Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep
Informatica

B

Controlling Quality of Service in an Open Distributed Environment

Kristian A. Helmholt

advisors:
Prof.dr.ir. L.J.M. Nieuwenhuis
Ir. A.T. Halteren
Ir. D. Straat

May 1997

RuG

Contents

Preface

This Master's thesis is the result of my graduation assignment for Computing Science (Technische Informatica) at the University of Groningen. The work was conducted at KPN Research Groningen at the department of Communication Architectures and Open Systems (CAS)) during a period of ten months, which started at the first of July 1996.

The time I spent at KPN Research was a wonderful time. Not only was I surrounded with pleasant people and a healthy working atmosphere, my stay there also taught me what kind of scientist I would like to be. KPN Research also gave me an opportunity to explore the realms of computing science and the telecommunication industry. This had not been possible if several people had not been there to support me.

First of all I want to thank my supervisor Ir. Aart van Halteren who again and again spent time to bring order into the chaos of all my ideas. I want to thank my supervisor Ir. Dirk Straat for his comments and reