basically required is that the interacting objects possess the mutual goal of achieving agreement, with respect to some set of negotia, or entities of negotiation. Usually, several dimensions of negotia are negotiated simultaneously.

Negotiation is an activity that must take place after the deployment of application components. Establishment of QoS agreements at design time it too limited for open distributed systems. In some cases the QoS level of application components that execute on top of a real-time operating system is calculated off-line. Off-line schedulability analysis [XuPa90, AbSh95] is used to verify that the resources are sufficient to meet all QoS constraints. In a distributed system it is not possible to calculate QoS levels off-line, as this requires worst-case conditions to be known at design time.

To establish a QoS agreement, an agreement must be reached between the client application object (represented by a client BEO), the server application object (represented by a server BEO) and the DRP. This agreement is reached through a negotiation process that is performed according to a negotiation model.

Negotiation model

In our QoS negotiation model, an object has either a *client* or a *server* role. We associate a certain QoS level with a *binding* between a client and a

server. The binding is a binary relation denoted as $b(c,s)$, where c is a client, and s is the server.

The QoS level of a binding depends on several factors, such as the network situation, the object implementation or other resources that the object depends on. The client specifies a required QoS level denoted as $Q_{required}(c)$ for a binding $b(c,s)$. The server is associated with the offered QoS level, denoted as $Q_{offered}(s)$. Associating a required or offered QoS for objects is mandatory, both on the client-side and on the server-side. The QoS level that is associated with the binding is a result of a negotiation between the client and the server. The negotiation is the process of reconciling the potentially diverging $Q_{required}(c)$ and $Q_{offered}(s)$, for a binding $b(c,s)$. A successful negotiation results in an *agreed* quality level $Q_{agreed}(c,s)$ also denoted as $Q_{agreed}(b)$ where $b(c,s)$.

Negotiation strategies

In [Ko97], several QoS categories are identified, such as *reliability*, *performance*, *availability* and *security*. An infrastructure designer can define a *QoSContractType* for each of these categories in accordance with the QoS meta-model described in Chapter 5.

Consider the performance category as an example to illustrate negotiation strategies. A QoS contract type that has dimensions delay and rate represents the performance category. *Delay is* the time that elapses between the request and response of an invocation and *rate* is the number of invocations that are responded by a server within a time unit. To be able to compare different quality levels, an ordering "better than" is necessary for the value domains of dimensions. The *rate* dimension has an *increasing* direction type, which means that higher values for rate are better than lower values. The *delay* dimension has a *decreasing* direction type, which means that lower delay values are conceived as better QoS. The direction type determines how QoS dimensions should be compared.

Negotiation can take place between objects that are going to be bound together. The negotiation process is defined per dimension. It starts with the two parties having a required and an offered QoS of the same dimension. The required and the offered quality levels are compared, and an agreed quality level is calculated. An agreement can only be reached if there is a common range of values between the dimension of the offered and required QoS. The QoS agreement is then used as a target level for the binding that must be maintained during the operate phase.

Calculation of the agreed quality level can be performed according to different strategies. It may also take into account other conditions such as a price limit or alternative $Q_{required}$ specifications. The default strategy is straightforward and defines the agreed quality level to be the $Q_{required}(c)$, where $Q_{required}(c)$ is not better than $Q_{offered}(s)$, for a binding $b(c,s)$. Negotiation

fails when $Q_{required}(c,s)$ is better than $Q_{offered}(s)$, or if the server provides no offered QoS. QPS allows for the configuration of alternative negotiation schemes through the strategy pattern [Ga+95].

Negotiation framework

We have identified two sub phases during the negotiation phase: a) a server BEO negotiates with the object middleware, in order to achieve an offered QoS, b) a client BEO initiates a negotiation using its required QoS and the offered QoS (that results from sub phase a) to converge to a QoS agreement.

Negotiation in sub phase a) starts with the $Q_{offered}$ that a server object gets assigned by the application designer during the computational design. A $Q_{offered}$ is what a server object intends to offer to a client object. In this phase the server object acts as a QoS user and the middleware as a QoS provider. The middleware decides whether sufficient resources are available to support the $Q_{offered}$ of the server object.

In a more elaborate scheme, an application designer may provide several alternative $Q_{offered}$ specifications, each with an associated benefit or utility value. Such a specification seems suitable as a starting point for graceful QoS degradation in case of system overload. A QoS negotiation algorithm for this approach is discussed in [AAS02]. However, for QPS we assume a boolean approach, in which the middleware either accepts or rejects a $Q_{offered}$ from a server object.

Negotiation in sub phase b) starts from a client that requests a QoS agreement that satisfies the Q_{agreed}, $Q_{offered}$ and the available resources that the object middleware can assign to the binding between that client and server object.

The client takes explicit actions to initiate the negotiation. The negotiator is an entity that implements a set of algorithms that determine the values of QoS parameters and resource reservations acceptable to all involved parties. The negotiator is an entity that is inspired by the whiteboard negotiation approach [PTM92], where a whiteboard acts as a central repository for the whole negotiation phase (see Figure 6-2). It contains the configured problem-solving strategy as well as the knowledge base, and all objects involved in the negotiation access it.

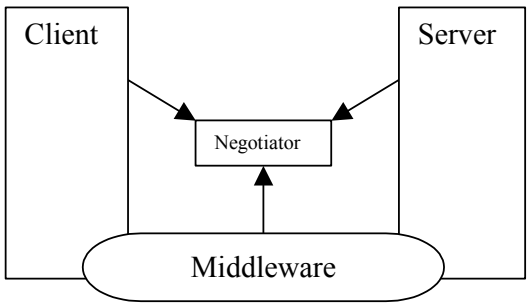


Figure 6-2 The negotiator is the whiteboard for negotiating parties

The whiteboard is so named to distinguish it from the traditional view of a blackboard [Ni86]. As in the blackboard architecture, negotiation objects contribute incrementally to the whiteboard in building a solution (i.e., a problem solver). In contrast to blackboard systems, however, the negotiating objects cannot access the whiteboard opportunistically. Instead, access must follow a fixed order of events.

Negotiation steps

QPS assumes that a QoS specification can be decomposed into atomic expressions, each related to one or more QoS dimensions. Such decomposition allows a specific negotiator to extract information from the QoS specification that is relevant to the QoS service it belongs to.

The generic negotiator performs negotiation in the following steps (Figure 6-3):

(a) Collect required and offered QoS specifications;
(b) Find relevant specific QoS negotiators;
(c) Delegate the negotiation to the specific negotiators;
(d) Assemble the resource reservations of the specific negotiators;
(e) Perform reservations;
(f) Assemble the agreed QoS specification;

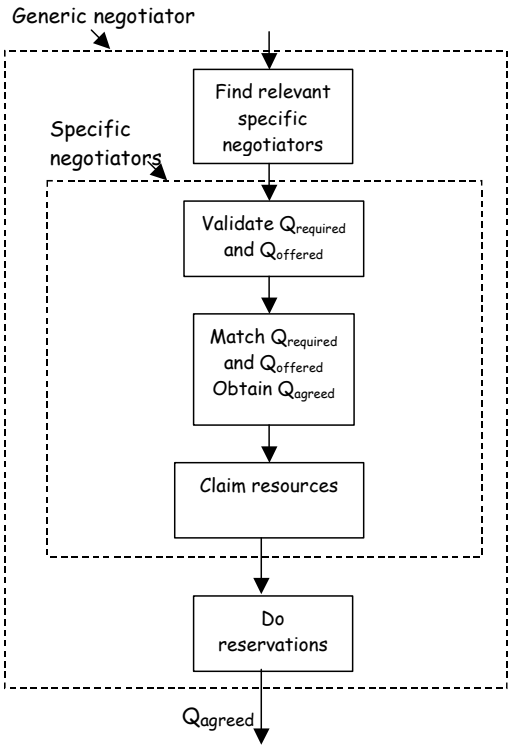


Figure 6-3 Negotiation algorithm

In Figure 6-3, the generic negotiator delegates the QoS specification to the specific negotiators. A specific negotiator is responsible for validating the part of the QoS required and QoS offered that is related to QoS service associated with this specific negotiator. A specific negotiator performs matching of the relevant parts of the required and offered QoS and the result is a specific agreed QoS. Once a specific agreed QoS is reached, the specific negotiator claims the resources needed to achieve the agreed QoS level. These claims prepare reservation structures that contain all necessary information for establishment of reservations. The generic negotiator uses these reservation structures to perform the actual reservations.

6.1.4 Framework for establishment and maintenance of QoS

Phase 3 and 4 of the binding lifecycle are concerned with the establishment and maintenance of a QoS agreement. This section introduces our approach and subsequently discusses a specialisation of a generic control system model that coincides with the operate phase of the QoS support life cycle.

Control framework for QoS provisioning

The design of QPS for phase 3 and 4 of the QoS support lifecycle is constrained by two conflicting requirements: a) the design has to be flexible enough such that it enables us to experiment with different QoS strategies and cope with different kinds of application demands; and b) certain aspects of the design have to be fixed so that the robustness and portability of the design can be guaranteed.

For this reason we start off with a generic control system model, which we specialise, such that it applies to QoS-control of a QoS agreement during the operate phase. This specialised model forms our framework, i.e. the fixed part of our design. Although some decisions are made with respect to the scope of control, the framework is independent of any specific QoS-control strategy or algorithm. Therefore, different solutions can be compared and evaluated with this framework.

A possible way to arrive at a complete QoS-control design, e.g. for a specific object middleware platform, is to apply a synthesis-based approach [Te00]. In this approach, requirements are converted into technical problems. For each technical problem, possible solution techniques are sought. The candidate solution techniques are then compared with each other from the perspective of relevance, robustness, adaptability and performance. Whenever a suitable solution technique is found, the fundamental abstractions of this technique are used to refine the design. This process is repeated until all the problems are considered and solved. Finally, the architectural abstractions are specified and integrated within the overall framework. Since solution domain knowledge changes smoothly, this approach provides us with stable and robust abstractions with rich semantics. The discussion of technical issues in section 6.1.5 follows this approach.

Generic control system

The main objective of the QPS operate phase is to establish and maintain an agreed QoS that satisfies the demands of applications and the capabilities of available resources. This objective can be fulfilled by a control system in two phases: (a) establishing an agreed QoS corresponds to setting up the desired parameters of the control system; and (b) enforcing the agreed QoS can be realised by controlling actions. The control system must be naturally embedded in the middleware system. Our QoS-control framework should therefore be synthesised from the fundamental abstractions of middleware and control systems.

A *control system* [Le90, KiGi87] consists of a *controlled system* in combination with a *controller*. The interactions between the controlled system and the controller consist of observations and manipulations performed by the controller on the controlled system.

Figure 6-4 shows the elements of a control system.

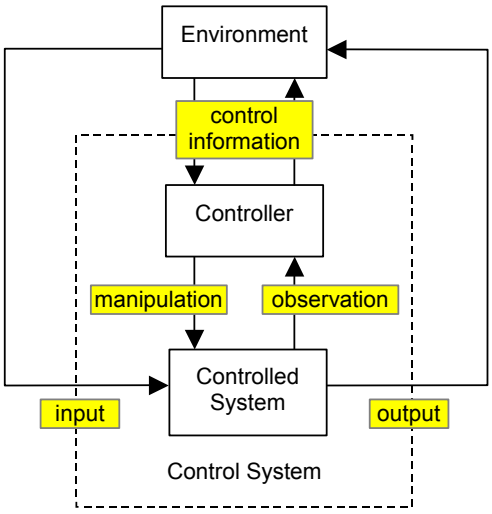


Figure 6-4 Elements of a control system

The generic control model abstracts from the type of observation and the type of manipulation that can be employed by the controller on the controlled system. The relationship between the controlled system and the controller can be realised using different strategies. With a *feed-forward control* strategy manipulation through control actions is determined based on observation of the input to the controlled system. The controller steers the controlled system in such a manner that the controlled system delivers the desired behaviour. A *feed-back control* strategy can be applied for behaviour optimisation. According to this strategy, measurements of the output delivered by the controlled system are compared with a desired behaviour (a *reference*) and the controller uses the *difference* between them to decide on the control actions to be taken.

QoS-control system

In QoS aware middleware, the controlled system is the middleware functionality responsible for the support of interactions between application objects, while the controller provides QoS control capabilities and is embedded in the middleware platform. Here, the environment represents the operational context of the middleware, which consists of application objects with QoS requirements and QoS offers. The middleware platform encapsulates the computing and communication resources at each individual processing node, which may be manipulated in order to maintain the agreed QoS.

Figure 6-5 shows the specialisation of the generic control model for controlling the QoS provided by a middleware.

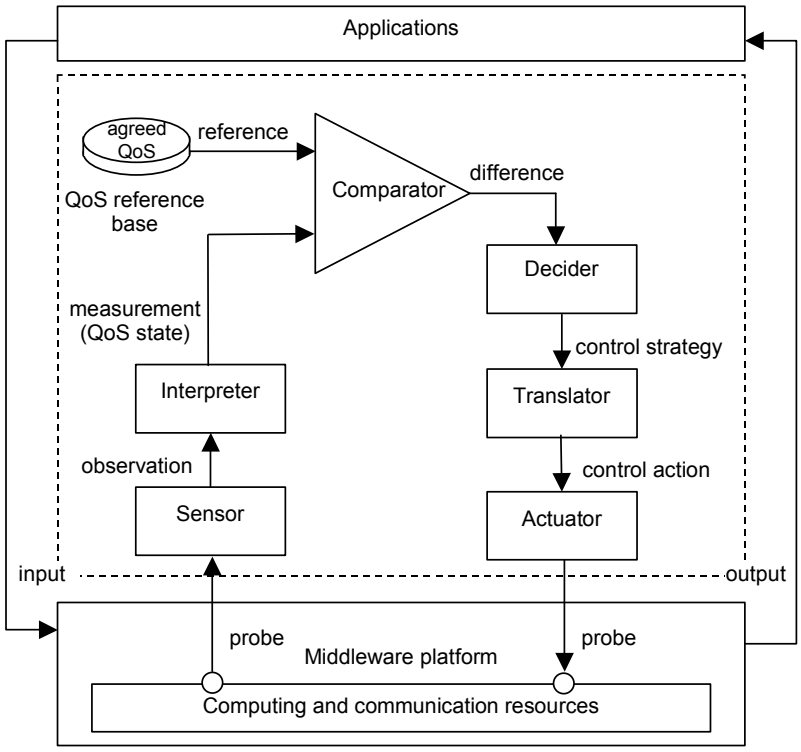


Figure 6-5 QoS-control system in a middleware context

In Figure 6-5 we identify two symmetrical structures, one for handling QoS measurement concerns and another for handling QoS manipulation concerns. A *probe* is a point of observation or manipulation that is available or must be planted in the controlled system, i.e., the middleware platform. Many probes may be planted in the controlled system, for both observations and manipulations.

A *sensor* is a mechanism that uses a probe to obtain observations. Multiple sensors may be used in a control system, e.g., one for each probe type. Observations can only be useful if they are interpreted in terms of measurements that can be compared with the reference, i.e., they are represented using the same units and have the same semantics. For example, observations can be the moments in time of the sending of a request and the receiving of the corresponding response. The needed measurement could be the average response time, which implies that the average of the difference between the moments in time observed is

calculated in order to generate the measurement. An *interpreter* performs this calculation. In general, the interpreter combines observations, which could even come from different sensors, in order to generate measurements.

A *comparator* compares the measurement and an associated reference value (an agreed QoS measure), determining the difference. A *decider* gets the difference and applies some algorithm to establish a control strategy, consisting of the objectives to be reached in this execution of the control loop. The control strategy must be translated in a collection of control actions, i.e., manipulations of the controlled system. A *translator* is responsible for translating the control strategy to a collection of control actions. An *actuator* schedules the control actions such that they are carried out using one or more probes. Multiple actuators may be used in a control system, e.g., one for each type of probe. The translator distributes the control actions among the actuators, realising in this way the control strategy.

Since we intend to use our design mainly to investigate mechanisms for controlling QoS through the middleware platform, we have not introduced any facilities that control applications objects. Furthermore, controlling applications requires either specific knowledge about the applications, which prohibits any general solution, or it requires the applications to offer certain adaptability or manipulation interfaces, which imposes a serious restriction on the applications that could use our QoS provisioning service.

6.1.5 Technical issues

This section identifies and elaborates the technical issues that have to be addressed in order to realise the QoS control design.

Identification of requirements

Referring to Figure 6-5, the following requirements and problems can be identified when developing a more concrete QoS-control design, e.g., to suit a specific application or system environment:

1. Collecting observation values.
2. Interpreting the observation values to create a measurement.
3. Determination and representation of the difference.
4. Executing a controlling algorithm.
5. Applying a control strategy and performing middleware manipulations.
6. Feasibility of the overall control loop.

We explore these requirements in the sequel, and propose some possible solutions and solution strategies.

Collecting observation values

In order to collect observation values we have to develop probes and sensors. Probes connect the middleware to the control mechanisms and are independent of the actual measurements. Sensors collect the actual measurements and they typically depend on the amount and types of data that are collected.

The fundamental requirement on the probes and sensors is that they must have a minimal impact on the middleware platform. More detailed requirements are:

1. *Minimal impact on the code*: the insertion of probes into the middleware platform should have little or no impact from the manageability point-of-view. In other words, we need non-invasive addition of probes. A special case is the run-time insertion and removal of probes, which may also have beneficial performance consequences.
2. *Mapping probes to the controlled system:* typically, it may appear that for certain types of measurements a particular probe must be inserted in a multitude of places of the middleware implementation. This problem is a special case of *crosscutting of concerns* [KLM+97].

Reflection is a technique in which a system is explicitly represented in terms of a meta-object, allowing one to manipulate the (structure of the) system by manipulating its meta-object. A reflection-based approach suits well to the collection of observation values, as shown in Figure 6-6.

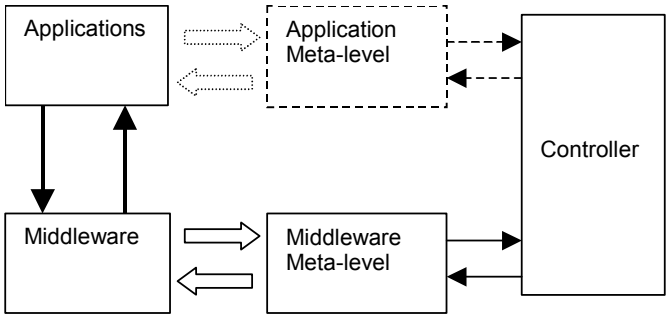


Figure 6-6 A reflective approach to the QoS-control design

Figure 6-6 presents explicit meta-objects of the middleware. Such an approach allows one to observe and manipulate the middleware platform, even in a non-intrusive way. The manipulation could be based on observation of the middleware itself and its inputs and outputs, as well as the QoS specifications that are provided by the applications. Although observations of the applications are represented in Figure 6-6, they are further ignored in this thesis.

Reflective languages or reflective language features, such as, e.g., Java introspection, can implement reflection in various ways. A language that supports the full power of reflection has the benefit that observation and manipulation probes can be installed without affecting the base level code. However, it may also incur significant performance overhead, in some cases even when no reflective features are active.

To manage the cost of performance overhead, we assume that tailored implementations of the observation and manipulation probes are necessary. Individual probe implementations can make a trade-off between modularity and flexibility on one hand, and performance overhead on the other hand. Examples of different probe implementations are reading variables, function calls inserted in the code, callback methods, the Observer pattern [Ga+95], and many more.

Crosscutting of concerns requires either careful documentation and management of probe insertion points, or entirely new tools and techniques for specifying and implementing crosscut concerns. Recent work in the area of Aspect-Oriented Programming [KLM+97] addresses these issues.

Interpretation of observations

The interpretation process depends on many factors: the involved observation data, the required measurements, and the rules or strategies for interpretation. The number of interpretation rules and their complexity also determine the interpretation process.

The interpretation part should not become a possibly large collection of unstructured ad-hoc code. This implies that a generic model should be developed to define how observations are translated to measurements, such that interpretation code can be generated as automatically as possible. In case statistical information determines measurements, a lot of input data may be required, such that the amount of storage and processing should be reduced as much as possible.

The interpretation process is essentially a transformation from a set of input values to a set of output values. The variation in input values lies both in sources, types and time, and depends on the sources of the input, i.e., how the middleware has been designed. The resulting output should be independent of the specific implementation details of certain middleware and applications, and it should be suitable for the comparison process. A suitable run-time representation of our QoS meta-model described in Chapter 5 should be available to represent the types and values of both measurements and references.

The interpretation of observations can be done through calculations, heuristics (logic rules), stream interpreters or conversions. We need to model these different techniques in a uniform way, with explicit dependency relations to a structured representation of the observations and

measurements. Interpretation rules should all be a specialisation of a single abstraction, i.e., the interpreter. Each individual instantiation can be considered as a *micro-interpreter*. For each QoS measure, there should be a clear specification of the interpretation rules in terms of formulas or guidelines.

Many algorithmic and data structure optimisations are possible. The most effective optimisations depend on the required output (i.e., the required measurements). For example, for collecting the average of a large set of values, it is not necessary to store all these values, but we can just remember the sum of all the values and number of values. In some cases overhead can be reduced by, e.g., adopting sparse data structures.

Figure 6-7 shows the extensions to our design to meet the needs of interpretation.

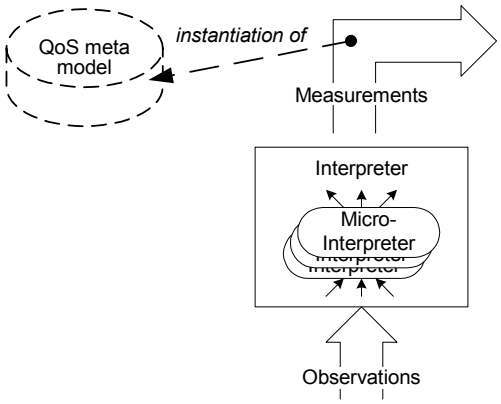


Figure 6-7 Details of the design related to the interpretation of observations

Determination and representation of the difference

The comparator compares the measurement with the reference model and determines the difference. This comparison can vary from subtraction in the simple case of one QoS characteristic with a numeric value, to complex calculations possibly using heuristics in the case of multi-faceted QoS characteristics. The main task of the comparator is to deliver an abstraction of the 'problem to be solved' that is as independent from the implementation details of the environment as feasible.

The difference produced by the comparator serves to detect (potential) violations of the QoS. Such violations depend on the agreed QoS. Hence, the difference must be obtained by comparing the actual measurements with corresponding references specified by the agreed QoS.

The difference could be represented as a 'distance' vector, where each element of the vector corresponds to a relevant QoS dimension.

Measurements and references should be described in such a way that they can be compared. For this purpose we apply the QoS meta-model discussed in Chapter 5 to specify both the measurement and the reference, and the difference. Another benefit of having a QoS meta-model is the ability to build QoS specification repositories.

Figure 6-8 illustrates the use of the QoS meta-model.

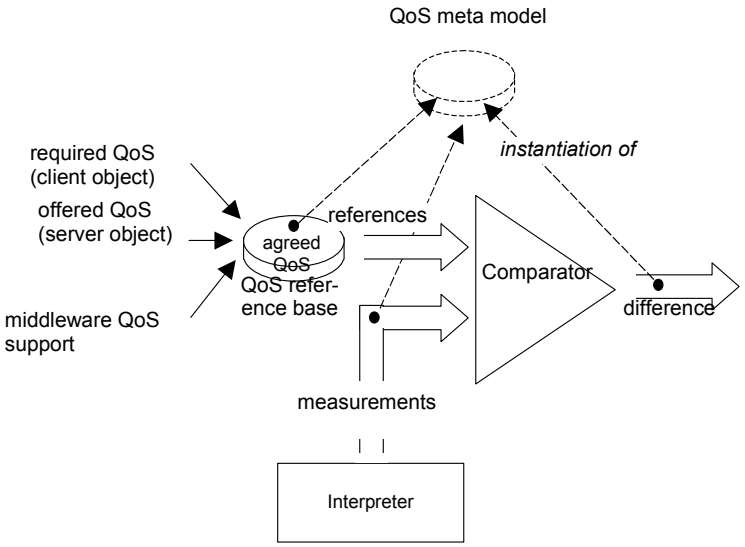


Figure 6-8 Details of the design related to the difference of measurements and references

The agreed QoS is determined before entering the operate phase, through negotiation based on QoS requirements, QoS offers and the capabilities of the middleware platform. We assume that the agreed QoS is not modified during the operate phase.

Controlling algorithm

The difference or distance vector computed by the comparator may define a situation that requires controlling (i.e., correcting) actions to be taken. The controlling algorithm is responsible for selecting an appropriate strategy. The strategy to be chosen depends partially on the specific state and configuration of the middleware. Rather than mixing middleware state and configuration information with the measurements and difference, this information must be available independently. For this reason, we introduce a *middleware control model*. This model is an abstraction (model at a meta-level) of the middleware, which specifies what can be parameterised or tuned in the middleware, or which components can be plugged in, deactivated and activated.

The goal of the controlling algorithm is two-fold: firstly to ensure that the agreed QoS can indeed be supported by the middleware platform, and secondly to optimise the overall QoS characteristics, by balancing the different, often contradictory, requirements. In its most general form, controlling is an artificial intelligence task that involves domain knowledge and heuristics about managing and controlling QoS, and the interdependencies between QoS characteristics.

We have not selected a particular solution for the controlling algorithm, but give here some options, some of which may be used in combination:

- The controlling algorithm may be implemented as a set of heuristics, e.g., as a small rule-based expert system;
- Fuzzy logic may be suitable for expressing and reasoning about weak but conflicting optimisation goals for the various QoS characteristics [MaAs75];
- Based on behaviour and control theory, a combination of mathematical computations and algorithms may be used to select the most appropriate control strategy;
- For each element of the middleware control model we may provide a set of alternatives or ranges of parameter settings, and annotate these with statements that define how the various QoS characteristics influence each other. The permutations of the possible alternatives form a design space from which one can select an optimal configuration. The resulting alternatives and settings can determine the resulting control strategy.

Figure 6-9 shows the extension of our design with an explicit middleware control model.

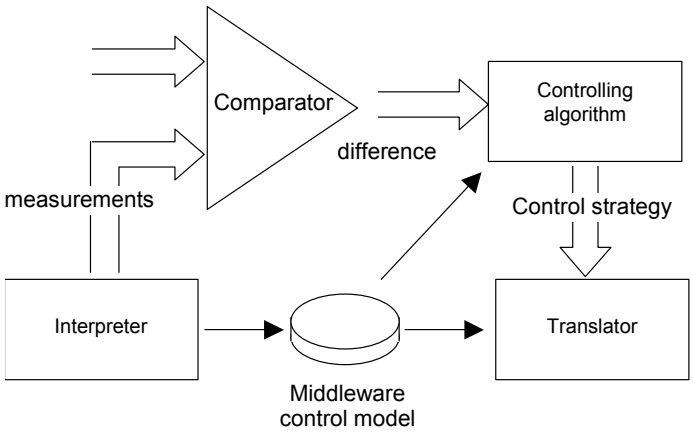


Figure 6-9 Details of the design related to the controlling algorithm

Control strategy and middleware manipulation

A control strategy is the output of the controlling algorithm, and it should be an implementation-independent representation of the solution strategy for maintaining a QoS agreement. Control strategies are strongly related to the controlling algorithm.

Control actions are abstractions that represent concrete functional behaviour, but are independent of the implementation details of the specific middleware software. Control strategies represent sets of control actions that are to be applied to the middleware in a co-ordinated way. The representation of control strategies must consist of at least the following parts: a) set of control actions; b) a set of actuators in the middleware where the control actions can be applied, and c) a co-ordination specification, which could be a script or any other form of executable specification.

There are a few ways to affect the behaviour of a running system like a middleware platform: a) by invoking operations of a local API; b) by modifying the internal state of the system, c) by replacing components of the system with different implementations, and d) by meta-level manipulation of the system itself. A control action is a specialisation or instantiation of one of these.

We consider the use of APIs as an important and feasible approach, but it relies on fixed, static assumptions about the ways of manipulating the middleware, which cannot be always guaranteed. Directly manipulation of the internal state is undesirable from an object-oriented software engineering point-of-view, and should be achieved indirectly by one of the other approaches. Replacement of components is an interesting alternative, as it allows for the dynamic replacement of behaviour. The use of meta-level facilities can be beneficial, but its suitability depends strongly on the abstraction level used to develop these facilities. Furthermore, meta-models should be structured in terms of well-defined meta-spaces, avoiding in this way the proliferation of ad hoc meta-models (see [BlSt97]).

The implementation of control actions through actuators introduces technical issues comparable to the ones discussed before, and therefore they are not discussed further.

Feasibility of the overall control loop

The performance overhead introduced by our architectural framework has to be carefully considered when using the framework in practical settings. The technical solutions should not make the overall QoS worse than what it would be without them. Several QoS requirements are related to performance (e.g., delays and throughput). Implementations of our design may require a lot of additional activities and overhead, which may conflict with the QoS agreements they try to maintain. By adopting a tailorable

framework approach, we may choose to build instances of the framework with components ranging from simple, low-overhead components up to complex components, which can help coping with the performance overhead introduced by the control loop.

Feed-back control loops may make the controlled system oscillate between two undesirable states, depending on the corrective measures and their effects. In some cases, mathematical models based on control theory can help predicting whether the system is stable during operation, allowing one to avoid oscillation. In case mathematical models are not available or are not precise enough, some heuristics may show whether the system is stable or not. Alternatively, additional (meta-level) controllers could be introduced to detect instability and take measures to avoid it, e.g., by actuating on the controlling algorithm. The use of fuzzy logic in the controlling algorithms may also help avoiding that the control loop oscillates during operation.

6.2 Engineering view of QPS

This section discusses the engineering objects of QPS. The discussion starts with an overview of the path that an object invocation takes and reviews the layers that an invocation traverses. From this discussion follow the layers that are controlled by QPS. The remainder of the discussion concentrates on the client-side objects and the server-side engineering objects that collaborate to provide the services of QPS.

6.2.1 End to end view

The QoS of a binding between a client and a server object is determined by the quality agreements that are achieved at the various layers of the object communication middleware.

An object invocation is initiated on the client side and traverses from the client object through the object interaction layer, the message distribution layer and then through the network adaptation layer down to the network. The object interaction layer converts an object invocation into a request message, which is then conveyed by the message distribution layer. The message distribution uses the network adaptation layer for the actual transportation of a message. The network adaptation layer ensures that a message is transported using a connection oriented reliable data flow.

On the server side the data received from the network traverses up these layers in a reverse order. This results in a received message at the message distribution layer and an invocation from the object interaction layer to the server BEO.

After processing the invocation the server BEO returns a result, which again traverses down the layers to the network on the server side and up the layers on the client side. Traversing the layers of the object communication middleware up and down at the client and server side is needed for one object invocation. Each layer affects the achieved QoS of that object invocation.

Following the integration principle, as discussed in Chapter 5, QPS must establish and control QoS agreements at each layer in order to ensure an end-to-end QoS agreement between a client and server BEO.

An agreed QoS (Q_{agreed}) between a computational client and server object corresponds to an agreed QoS at the object interaction layer. QPS must map this Q_{agreed} to a QoS agreement at the message distribution layers, which again must be mapped to a Q_{agreed} at the network adaptation layer.

Figure 6-10 shows the flow of interaction data, the interactions between the layers of the object communication middleware and related QoS agreements at the various layers.

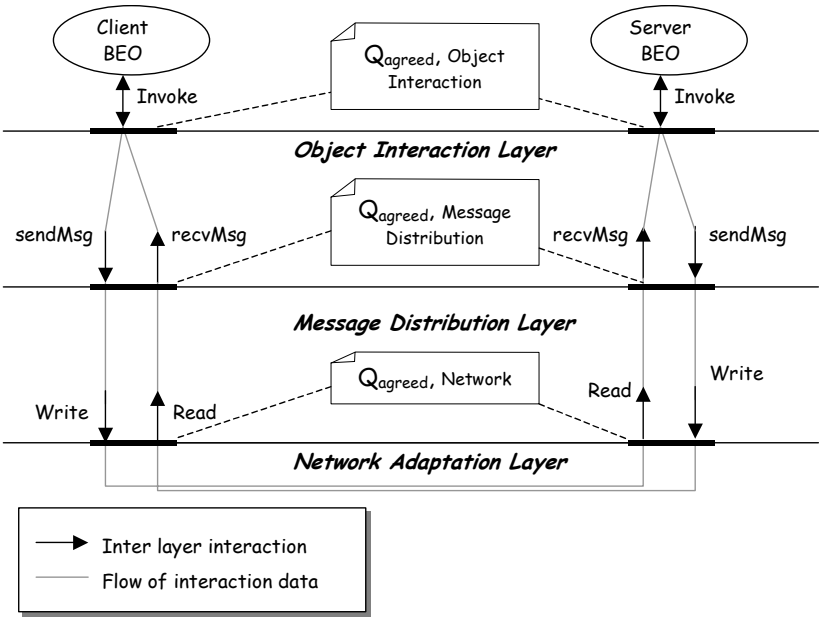


Figure 6-10 Flow of interaction data and associated QoS agreements

The dimensions and units used in the QoS agreements shown in Figure 6-10 often differ from layer to layer. For example, consider a performance QoS type that defines the rate in number of *invocations per second* to which the Q_{agreed} at the object interaction layer is associated, so the object interaction layer must support a minimum number (say X) of invocations

per second. Assume further that one invocation results in sending two and receiving one message by the message distribution layer. As a result, QPS must establish a Q_{agreed} with the message distribution layer that supports at least 2X *messages per second* for sending a message and at least X messages per second for receiving a message. Depending on the size of the messages sent and received, the message distribution layer must establish a Q_{agreed} with the network adaptation layer in terms of *bytes per second* that must at least be transmitted or received.

In case the message distribution layer does not support the QoS agreements, QPS may bypass this layer and directly map a Q_{agreed} at the object interaction layer to a Q_{agreed} at the network adaptation layer.

The entities to which a QoS contract can be associated depend on the layer where the QoS contract applies. At the object interaction layer a QoS contract is associated with either an interface or with the individual methods that this interface defines. At the message distribution layer a QoS contract is associated with a sequence of messages. At the network adaptation layer, a QoS contract is associated with a data flow or the sequence of bytes that constitute that data flow.

6.2.2 Client side objects

QPS objects on the client side are: the client QoS repository, the client negotiator, a binding factory and binding control objects

The client QoS repository is used to store the required QoS of a client BEO and is invoked during the inform phase

The client negotiator takes the required QoS and invokes its peer negotiator on the server side. The detailed operation of the negotiation is explained when the server negotiator is discussed in section 6.2.3. For now it is sufficient to know that the client negotiator is invoked during the negotiate phase and if phase is successfully completed it results in a QoS agreement at the object interaction layer. This QoS agreement is mapped to QoS agreements at the message distribution and network adaptation layers. This results in a QoS agreement for an end-to-end QoS.

Once the QoS agreements for an end-to-end QoS have been created, a binding manager establishes the QoS agreements, thus completing the establish phase (phase 3) and then creates a binding control object.

The binding control object is concerned with the operate phase of the QoS binding lifecycle. This object takes part in the QoS control loop as discussed in section 6.1.4. It compares QoS measurements with the QoS agreements and applies a control strategy to maintain the QoS agreements. In fact, it provides the functions for the interpreter, comparator, decider and translator building blocks depicted in Figure 6-5.

Figure 6-11 shows the client side of the QPS. The numbers in the figure relate to the QoS binding life cycle discussed in section 6.1.2.

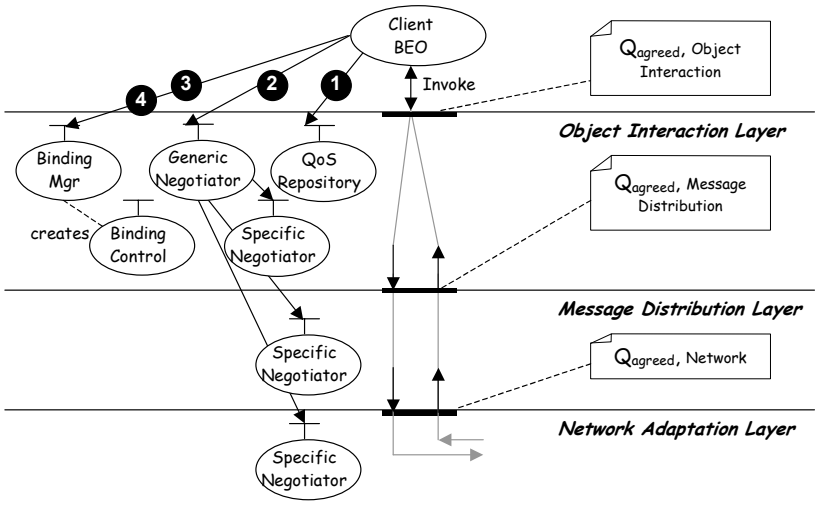


Figure 6-11 Client side QPS objects

The figure does not show the probes, sensors and actuators as found in our QoS control design discussed in section 6.1.4. These objects are specific to the QoS dimension, which is to be measured and controlled, and if needed are found in all layers. The sensor and actuator objects interact with the binding control object, to maintain the agreed QoS.

6.2.3 Server side objects

On the server side a local QoS repository is used to store the offered QoS of the server BEO. This repository interacts with the QoS object manager to determine if the QoS that the server intends to offer can be supported. The QoS object manager is a special case of an object manager as found in the object middleware reference model. It acts also as access controller to determine if an offered QoS can be admitted if it is feasible.

Each layer has a specific negotiator. These negotiators interact with the generic negotiator positioned at the client side during the negotiate phase.

Figure 6-12 shows the server side of the QPS. The numbers in the figure relate to the QoS binding lifecycle discussed in section 6.1.2.

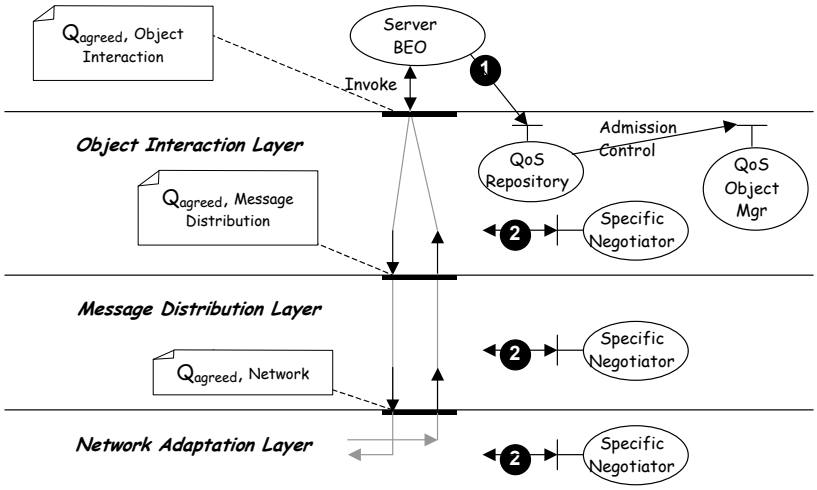


Figure 6-12 Server side QPS objects

The figure does not show the probes, sensors and actuators found in our QoS control design as discussed in section 6.1.4. These objects are specific to the QoS dimension, which is to be measured and controlled, and if needed are found in all layers. The sensor and actuator objects interact with the binding control object that resides on the client side, to maintain the agreed QoS.

6.3 Transformation of QPS to CORBA

This section discusses the implementation of the QPS design in a CORBA 2.3 context. The discussion concentrates on the client-side objects, server-side objects and the CORBA specific interfaces that have been used to implement the services of QPS.

The QPS implementation is available as an open source project from http://quamj.sourceforge.net/.

6.3.1 Client side objects

On the client side, QPS has a QoSRepository (QR), a Client Interceptor and a transport plugin, called QIOP. Figure 6-13 depicts these client-side objects. Below we describe the QoSRepository and the QoS Client Interceptor. The QIOP protocol plugin is described in Section 6.5.

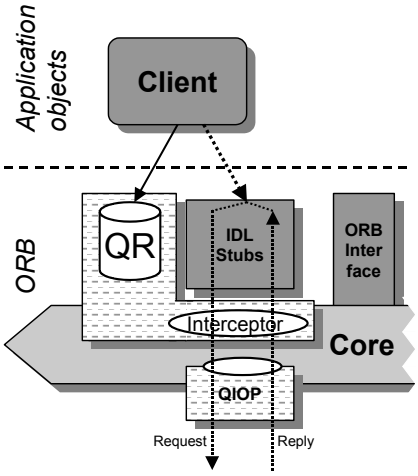


Figure 6-13 QPS client side objects

QoS Repository

The QoS Repository is implemented as a client-side specific CORBA service. A reference to a QoS Repository object is obtained by client applications by calling the `resolve_initial_references` operation on the ORB, with the identifier parameter set to "`QoSRepository`". The QoS Repository interface is defined as follows:

Figure 6-14 The QoSRepository Interface

```
interface QoSRepository {

  Object  set_required_qos(in Object o, in string qos );

  void negotiate_qos( in Object o );

  string  get_agreed_qos( in Object o );

  string  get_required_qos( in Object o )

  boolean agreed_qos( in Object o );

};
```

A client registers a required quality level for server object *o* by calling `set_required_qos`(o,$Q_{required}(o)$). QoS specifications are passed as strings containing an XML specification of the required QoS. Internally, the XML specification is validated against the meta-model presented in Chapter 5, based on the MOF XMI mapping rules. The return value of the operation is an object reference to the same object as the argument, but with different

policies set for this object. The registered required QoS levels can be obtained by the `get_required_qos` operation.

The client can initiate QoS negotiation for object *o* explicitly by calling `negotiate_qos(o)` on the QoS Repository. As a result, the QoS Repository sends a request "`negotiate_qos`" to object *o* with the previously registered required QoS as input parameter. This is a Dynamic Interface Invocation (DII) request that uses the object reference as a target.

Registration of an agreed QoS is an internal operation of the QPS service. Therefore, no operation is exposed through the QoS Repository interface. At any time in the lifetime of a binding, the client can register a new required QoS level and request a negotiation. The `agreed_qos` operation returns true, if an agreement is achieved for the last registered required QoS level. Otherwise, it returns false.

QoS Client Interceptor

QPS registers one QoS Client Interceptor instance during initialization of the ORB, so that all calls made by clients are intercepted. The interception points as defined by the Portable Client Interceptor interface [HNNW99] are `send_request`, `send_poll`, `receive_reply`, `receive_exception` and `receive_other`. Currently, the `send_request` interception point is re-implemented in order to initiate QoS negotiation. If a request is intercepted that has a target object for which a required QoS is registered, but no QoS level is negotiated, the Client Interceptor instructs the QoS Repository to perform the negotiation.

6.3.2 Server side objects

On the server side, QPS has a dedicated Portable Object Adapter (QOA), a ServantLocator, a Negotiator and a QIOP transport plugin. Figure 6-15 depicts the server side objects.

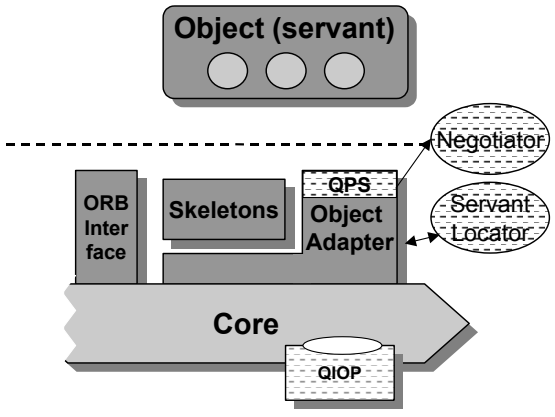


Figure 6-15 QPS server side objects

QoS Object Adaptor

So far, in this chapter we have assumed that server objects can define their offered QoS. A server object is also what a client application sees and interacts with. In CORBA however, on the server side, servants implement the server objects. It is even possible that several servants implement the same object.

One can argue that the QoS should be associated with a servant rather than an object, since different implementations of the same object may offer different QoS levels. In order to keep our implementation simple, in QPS, we require that only one servant implements a QoS aware CORBA server object.

Servers that expose objects with QoS support use a special Portable Object Adapter (POA) called the QoS Object Adapter (QOA). The QOA registers servants with offered quality levels and takes care of the routing of requests to these servants. The QOA therefore holds $<o,\ servant,\ Q_{offered}(o)>$ tuples where o denotes an object.

To ensure the static object/servant association, the QOA is created with the policies `PERSISTENT, USER_ID, USE_SERVANT_MANAGER, NON_RETAIN` and `NO_IMPLICIT_ACTIVATION`. These are actually parameters at the creation time of the QOA. The `USE_SERVANT_MANAGER` and the `NON_RETAIN` policies together ensure that the QOA uses the QoS Servant Locator to locate the servants of the incoming requests. The `NO_IMPLICIT_ACTIVATION` policy disables implicit object activation. The `PERSISTENT` and `USER_ID` policies are necessary for the static object/servant association.

The QOA extends the standard POA operations with two additional operations shown in Figure 6-16.

Figure 6-16 The QOA interface

```
interface QOA : PortableServer::POA {
  void register_servant_with_id(
    in PortableServer::Servant servant,
    in PortableServer::ObjectId oid )
  void set_offered_qos_for_id(
    in PortableServer::ObjectId oid,
    in string offered_qos )
};
```

The `register_servant_with_id` allows QoS aware objects to register themselves with an objectId. Servants invoke the `set_offered_qos` operation to register their offered qos.

QoS Servant Locator

In CORBA version 2.3 and later, request dispatching to the servant that implements a CORBA object can be dynamic and is managed by the servant manager that each POA possesses. The POA interface offers two kinds of servant managers: Servant Locators and Servant Activators. The POA passes the ObjectId of the target object to its servant manager, expecting it to return a servant that incarnates the object. The QOA uses a customised Servant Locator called *QoS Servant Locator* that is derived from the default Servant Locator interface. This interface defines two operations: `preinvoke` and `postinvoke`. The `preinvoke` operation is called on the Servant Locator before the request is dispatched to the object that handles the request. The servant returned by this operation shall handle the request. Similarly, after the request is handled and before the reply is returned, the `postinvoke` operation is called on the Servant Locator. The QoS Servant Locator re-implements the `preinvoke` operation shown in Figure 6-17.

Figure 6-17 The preinvoke operation

```
Servant preinvoke(in ObjectId oid, in POA adapter,
                  in CORBA::Identifier operation,
             out cookie the_cookie)
   raises (ForwardRequest);
```

During negotiation, the request `negotiate_qos(o)` issued by the QoSRepository at the client side arrives at the server side to the QOA. As mentioned before, the QOA uses the QoS Servant Locator that implements the standard Servant Locator interface.

To handle the `negotiate_qos` request the QOA calls the `preinvoke` operation of its QoS Servant Locator object. This returns the QoS Negotiator servant, if the `operation` argument is "`negotiate_qos`", otherwise it returns the servant that has been registered with the ObjectId.

6.4 Design decisions

This section motivates some of the design decisions that where made during the design of QPS for CORBA.

6.4.1 Servers, servants and objects in CORBA

CORBA introduces the notion of a *servant*. A servant is a particular implementation of a CORBA object that performs the work on behalf of a CORBA object. It can be a function written in C or FORTRAN or a C++ or Java object. Using the Portable Object Adapter (POA), it is possible to

associate servants with a CORBA object within one server dynamically. This may result in scenarios, in which different servants handle consecutive requests sent to the same CORBA object, but this is completely transparent to clients. This technique is used for load balancing, or to keep objects in persistent storage and load them into memory on demand. However, as explained before, we have decided to have one servant implementation per object, in other words, it is required that requests to published object references are routed to the same servant, in the same server. This is true for example, in the Internet InterORB Protocol (IIOP) where the server location is fixed and is part of the object reference. This also implies that QPS does not support location forwarding.

6.4.2 CORBA implementation of the client/server bindings

CORBA recognises client and server roles, but only in the context of a request/reply. The client is the entity originating the request and the server is the entity in which the object resides to which the request is sent. Clients, however, have no identity in CORBA.

When implementing the binding support, we wanted to have a binding repository where tuples of the form <client id,server objects> would be stored. This is not possible at this moment in standard CORBA, since there is no client identifier.

An alternative way for implementing bindings has been sought, based on the fact that there is one ORB instance per client. Associated to the ORB instance, we can store the references to the server objects for each binding. We call this solution the client-side implementation of bindings. In a recent real-time CORBA specification [Ch96], there is a facility to register a client/server-object binding explicitly. Similarly to our approach, this registration is also done on the client-side.

A binding object represents a binding between a client and a server object. A binding object associates a client and a (server) object reference. A binding is created implicitly when a client obtains the object reference. A binding is removed when the client or the CORBA server terminates. The binding remains valid for a series of requests of the client to the object. Similarly, an agreed quality level remains valid for a series of requests, although a client may renegotiate the agreed quality level at any time during the existence of a binding. A binding does not imply an uninterrupted connection between client and server. A new connection can be established as the result of a renegotiation.

6.4.3 Dynamic Invocation Interface call

During negotiation, the `negotiate_qos` DII request is sent by the QoSRepository using object references as targets that do not define the

`negotiate_qos` operation in their interface. This is not compatible with the CORBA 2.3 specification, although extension of CORBA::Object in this way seems to be a silently accepted technique supported by most ORB implementations. Theoretically, the ORB implementation may perform run-time type checking on the parameters of the DII request [PI99]. This may be necessary for correct marshalling of certain types, such as object references. The problem is that if an ORB would indeed perform type checking it would reject to process the `negotiate_qos` request, since the operation is unknown for the object it is targeted at.

There are at least two different ways to solve this problem:

1. An object for which a certain quality level can be negotiated must support the `negotiate_qos` operation, for example, by inheriting from a special QoS interface that declares this operation. The advantage of this approach is that is makes QoS support independent of present and future CORBA specifications. The disadvantage is that adding a QoS support later to objects requires a change of interfaces.
2. The CORBA::Object interface from which all CORBA objects inherit should support the `negotiate_qos` operation. This effectively positions the negotiation request besides implicit object reference operations such as `non_existent` and `is_a`, and requires extension to the CORBA::Object interface and explicit support for it in the ORB and in GIOP. The advantage of this approach is that no interface change is necessary to build QoS support into existing objects. The disadvantage is that modification of the CORBA specification and the ORB implementation are necessary.

In the current QPS implementation, we have assumed that no type checking by the ORB for simple types such as `string` is performed, which is the case for ORBacus [ORBacus]. Hence, we do not add the operation `negotiate_qos` to the CORBA::Object interface. Thus the request is sent as if the object supports it. As a consequence, the operation name we have inserted into the request is `negotiate_qos` and not `_negotiate_qos` as it would be the case for implicit object reference operations.

This choice allows the separation of QoS support on the client-side from QoS support on the server side. Whether a CORBA object is instantiated on a QoS aware ORB should not be reflected in its (functional) interface.

6.5 QIOP

QIOP is an inter-ORB protocol that conveys standard inter-ORB messages via dedicated channels offering guaranteed (system-level) QoS for messages sent through these channels. QIOP offers an ORB all the facilities needed to convey General Inter-ORB Protocol (GIOP) messages, in a similar way as IIOP does. The IIOP protocol specifies how GIOP messages are transported over TCP/IP connections. However, the IIOP protocol cannot provide guarantees on throughput and/or delay for message delivery. With QIOP such guarantees can be provided. Resource Reservation Protocol (RSVP) control messages are used to query available resources for reservation.

QIOP builds on the acceptor/connector pattern [Sc97]. It uses the Open Communication Interface (OCI) [FHKV99] to register and interact with the ORB. Figure 6-18 shows how a QIOP transport connection is established. The QosRepository uses the QIOP ConFactory to create a Connector. The Connector establishes a TCP/IP connection with the server side and creates QIOP transport objects. These transport objects create two RSVP sessions, one for network traffic from the client to the server side and one for network traffic in the opposite direction. This is necessary because RSVP can only reserve network resources for a unidirectional flow.

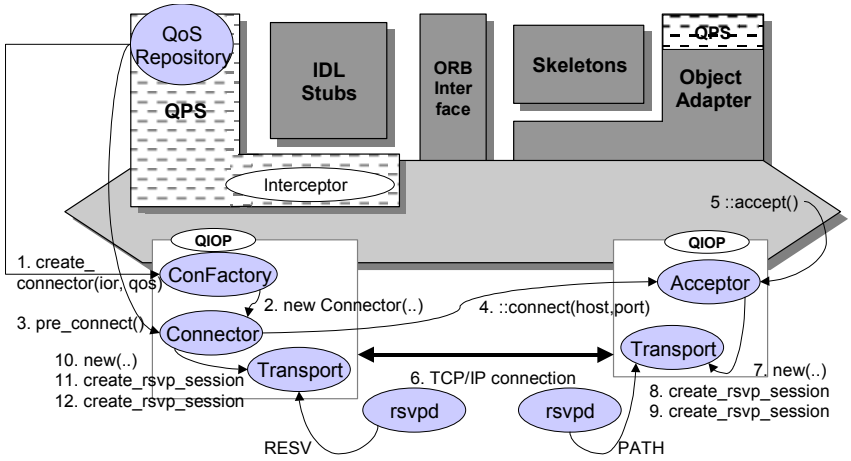


Figure 6-18 QIOP interactions

The Transport objects create RSVP reservations for both RSVP sessions according to Q_{agreed}.

6.6 QIOP experiment

To demonstrate the benefits of QIOP over IIOP, we have conducted some experiments. The experiment system consists of three PCs running Linux. One PC serves as a host for client objects, another PC serves as a host for a server object. The third PC is configured as a router with two Ethernet interfaces that connect to the client and server hosts. All PCs run the KOM-RSVP implementation [KSS01] and the client and the server hosts run an ORB with QPS and QIOP extensions.

In the experiment, two client objects are running on the client host: one with a QoS requirement $Q_{required}$ and one without a QoS requirement. Both clients connect to a single server object, i.e., they use the same object reference. As a result, the client without QoS requirements communicates using IIOP and the other client communicates using QIOP. To show the behaviour of QIOP in a saturated network a heavy data stream, with occasional bursts has been injected into the network. Figure 6-19 shows the variation of the response times of the two clients in time.

Figure 6-19 Response times in a saturated network

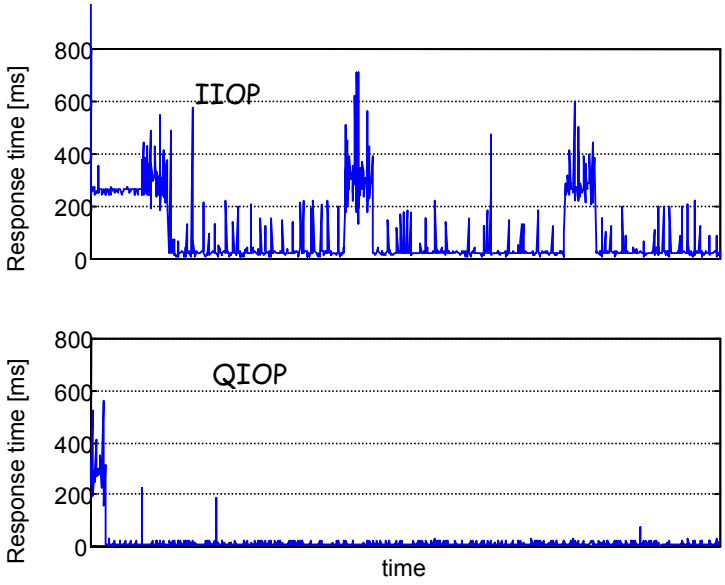


Figure 6-19 shows that the response times for messages carried over QIOP are not sensitive to heavy traffic bursts on the network (they stay below 100 ms), whereas messages carried over IIOP can are strongly affected by the occasional bursts. This demonstrates that applications with more stringent

requirements on the response time of remote object invocations can benefit from QPS with a QIOP plugin.

In the future, more QoS mechanism plugins could be implemented, similar the QIOP plugin for RSVP, in order to increase the number of QoS mechanisms we can use in QPS.

6.7 Conclusions

Next generation middleware must meet the challenge of evolutionary changes and run-time changes in a heterogeneous distributed computing environment, in order to provide distributed objects support for QoS. This can be achieved by meeting the following requirements: *1)* support of application-level QoS concepts *2)* flexible and extensible software design and *3)* adaptable QoS support.

The QoS Provisioning Service (QPS) enables control plane functions to be added to off-the-shelf object middleware, for controlling the QoS of individual client-server bindings. It has been developed according to a five phase life cycle model to establish and control a QoS agreement between a client and a server. QPS has been designed to meet the above listed requirements on QoS aware middleware.

To support the binding life cycle two frameworks have been developed: a QoS *negotiation* framework and a *QoS-control* framework.

The QoS negotiation framework establishes a QoS agreement, based on the required and offered QoS of the client and server object, respectively. The negotiation framework is based on the whiteboard negotiation approach and allows for specific QoS negotiators to be inserted into the QoS negotiation process.

The QoS control system observes and, if necessary, manipulates the state of the controlled system, i.e. the middleware platform and DRP, to maintain a QoS agreement. The design of the QoS controller is an architectural framework that is based on models from control theory. This ensures stability with respect to evolving requirements, and applicability to a wide range of controlling techniques.

The QoS-control design has been discussed in more detail by examining a number of technical issues that must be addressed when realizing the proposed design. For each of these issues, we discussed requirements and corresponding solutions or solution approaches.

The prototype implementation of QPS for CORBA measures the QoS by using Portable Interceptors during system operation, and controls QoS through a feedback loop. Control actions are taken by configuring the system, but only at the transport level, by means of pluggable protocols. The specific functions and mechanisms for establishment and maintenance

of a QoS agreement at the transport level are defined in a protocol called QIOP. QIOP is a CORBA communication module that uses RSVP for reserving network resources. We demonstrated its performance benefits compared to communication over IIOP.

We have identified several topics for interesting future work. These topics address the further development and prototyping of our control design, as well as exploring controlling strategies and algorithms that could not be considered so far. In addition, we would like to profit from results of related work:

- One of the characteristics of our proposal is that the design is largely independent of a specific middleware platform. The QoS controller independent from the middleware (and applications) and may interact with these through a number of *probes* (a generic term for interfaces that abstracts from specific implementations). Conceptually, this is a reflective model; our QoS controller observes and manipulates the middleware at a meta-level. Other proposals for reflective middleware have been made, e.g. [BCRP98], and we would like to see how QPS integrates with reflective middleware.
- A middleware framework for QoS adaptation has been described in [LiNa99]. Both a task control model and a fuzzy control model have been used in this framework to formalize and calculate the control actions necessary to keep the application QoS between bounds. This framework shares many design concerns with our framework, although it has been targeted to the control of applications. Fuzzy logic seems also a promising technique for QPS to determine control actions.
- OMG has developed Real-time CORBA standards in the scope of the CORBA 3.0 standard [ScKu00]. These facilities allow one to manipulate some middleware characteristics that influence the QoS, such as, e.g., the properties of protocols underlying the ORB and the threading and priority polices applied to the handling of requests by server objects. These facilities are defined in terms of interfaces that have to be implemented in the middleware platform, generalising in this way the control capabilities of the platform. Further investigation is needed to realise an ORB implementation independent implementation of QPS.

Chapter 7

Conclusions

This chapter presents the main conclusions of this thesis and identifies directions for further research.

7.1 General conclusions

This thesis focuses on the simplification of the design, development and deployment of telematics services. We assume that a designer of such a service benefits from the use of object technology and middleware technology. It has been shown that object middleware is an important infrastructure that supports the design, development and deployment of telematics services. Middleware hides the functions and mechanisms needed to overcome the problems that are caused by the distribution of resources.

Our main premise is that *QoS support must be an intrinsic part of an object middleware platform*. Such a middleware is called a QoS aware object middleware, since it facilitates the realisation of the QoS concerns of a telematics service. A QoS aware object middleware hides the functions and mechanisms needed to realise QoS requirements.

A QoS aware object middleware is distinguished from a non-QoS aware object middleware by establishing and maintaining the QoS concerns that have been defined during the design of a telematics service. Non-QoS aware object middleware offers a best-effort QoS to telematics services.

In case a telematics service is offered under a service level agreement with strict QoS constraints, the designer of that service has to design a QoS critical application. Object middleware that only supports a best-effort QoS constitutes an obstacle to the realisation of QoS critical applications. Since middleware hides the functions and mechanisms needed to overcome problems of distribution, application components are generally shielded from direct access to communication and computing functions. However,

application components need this access to control these resources to establish and maintain the QoS concerns of the telematics service.

A QoS aware object middleware also shields application components from access to the computing and communication resources, but provides the means for application components to inform the middleware about QoS requirements. The QoS aware middleware aims to configure the computing and communication resources in accordance with the QoS requirements.

We have shown that a QoS aware middleware has to control changes in the resources that impact the QoS of a telematics service. Therefore, we propose that QoS aware object middleware is adaptable to the run-time and evolutionary changes that impact the QoS delivered by an open distributed system.

The main objectives of this thesis are summarised in the following three points:

1. Construct a reference model of object middleware and clearly separate the qualitative aspects of the object middleware infrastructure from the QoS concerns of a telematics service;
2. Advance object middleware technology through the addition of facilities that can control the qualitative aspects of the objects deployed on the middleware;
3. Validate our objective to make middleware QoS aware by developing an infrastructure service that can leverage existing mechanisms for QoS establishment and control to the middleware level.

We evaluate the results of this thesis against these objectives in the sequel.

7.2 Modelling QoS aware middleware

The first objective of this thesis has been achieved by the analyses of open distributed systems, identification of the role of object middleware in these systems and the development of models for QoS aware middleware.

In Chapter 2, we have illustrated that the resources of an open distributed system are not manufactured or owned by a single organisation. As a result, a distributed system often crosses multiple technological and organisational boundaries. To construct an open distributed system from parts manufactured by various organisations, rules that guarantee the interoperability of these parts must be established.

Characteristics of an open distributed system such as remoteness, concurrency, lack of global state, partial failures, asynchrony, heterogeneity, autonomy, evolution and mobility complicate the design of such a system. A

designer of a telematics service faces a complex task when all these characteristics have to be taken into account. Modelling techniques and design principles such as abstraction and refinement provide a designer with the means to manage this complexity.

We have applied the notion of *object* to model the parts of an open distributed system. Designers create object models consisting of objects and their relations to capture the conceptual parts or concrete software parts of an open distributed system.

The concepts used to develop an object model are captured in a meta-model. A viewpoint is defined using a selected set of concepts that constitute a meta-model for that viewpoint. We have defined a view on a distributed system as an instance of the associated viewpoint meta-model.

To model the aspects of an open distributed system that are of concern of this thesis we have defined the computational, engineering and deployment viewpoint. The meta-models that define each of these viewpoints and the correspondence relation between concepts in these meta-models constitute a modelling concept space for open distributed systems.

Three designer roles have been identified. The *application designer* is responsible for the design of application objects and is concerned with developing computational designs. The *infrastructure designer* is responsible for the design of the supporting infrastructure for distributed applications and is mainly concerned with the development of engineering designs but may also employ computational concepts to express the design of the infrastructure. The *deployment designer* constructs units of deployment denoted as components. An application component is constructed from an assembly of computational classes; an infrastructure component is constructed from the classes defined by an infrastructure designer. The deployment designer ensures that application components and infrastructure components are compatible in the sense that an infrastructure component can be used as the run-time environment for application components.

QoS aware middleware for distributed objects enables the establishment of bindings between two computational objects that are subject to QoS agreements. A QoS agreement (Q_{agreed}) is the result of a negotiation process between the offered QoS ($Q_{offered}$) of a server object, the required QoS ($Q_{required}$) of a client object and the resources available to the object middleware.

In Chapter 3, we have shown that in the area of software engineering technologies and software design, the Meta-Object Facility (MOF) is an important standard that suits our need to construct multiple meta-models that designers use to develop telematics services.

In Chapter 4, the common concerns of contemporary and early middleware platforms have been identified. The analysis presented there has resulted in an object middleware reference model. Our reference model consists of object communication middleware, general purpose services and the component execution environment.

Our reference model also defines the interoperability and portability reference points to which manufacturers of object middleware should adhere in order to produce interoperable and portable products.

In Chapter 5, we have introduced two distribution transparencies: the QoS *enforcement* and the QoS *control* transparency. Functions that provide the establishment of a QoS agreement support the QoS enforcement transparency. Functions that provide the establishment and maintenance of a QoS agreement support the QoS control transparency.

Through these QoS transparencies we have enabled the separation of the qualitative aspects of the object middleware infrastructure from the QoS concerns of a telematics service

7.3 Advancing object middleware

Our second objective is to advance object middleware through the addition of functions and mechanisms that establish and maintain QoS agreements. This thesis contributes to this objective in several ways.

In Chapter 3, we have presented the area of object middleware architectures, QoS architectures, network technologies and software engineering technologies. Organisations that impact these are IETF, W3C, ISO-ITU, OMG and SUN JCP. We have shown that an infrastructure designer must consider the ongoing developments in these organisations and consortia when designing the supporting infrastructure for computational objects. In fact, we have identified middleware as a point of convergence where several standards, architectures and technologies must be aligned in order to introduce QoS awareness into object middleware platforms. Figure 7-1 shows the forces that affect the introduction of QoS awareness into object middleware.

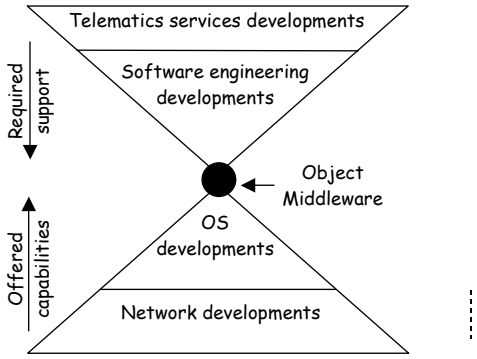


Figure 7-1 Middleware as a point of convergence of technological developments

We have described the developments in the area of network technologies that contribute to introduction of QoS support in packet networks. Two alternative approaches, i.e., IntServ and DiffServ, for controlling the QoS of a packet network have been identified.

From these observations we have concluded that new protocols and mechanisms for the control of QoS in packet networks will emerge. Our goal is to shield an application designer from these developments. Consequently, we advocate a service driven approach to the design of a QoS aware middleware. An infrastructure designer should shield the application designer from the protocols, interfaces and mechanisms used by the network to control the QoS.

Chapter 3 contributes to our objective to advance object middleware by identifying the main organisations and standards in the area of object middleware and network technology. The interdependencies of various object middleware standards have been identified. In addition, various QoS architectures have been reviewed. As a result, our proposals to add facilities for QoS establishment and control are aligned with existing architectures, products and standards used to realise open distributed systems.

In Chapter 5, we have defined the concepts to model QoS aspects of an open distributed system.

We provide the application designer with the modelling concepts to express the QoS aspects of a computational specification. We have extended the modelling concept space with meta-model concepts that we use to develop *QoS contracts*. A client computational object is associated with a QoS contract that states the clients' required QoS. A server computational object is associated with a QoS contract that states the servers' offered QoS.

We provide the infrastructure designer with the design concepts that express QoS aspects of a QoS aware object middleware. Therefore, we have extended the modelling concept space with meta-model concepts that we use to develop *QoS contract types*. A QoS contract type defines a class of

potential QoS contracts that an application designer may use to design QoS contracts. A QoS contract type is the means by which an infrastructure designer communicates the potential QoS capabilities of a QoS aware middleware.

QoS aware middleware supports one or more QoS contract types. To support a particular QoS contract type the infrastructure designer needs to design functions and mechanisms needed to establish and maintain a QoS agreement. We provide the infrastructure designer with design principles such as the user-provider principle, the separation principle and the integration principle.

7.4 Validation

Our third objective is to validate our approach to the construction of QoS aware object middleware. Chapter 6 mainly contributes to this objective. This chapter presents the design of a general purpose service that provides QoS support. This service is called the QoS Provisioning Service (QPS).

The design concepts and principles, as well as the service driven approach to QoS provisioning have been evaluated by the design and implementation of a prototype QPS.

We have designed QPS to manage evolutionary changes of the QoS functions and mechanisms offered by an open distributed by shielding the use of these functions from the application designer. The infrastructure designer creates functions and mechanisms at the middleware layer that control the QoS offered by the network and computing nodes. The QoS capabilities that a QPS enhanced object middleware supports are communicated to the application designer as QoS contract types. A QoS contract type only reveals *what* and not *how* QoS capabilities are supported.

A framework for QoS negotiation and QoS control has been discussed. We have designed QPS to manage the run-time changes through run-time establishment of QoS agreements and to maintain QoS agreements by means of a QoS control loop.

QIOP is an example of a protocol that has been implemented according to frameworks prescribed for the QPS. QIOP shields the developer of a CORBA application from the means to control the network QoS. It enables an application developer to define performance QoS contracts that specify requirements for the number of invocations per second and end-to-end delay of an invocation.

7.5 Directions for further research

To further understand the models and concepts discussed and applied in this thesis, various topics could be further investigated. This section discusses some of these topics and identifies how study of these topics can further contribute to the goals of this thesis.

Additional QoS mechanisms

We have claimed that our current QPS design is adaptable to evolutionary changes. If new mechanisms for the control of QoS aspects of a packet network become available, it should be possible and relatively easy to include these mechanisms into the QPS. This could be studied, for example, by modification of QIOP and replacing the RSVP network reservation mechanisms with the mechanisms that Boomerang provides. Such a replacement should not require changes to application components, since these application components can still construct performance contracts according to the performance contract type offered by the QPS.

Successful replacement of RSVP with Boomerang would demonstrate the evolutionary adaptability of the QPS.

QoS contract types

Our QoS meta-model supports many contract types. Our QIOP implementation shows how a performance contract type can be implemented. Further study could concentrate on the development of other contract types, such as contract types that support availability or safety.

A study on availability contract types could benefit from ongoing research in the area of replication, load balancing and load distribution.

A study on safety contract types could benefit from security mechanisms for packet networks such as IPsec and public key infrastructures. An early start of this research has already been made [Ko01].

Further research to construct and support additional contract types, supports our objective to separate the qualitative aspects of the object middleware infrastructure from the QoS concerns of a telematics service by means of QoS contract types.

Aspect oriented software engineering technologies

An infrastructure designer could benefit from novel software engineering technologies. Aspect oriented software engineering techniques provide a means to compose software by weaving aspects together [KLM+97]. Middleware functions and mechanisms could be defined in an aspect oriented manner and woven together with QoS functions and mechanisms to realise a QoS aware object middleware.

If aspect oriented software techniques simplify the task of an infrastructure designer, these techniques also contribute to our goal to simplify the design and development of telematics services.

Integration with the UML

The UML is a broadly used modelling language for specifying and constructing software artefacts. In case the UML is used to develop computational specifications of telematics services, with the assumption that there is a supporting infrastructure that provides distribution transparencies to the computational objects, we advice to integrate our QoS meta-model with the UML meta-model.

Such integration would enable the specification of QoS contracts using UML artefacts. Tools could be developed that enable an application designer to manipulate the specifications of telematics service, including the QoS aspects of that service, using a UML notation. Such tools further simplify the design of telematics services and thus contribute to our goals.

Refinement of the deployment viewpoint

We have identified the deployment of application components and infrastructure components as an important concern for the realisation of a telematics service. In Chapter 2 we have defined concepts for the deployment viewpoint, such as component, deployment descriptor, node and run-time environment. Further refinement of these concepts is needed. This refinement should be directed by the deployment concepts found in contemporary object middleware specifications, such as J2EE and the Microsoft .NET specifications.

Refinement of the deployment meta-model and mapping of the meta-model concepts to the implementation concept space of object middleware platforms enables further automation of the deployment of telematics services. OMG already solicits proposals in this direction [Depl02]

Application to emerging infrastructures

A recent development for sharing computing resources over communication networks the grid computing [LFG+00, FKN02]. The grid is an emerging infrastructure for distributed computing, to which the design of QPS could be applied. As a result, QoS contracts can be negotiated between various application components and providers of computing resources can offer differentiated QoS to their users.

Successful application of the QPS design in the context of grid computing would further support our premise that QoS should be an intrinsic part of the supporting infrastructure for distributed applications. `

Appendix A

MODL specification of the QoS meta-model

This appendix presents the MODL specification of the QoS meta-model described in Chapter 5. MODL is the specification language that the dMOF toolset of DSTC uses to specify a meta-model.

```
//
// QoSContract.modl
// MODL definitions for a QoSContract model

package qos_contract_repository {

  // ********************************************
  // * Container and contained definitions
  // ********************************************
  //
  abstract class contained{
        attribute string name;
        // make this class aware that it is
        // the end of an association
        reference defined_in to
           the_container of contains;
  };

  abstract class container : contained {
        // make this class aware that it is the end
        // of an association
        reference contents to
           the_contained_element of contains;
   };

  // ********************************************
  // * Contains definition
  // ********************************************
  //
  // The 'contains' association describes the
  // relationship between a container and its
  // contained elements.
  association contains {
```

```
  composite end bag [0..1]
      of container the_container;
  end ordered set [1..*] of
      contained the_contained_element;
};

// ***********************************************
// * QoSContractType definition
// ***********************************************
//
class qos_contract_type : container {
      readonly attribute short major_version;
      readonly attribute short minor_version;
};

// ***********************************************
// * Dimension definition
// ***********************************************
//
enum direction_kind {
        dk_increasing, dk_decreasing
        };

class dimension : contained {
      attribute direction_kind direction;
      attribute TypeCode dimension_type;
      attribute string unitDescription;
      attribute boolean allowMultiConstraint;
};

// ***********************************************
// * QoSContract definition
// ***********************************************
//
class qos_contract : container {
      readonly attribute qos_contract_type
                           the_contract_type;
};

// ***********************************************
// * DimensionMultiConstraint definition
// ***********************************************
//
class dimension_multi_constraint : container {
      // no attributes: they are all inherited...
};

// ***********************************************
// * DimensionSingleConstraint definition
// ***********************************************
//
enum constraining_operator_kind {
   co_eq, co_lt, co_gt, co_ge, co_le
   };

class dimension_single_constraint : contained {
      attribute any parameter;
```

```
        attribute constraining_operator_kind
                                    operator;
    };

    // ********************************************
    // * QoSDimensionStatisticalConstraint definition
    // ********************************************
    //
    enum statistical_operator_kind {
     so_percentile, so_frequency, so_mean, so_variance
     };

    class dimension_statistical_constraint :
                dimension_single_constraint {
        attribute statistical_operator_kind
                      statistical_operator;
        attribute any statistical_parameter;
    };

};
```

Samenvatting

Eén van de meest spectaculaire ontwikkelingen op het gebied van Telecommunicatie is het Internet. Internet heeft grote invloed gehad op de samenleving. Onze economie en de manier waarop we zaken doen is er sterk afhankelijk van geworden. Met de opkomst van het Internet hebben ook telecommunicatiebedrijven enorme veranderingen ondergaan. Het zijn ondernemingen die hun diensten leveren op een competitieve markt. Dienstverleners kunnen zich onderscheiden op hun markt door de kwaliteit waarmee zij hun diensten aanbieden. Deze dienstkwaliteit (eng: Quality of Service genoemd) is het onderwerp van studie in dit proefschrift. Het onderzoek heeft geleid tot concepten en technologieën die ontwerpers van diensten in staat stelt de kwaliteit van een dienst eenvoudiger te realiseren.

In het algemeen koppelt men de kwaliteit van telematicadiensten één op één met de kwaliteit van de onderliggende communicatienetwerken. Daarbij denken we aan eigenschappen zoals de bandbreedte, de vertraging en variatie op die vertraging. Dat is echter niet, waar dit proefschrift zich op richt. Hier ligt de focus op de generieke software componenten voor de levering van diensten. Het gaat dus om software-infrastructuren die generieke functies leveren waar specifieke applicatiecomponenten gebruik van kunnen maken. De kern van dit proefschrift is de generieke functionaliteit van het dienstenplatform waar ontwerpers van diensten een beroep op kunnen doen om gebruikers van deze diensten een gewenst kwaliteitsniveau te kunnen leveren.

In dit proefschrift beschouwen we de klassen van dienstenplatformen die zijn gebaseerd op object middleware. Middleware is de laag die zich bevindt tussen de applicatiecomponenten en de communicatiesystemen (datacommunicatie en computers). De huidige dienstenplatformen bieden onvoldoende ondersteuning voor de kwaliteitsaspecten van een telematica dienst, zoals snelheid, betrouwbaarheid en veiligheid. Het onderzoek richt zich op uitbreiding van de dienstenplatformen met functies voor

kwaliteitsondersteuning. Een ontwerper van telematicadiensten kan hiervan gebruik maken en hoeft zich niet te verdiepen in de functies en mechanismen die nodig zijn om de gewenste kwaliteitsprestaties van een telematicadienst te realiseren.

De vraag welke mechanismen dienstenplatformen moeten bieden kan niet los worden gezien van de vraag hoe telematicadiensten moeten worden ontworpen. Platformen stellen ontwerpers van diensten in staat te abstraheren van allerlei details. Hierbij gaat het met name om het verbergen van de aspecten die te maken hebben met distributie van applicatie-componenten (eng: distribution transparency). Uitgangspunt is dat die functionaliteit waarvan de applicatieontwerper abstraheert (vrijwel) automatisch kan worden geïmplementeerd door functies die het dienstenplatform levert. Dit proefschrift presenteert modellen en prototypes en laat daarmee zien dat dit principe ook toepasbaar is voor het ontwerpen en realiseren van de kwaliteitsaspecten van telematicadiensten.

Hoofdstuk 2 van dit proefschrift geeft concepten voor het ontwerpen van open gedistribueerde systemen. Dit zijn systemen die zijn opgebouwd uit gestandaardiseerde componenten, die in de regel door diverse fabrikanten worden geproduceerd. In het algemeen maken open gedistribueerde systemen deel uit van grootschalige infrastructuren, die door diverse organisaties worden beheerd. Open gedistribueerd systemen overschrijden daarmee zowel technologische als organisatorische grenzen.

Om dergelijke infrastructuren mogelijk te maken, zijn er afspraken nodig over de regels volgens welke de componenten met elkaar samenwerken. Ook andere eigenschappen zoals fysieke afstand tussen onderdelen, parallellisme, het ontbreken van een globale toestand, gebrek aan autonomie, invloed van technologische ontwikkelingen en mobiliteit zorgen ervoor dat het ontwikkelen van een open gedistribueerd systeem een complexe bezigheid is.

Om de complexiteit bij de ontwikkeling van open gedistribueerde systemen te beheersen, maken ontwerpers gebruik van modellering-technieken zoals abstractie en verfijning. In ons onderzoek gebruiken we object modellen. Een metamodel definieert de concepten die de ontwerper hanteert voor het ontwikkelen van de object modellen.

Een andere methode om de complexiteit te beheersen is het gebruik van *gezichtspunten* (eng: viewpoints). Een gezichtspunt, of perspectief, bestaat uit een selecte verzameling van modelleringconcepten. Deze concepten vormen het metamodel dat een gezichtspunt definieert. Een model volgens een gezichtspunt is een instantie van het daaraan verbonden metamodel. Een ontwerper hanteert verschillende gezichtspunten om de diverse aspecten van het ontwerp van een gedistribueerd systeem tot uitdrukking te brengen. De correspondentierelaties tussen concepten in de metamodellen

definiëren de relaties tussen gezichtspunten. Het onderzoek heeft geresulteerd in drie gezichtspunten, te weten computational, engineering en deployment gezichtspunt. De metamodellen voor elk van deze gezichtspunten en de correspondentie relaties tussen de concepten in deze modellen vormen zo een conceptruimte voor het modelleren van open gedistribueerde systemen.

Hoofdstuk 3 gaat in op de ontwikkelingen van referentiemodellen, standaarden en technologieën voor open gedistribueerde systemen. Deze studie borgt dat de resultaten van het onderzoek naar platformen met kwaliteitsondersteuning hierbij aansluiten. Deze studie vormt tevens de basis voor het referentiemodel voor object middleware dat tijdens het onderzoek is ontwikkeld. Object middleware met kwaliteitsondersteuning blijkt een convergentiepunt te zijn van verschillende standaarden, architecturen en technologieën.

Hoofdstuk 4 presenteert het referentiemodel voor object middleware. Het referentiemodel modelleert de functionaliteit van de bestaande dienstenplatformen, waarbij is geabstraheerd van de gebruikt implementatietechnologieën. Het referentiemodel tilt het onderzoek naar het gewenste conceptuele niveau en garandeert dat de resultaten voor diverse technologieën kunnen worden toegepast.

Het referentiemodel is een belangrijk deelresultaat van dit onderzoek. Het definieert onder meer de referentiepunten voor portabiliteit en interoperabiliteit waaraan object middleware moet voldoen om als portable en interoperabel aangemerkt te kunnen worden.

Hoofdstuk 5 beschrijft de concepten die van belang zijn om de kwaliteitsaspecten van een telematicadienst te modelleren. De ontwerpers van zowel applicaties als dienstenplatformen gebruiken deze modellen. Ten behoeve van de applicatieontwerper is de modelleringsconceptruimte uitgebreid met concepten voor het beschrijven van kwaliteitscontracten. De applicatieontwerper specificeert de vereiste kwaliteit van een client object door aan dit object een Q_{agreed} contract te verbinden. De kwaliteit die een server object levert wordt gespecificeerd door aan dit object een $Q_{offered}$ contract te verbinden.

Ten behoeve van de infrastructuur ontwerper is de modelleringsconceptruimte uitgebreid met concepten voor het beschrijven van kwaliteitscontract *types*. Een contract type beschrijft een klasse van potentiële kwaliteitscontracten die applicatieontwerpers gebruiken voor het specificeren van kwaliteitscontracten.

Een kwaliteitscontract type is het middel waarmee een infrastructuur ontwerper de applicatie ontwerper duidelijk maakt welke potentiële kwaliteitseigenschappen de middleware ondersteunt.

Hoofdstuk 6 beschrijft het ontwerp van een systeem dat de kwaliteitsondersteuning vanuit het dienstenplatform biedt. Dit systeem is het hoofdresultaat van dit promotieonderzoek en heet de QoS Provisioning Service (QPS). QPS is een generieke dienst die een onderdeel is van object gebaseerde dienstenplatformen. Dankzij QPS hoeft de applicatieontwerper geen kennis te hebben van de functies en mechanismen die de kwaliteitsprestaties van applicatie objecten verzorgen. De infrastructuurontwerper zorgt voor de aansturing van de kwaliteitsprestaties die het netwerk en computer systemen leveren. Door middel van een contract type wordt aangegeven *welke* kwaliteits prestaties ondersteund worden en niet *hoe* dat gerealiseerd is.

Voor het ontwerp van QPS is gebruik gemaakt van een raamwerk voor de onderhandeling en aansturing van kwaliteitsprestaties. QPS onderhandelt over kwaliteitsafspraken en onderhoudt deze afspraken door middel van een controle lus.

Als voorbeeld hoe QPS de kwaliteitsprestaties van een netwerk kan aansturen is de QIOP module ontwikkeld. QIOP schermt de applicatieontwerper af van de manier waarop de kwaliteitsprestaties van het netwerk worden bestuurd. Met QIOP kan een clientobject afspraken maken met een serverobject over het aantal aanroepen per seconde en over de eind-eind vertragingstijd van een aanroep.

De vraag welke mechanismen dienstenplatformen moeten bieden kan niet los worden gezien van de vraag hoe telematicadiensten moeten worden ontworpen. Platformen stellen ontwerpers van diensten in staat te abstraheren van allerlei details. Hierbij gaat het met name om de aspecten die te maken hebben met distributie van applicatiecomponenten (eng: distribution transparency). Uitgangspunt is dat die functionaliteit waarvan de applicatieontwerper abstraheert (vrijwel) automatisch kan worden geïmplementeerd door functies die het dienstenplatform levert.
Het doel van dit promotieonderzoek is dienstenplatformen te voorzien functies die aansluiten bij concepten die applicatieontwerpers kunnen gebruiken voor telematicadiensten. Dit resultaat is bereikt in de vorm van de volgende drie deelresultaten:

– Er is een referentiemodel voor object middleware ontwikkeld dat een duidelijk onderscheid maakt tussen de kwaliteitsaspecten van het dienstenplatformen en die van de telematicadienst.
– Er zijn concepten ontwikkeld voor functies en mechanismen die voor een brede klasse van dienstenplatformen kunnen worden toegepast. De

ontwikkelde functionaliteit sluit aan bij concepten die applicatieontwerpers hanteren bij het ontwerp van telematicadiensten;
- Er is een werkend prototype dienstenplatform ontwikkeld dat kwaliteitsondersteuning biedt. Het platform is gevalideerd in een proefomgeving waar de kwaliteitsverschillen ten opzichte van een dienstenplatform zonder kwaliteitsondersteuning zijn aangetoond.

References

URLs are provided, where available, for easy access to online versions of works cited. For the full text of some of the cited works, you may need an account for the ACM Digital Library or the IEEE Computer Society Digital Library.

[AA97] F.A. Aagesen, QoS Frameworks for open distributed processing systems Telektronik, vol. 1.97, no. magazine on Quality of Service in Telecommunications, pp. 26-41, 1997.

[AAS02] T.F. Abdelzaher, E.M. Atkins, K.G. Shin, “QoS negotiation in Real-time sytems and its application to automated Flight Control”, June 2002.

[AbSh95] T.F. Abdelzaher and K.G. Shin, “Optimal combined task and message scheduling in distributed real-time system”, in IEEE Real-time Systems Symposium, Pisa, Italy, December 1995.

[ACH98] C. Aurrecoechea, A.T. Campbell and L. Hauw, "A Survey of QoS Architectures," ACM/Springer Verlag *Multimedia Systems Journal , Special Issue on QoS Architecture*, Vol. 6 No. 3, pg. 138-151, May 1998.

[BCRP98] G. Blair, C. Coulson, P. Robin, M. Papathomas, “An architecture for next generation middleware”, In: N. Davis, K. Raymond, J. Seitz (eds.). Middleware. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 191-206. Springer-Verlag, London, 1998.

[BeGe97] C. Becker and K. Geihs, "MAQS: management for adaptive QoS-enabled services", Proceedings of the IEEE Workshop on Middleware for Distributed Real-Time Systems ands ervices, 1997.

[BHK+96] E.M.M.A van den Broek, W.M. van Hulten, A. Koetluk-Florescu, L.J.M., Nieuwenhuis, E.M.Peeters, “Distributed objects in Telecommunications: a Managers Guide”, deliverable 3 of EURESCOM project P517, 1996.

[BHK00] M. Born, A. van Halteren, O. Kath, "Modeling and Runtime Support for Quality of Service in Distributed Component Platforms", work-in-progress paper, DSOM 2000, Austin, Texas, USA, December 2000.

[BHP+00] L. Bergmans, A. van Halteren, L. Ferreira Pires, M. van Sinderen and M. Aksit, “A QoS-Control Architecture for Object Middleware”, IDMS 2000, Enschede, The Netherlands, October 2000.

[BiNe84] A.D. Birrell and B.J. Nelson, “Implementing Remote Procedure Calls”, ACM Transactions on Computer Systems 2(1), 39-59, February 1984.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, “Making components contract aware”, *IEEE Computer*, 13(7), July 1999.

[BlSt97] Blair, G. and Stefani, J.B., “Open Distributed Processing and Multimedia", Addison-Wesley, 1997.

[Bo98] Jon Bosak, et al., *W3C XML Specification DTD*, http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.

[BoGa88] A. H. Bond and L. Gasser, “An Analysis of Problems and Research in DAI”, pages 3-35. Morgan Kaufmann Publishers Inc., Los Angeles, CA, 1988.

[BoKa02] M. Born, O. Kath, “COrE – Komponentorientierte Entwicklung offener, verteilter Software systeme im Telecommunicationskontext”, Band I, II and III, Ph.D. Thesis, Humboldt Universität zu Berlin, 2002.

[Br99] Tim Bray, Dave Hollander, Andrew Layman, “Namespaces in XML”, January 1999, http://www.w3.org/TR/REC-xml-names/

[CCG+93] A. Campbell, G. Coulson, F. García, D. Hutchison and H. Leopold, “Integrated Quality of Service for Multimedia Communications”, Proc. IEEE INFOCOM’93, pp. 732-739, San Francisco, USA, April 1993.

[CCM01] OMG, CORBA 3.0 New Components Chapters, OMG document ptc/2001-11-03

[Ch88] D. Cheriton, "The V Distributed System", in *Communications of the ACM*, vol 31, no. 3, pp. 314-333, Mar 1988.

[Ch96] Chorus Systems, “Requirements for a Real-Time ORB”, ReTINA, Tech. Report RT/TR-96-8, May 1996.

[CORBA] OMG, Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.6, December 2001, OMG document formal/01-12-35

[De+99] A. Detti et al., “Supporting RSVP in a Differentiated Service Domain: an Architectural Framework and a Scalability Analysis”, ICC'99, Vancouver, Canada.

[Depl02] OMG, Deployment and Configuration of Component based Distributed Applications, RFP, OMG Document: orbos/2002-01-19.

[DiBa96] P. Dillenbourg, M. Baker, “Negotiation Spaces in Human-Computer Collaborative Learning”, in the proceedings of COOP'96. (Juan-Les-Pins, France, June).

[DKS90] A. Demers, S. Keshav, S. Shenker, “Analysis and simulation of a fair queueing algorithm”, *Journal of Internetworking: Research and Experience,* 1, January 1990, pp. 3–26.

[DYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, Sanjeev Krishnan, “Enterprise JavaBeans specification”, Version 2.0, Final Release, August 2001

[EDOC] OMG, “UML Profile for Enterprise Distributed Object Computing Specification”, final adopted specification, February 2002, OMG document PTC/02-02-05.

[ETS02] OMG, “Extensible Transports for Real-Time CORBA”, revised Submission, OMG document MARS/2002-06-01, Objective Interface Systems, Inc.

[Fa01] David C. Fallside, “XML Schema”, May 2001, http://www.w3.org/TR/2001/REC-xml-schema-0-20010502.

[FaHa01] G. Fábián, A.T. van Halteren, "The QoS Provisioning Service", DMMOS'2001 workshop at ECOOP.

[Fe99] G. Feher et al., "Boomerang---a simple protocol for resource reservation in IP networks", in IEEE Workshop on QoS Support for Real-Time Internet Applications, Vancouver, Canada, June 1999. http://boomerang.ttt.bme.hu/

[FHKV99] N. Fischbeck, E. Holz, O. Kath, V. Vogel, "Flexible support of ORB interoperability", 1999.

[FHLS02] G. Fábián, A.T. van Halteren, M. van de Logt, F. Stoinski, "Design of a middleware for QoS-aware distribution transparent content delivery", in proceedings of ISCC'02, Taormina/Giardini Naxos, Italy, July 2002.

[FKN02] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[FlJa95] S. Floyd, V. Jacobson, Link-sharing and resource management models for packet networks, *IEEE ACM Trans. Networking* 3 (4), August 1995, pp.365–386.

[Fr96] L.J.N. Franken, "Quality of Service Management: a Model-Based Approach" Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1996.

[FrKo98] Svend Frolund and Jari Koistinen, "Quality of Service Specification in Distributed Object Systems Design", Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS) Santa Fe, New Mexico, April 27-30, 1998.

[Ga+95] E. Gamma et al., "Design patterns: elements of reusable object-oriented software", Reading MA, Addison-Wesley Publishing Company, 1995.

[GuPe99] R. Guerin and V. Peris, "Quality-of-service in packet networks: Basic mechanisms and directions", Computer Networks, 31(3), 169-189, February 1999.

[Ha00] A.T. van Halteren, "A reflective QoS provisioning service for object middleware", 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.

[He92] A. Herbert, "The challenge of ODP", in Open Distributed Processing, 1992.

[HFG01] A.T. van Halteren, G. Fábián, E. Groeneveld, "Design and evaluation of a QoS provisioning service", DAIS'2001, Krakow, Poland, September 2001.

[HHW+00] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, Steve Byrne, "Document Object Model (DOM) Level 2 Core Specification", November 2000, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

[HHW97] S. Hall, D. Heimbigner, A van der Hoek, A.L. Wolf, "*An architecture for Post-Development Configuration Management in a Wide-Area Network*", Proc. Of Int. Conference on Distributed Computing Systems, Baltimore, Maryland, May 1997.

[HKB01] E. Holz, O. Kath and M.Born, "Manufacturing Software Components from OO-design models" The 5th IEEE International Enterprise Distributed Object Computing Conference EDOC 2001, September 01, Seattle (WA), USA

[HNNW99] A.T. van Halteren, A. Noutash , L.J.M. Nieuwenhuis, M. Wegdam, "Extending CORBA with specialized protocols for QoS provisioning", proceedings of International Symposium on Distributed Objects and Applications (DOA'99), September 1999.

[HNSW99] 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", Proceedings of Telecommunications Information Networking Architecture Conference 1999 (TINA '99), Apr. 1999.

[HTW98] Aart T. van Halteren, Patricia Tangney and Vincent Walsh, "An Interoperable Federated Naming Service Supporting a Pan-European Service Platform", ICAST 98, 1998.

[HWHN00] Cristian Hesselman, Ing Widya, Aart van Halteren, Bart Nieuwenhuis, "Middleware support for media streaming establishment driven by user-oriented QoS requirements", IDMS 2000, Enschede, The Netherlands, October 2000.

[IETF97] IETF, "Hypertext Transfer Protocol – HTTP/1.1", RFC 2068, 1997.

[ISO X.641] ISO/ITU-T Recommendation X.641 Information Technology – Quality of Service – Framework [ITU-T Recommendation X.641 | ISO/IEC 13236], 1997.

[ISO8402] ISO International Standard 8402, "Quality management and quality assurance – vocaluray", 27-01-2000.

[JacORB] Gerald Brose, "JacORB: Implementation and Design of a Java ORB", proceedings of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, September 30 - October 2, Cottbus, Germany, Chapman & Hall 1997, http://www.jacorb.org/.

[JMS02] Sun Microsystems Inc. "Java Message Service", version 1.1, April 2002.

[JNDI01] Sun Microsystems Inc., "Java Naming and Directory Interface Application Programming Interface", version 1.2, July 2001.

[KHSW00] O. Kath, A. v. Halteren, F. Stoinski, M. Wegdam, Mike Fisher, "Integrated Middleware Platform Management based on Portable Interceptors", DSOM 2000, Austin, Texas, USA, December 2000.

[KiGi87] W.J.M. Kickert, J.P. van Gigch, "A metasystem approach to organisational decision-making", in: *J.P. van Gigch (ed.), Decision making about decision making: meta-models and metasystems*, pp. 37-55, Abacus Press, 1987.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. "Aspect-Oriented Programming", in proceedings of ECOOP '97, Springer-Verlag LNCS 1241, June 1997.

[Ko01] R.P. Koster, "Integrating security in a quality aware multimedia delivery platform", MSc. Thesis, University of Twente, Enschede, The Netherlands, November 2001.

[Ko97] Jari Koistinen "Dimensions for Reliability Contracts in Distributed Object Systems", Technical Report HPL-97-119, Hewlett-Packard Laboratories, October 1997.

[Ko99] C. Kobryn, "UML 2001: A Standardization Odyssey", Communications of the ACM, vol. 42, no. 10, October, 1999.

[KSS01] M. Karsten, J. Schmitt, and R. Steinmetz, "Implementation and Evaluation of the KOM RSVP Engine", in proceedings of IEEE InfoCom 2001.

[La92] A.A. Lazar, "A Real-time Control, Management, and Information Transport Architecture for Broadband Networks", Proc. International Zurich Seminar on Digital Communications, pp. 281-295, 1992.

[Le90] A.C.J. de Leeuw. "Organisaties: management, analyse, ontwerp en verandering - een systeemvisie", Assen/Maastricht, Van Gorcum, 1990.

[LFG+00] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke., "CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids", In ACM Java Grande 2000 Conference, pages 97–106, San Francisco, CA, 3-5 June 2000.

[LiNa99] Baochun Li, Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*. Vol. 17, No. 9, 1632-1650, Sept. 1999.

[MaAs75] E.H. Mamdani and S. Assilian. An experiment in linguistic synthesis with a fuzzy logic controller. *Intl. J. of Man-Machine Studies* 7, 1-13, 1975.

[Me88] B. Meyer, Object-oriented Software Construction, Prentice Hall, 1988.

[Me91] Mejlbro, L, "QOSMIC-deliverable - General Aspects of Quality of Service and System Performance in IBC", 1991 RACE Deliverable RACE D510.

[Me92] B. Meyer, "Applying 'design by contract'", *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10), pp. 40-52, October 1992.

[MeHa98] J. de Meer and A. Hadif, "The Enterprise of QoS", tutorial at Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing, September 1998.

[Mej92] Mejlbro, L. "QOSMIC-deliverable D1.3C: QoS and Performance Relationships", 1992, RACE. Deliverable QOSMIC R1082.

[Mo65] G.E. Moore, "Cramming More Components Onto Integrated Circuits", Electronics, Volume 38, Number 8, April 19, 1965.

[Mo97] R. Moats, "URN syntax", RFC 2141, 1997, ftp://ftp.isi.edu/in-notes/rfc2141.txt

[MOF] Object Management Group: "Meta Object Facility, Version 1.3", OMG document ad/99-07-03.

[Mu93] Mullender, S. (Ed.), "Distributed Systems", Addison-Wesley, 1993

[NaBa01] Nagy, W and Ballinger, K, The WS-Inspection and UDDI relationship, Nov. 2001, http://www-106.ibm.com/developerworks/library/ws-wsiluddi.html.

[Ni86] Nii, H.P. "Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures", *AI Magazine,* **7**(2), 38–53, 1986.

[NiHa99] Lambert J.M. Nieuwenhuis, Aart T. van Halteren, "EURESCOM Services Platform", in proceedings of Telecommunications Information Networking Architecture Conference 1999 (TINA '99), Apr. 1999.

[NiWi00] L.J.M. Nieuwenhuis, I. Widya, "Quality of Service and Service Provisioning on a Competitive Market", in proceedings of USM2000, Munich, Germany, 2000.

[NWX00] Klara Nahrstedt, Duangdao Wichadakul, and Dongyan Xu. "*Distributed QoS Compilation and Runtime Instantiation*", in Proceedings of IEEE/IFIP International Workshop on QoS 2000 (IWQoS 2000), June 5-7, Pittsburgh, PA, 2000

[ODP1] ITU/ISO, Open Distributed Processing –Reference Model, "*Part 1: Overview*", International Standard 10746-1, ITU-T Recommendation X.901, 1996.

[ODP2] ITU/ISO, Open Distributed Processing –Reference Model, "*Part 2: Foundations*", International Standard 10746-2, ITU-T Recommendation X.902, 1995.

[ODP3] ITU/ISO, Open Distributed Processing –Reference Model, "*Part 3: Architecture*", International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[OES01] OMG, "CORBA Event Service", OMG document formal/01-03-01.

[OlHa98] Petra Oldengarm, Aart van Halteren, "A Multiview Visualisation Architecture for Open Distributed Systems", Proceedings of Computer Software & Applications (Compsac'98), 1998

[OLS01] OMG, "Life cycle Service Specification", September 2002, Version 1.2, OMG document formal/02-09-01.

[OMG-CWM] OMG, "Common Warehouse Metamodel (CWM) Specification", Version 1.0, October 2001, OMG document formal/01-10-01.

[ONaS02] OMG, "Naming Service Specification", September 2002, Version 1.2, OMG document formal/02-09-02

[ONoS02] OMG, "Notification Service Specification", August 2002, Version 1.0.1, OMG document formal/02-08-04

[ORBacus] ORBacus, http://www.iona.com/products/orbacus_home.htm

[OTrS00] OMG, "Trading Object Service Specification", Version 1.0, May2000, OMG document formal/00-06-27

[P806] EURESCOM P806-GI: A Common Framework for QoS/Network Performance in a Multi-Provider Environment; http://www.eurescom.de/Public/Projects/P800-series/P806/P806pr.htm.

[Pa92] A.K.J. Parekh, A generalized processor sharing approach to flow control in integrated services networks, Ph.D. thesis, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139, February 1992, No. LIDS-TH-2089.

[Pi94] L. Ferreira Pires, "Architectural notes: a framework for distributed systems development", Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1994.

[PI99] Object Management Group, "Portable Interceptors", OMG Document orbos/99-12-02 ed., December 1999.

[PLS+00] P.P. Pal, J.P. Loyall, R.E. Schantz, J.A. Zinky, R. Shapiro, J. Megquier, "Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration", in proceedings of ISORC 2000, The 3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing, March 15-17, 2000, Newport Beach, CA.

[PTM92] A. R. Puerta, S. W. Tu, & M. A. Musen, "The New World of Mechanisms", Fifth International Symposium on Knowledge Engineering, Seville, Spain, 38-46. 1992.

[Qu98] Quartel, D.A.C., "Action relations - Basic design concepts for behaviour modelling and refinement", CTIT Ph.D-thesis series, no. 98-18, University of Twente, Enschede, The Netherlands, 1998

[RAC+01] M. Rutherford, K. Anderson, A. Carzaniga, D. Heimgibner, A.L. Wolf, "Reconfiguration in the Enterprise JavaBean Component Model", Univerisity of Colorado, Technical Report CU-CS-925-01, December 2001

[RCV98] S.P. Romano, R. Canonico, and G. Ventre, “Enabling a QoS architecture for the Internet”, Final Report for the COST237 European Project, March 1998.

[RMI02] Sun Microsystems Inc. “Java Remote Method Invocation Specification, version 1.4”, 2002.

[Ru93] S. Rudkin, “Templates, types and classes in open distributed processing”, BT Technol. Journal, II(3), pp. 32-40, 1993.

[SAX98] XML-DEV, “The Simple API for XML(SAX)”, version 10. 11 May 1998. http://www.megginson.com/SAX/

[Sc97] D.C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services”, in Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA, Addison-Wesley, 1997.

[ScKu00] D.C. Schmidt, F. Kuhns. “An overview of the real-time CORBA specification”, in *IEEE Computer special issue on Object-oriented real-time distributed computing*, June 2000.

[ScVi97] D.C. Schmidt and S. Vinoski. “Object interconnections. Object adapters: concepts and terminology”, SIGS C++ Report, October 1997, http://www.cs.wustl.edu/~schmidt/PDF/C++-report-col11.pdf

[SeCa00] F. Siqueira and V. Cahill, “Quartz: A QoS architecture for Open Systems”, in proceedings of International Conference on Distributed Computing Systems, pp. 197-204, 2000.

[SGHP97] D.C. Schmidt, A.S. Gokhale, T.H. Harrison, G. Parulkar, "A High-Performance End System Architecture for Real-Time CORBA", *IEEE Communications Magazine*, Vol. 35, No. 2, Feb. 1997.

[SiCa00] F. Siqueira, V. Cahill, ”Quartz: A QoS Architecture for Open Systems”, 20th International Conference on Distributed Computing Systems (ICDCS’00), Taipei, Taiwan, April 2000.

[SOAP01] W3C, SOAP version 1.2 Working Draft, Oct. 2001, http://www.w3.org/TR/soap12-part1

[Sz97] C. Szyperski, “Component software – Beyond object-oriented programming”, ACM Press, New York, 1997.

[Te00] B. Tekinerdogan. “Synthesis-based software architecture design”, Ph.D. thesis. Univ. of Twente, Enschede, The Netherlands, 2000.

[TuBu01] P. Tuma, T. Buble, “Open CORBA Benchmarking”, Proceedings of SPECTS 2001, USA, 2001.

[UDDI] Universal Description, Discovery and Integration (UDDI) project, UDDI specifications, http://www.uddi.org/specification.html.

[UML] UML Revision Task Force, OMG Unified Modeling Language Specification, v. 1.3, document ad/99-06-08. Object Management Group, June 1999.

[UML-F] OMG, “Model interchange using CORBA IDL”, OMG document formal/01-09-76 and OMG document ad/01-02-17.

[VZL+98] R. Vanegas, J. A. Zinky, J. P. Loyal, D. Karr, R. E. Schantz, and D. E. Bakken. "Quo’s runtime support for quality of service in distributed objects” in proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing, September 1998.

[W3C98] W3C. "HTML 4.0 Specification", W3C Recommendation, 24-4-1998.

[WeHa00] 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.

[WOS00] Weibin Zhao, David Olshefski and Henning Schulzrinne, "Internet Quality of Service: an Overview", 2000.

[WPHN00] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, Bart Nieuwenhuis, "Using Message Reflection in a Management Architecture for CORBA", DSOM 2000, Austin, Texas, USA, December 2000.

[WPHN00] 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.

[WSDL01] W3C, Web Services Description Language (WSDL) 1.1, March 2001, http://www.w3.org/TR/wsdl.

[XuPa90] J. Xu and D.L. Parnas, "Scheduling process with release time, deadlines, precedence and exclusion relations", *IEEE Trans. Software Engineering*, Vol. SE-16, no. 3, pp. 360-369, March 1990.

Abbreviations

Short form	Expanded form
ANSA	Advanced Networked Systems Architecture
BEO	Basic Engineering Object
CCM	CORBA Component Model
CFS	Object Communication Middleware Feature Set
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
DPE	Distributed Processing Environment
DRP	Distributed Resource Platform
DSI	Dynamic Skeleton Interface
EAR	EJB Archive
EFS	Execution environment Feature Set
EJB	Enterprise Java Bean
GIOP	General Inter-ORB Protocol
HTTP	Hypertext Transport Protocol
ICT	Information and Communication Technology
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Networking
IP	Internet Protocol
IPC	Interprocess Communication
ISO	International Organization for Standardization
J2EE	Java 2 Enterprise Edition
JCP	Java Community Process
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface

JVM	Java Virtual Machine
MOF	Meta-Object Facility
NCCE	Native Computing and Communication Environment
OCI	Open Communication Interface
ODP-RM	Open Distributed Processing Reference Model
OMA	Object Management Architecture
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object Oriented Software Engineering
OSF	Open Software Foundation
OSF	Open Software Foundation
POA	Portable Object Adapter
QML	QoS Modeling Language
QOA	Quality Object Adaptor
QoS	Quality of Service
QPS	QoS Provisioning Service
QRR	QoS Runtime Representation
QuO	Quality Objects
RFP	Request for Proposal
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSVP	Resource reSerVation Protocol
SFS	General Purpose Services Feature Set
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
W3C	Worldwide Web Consortium
WSDL	Web Services Description Language
XML	eXtensible Markup Language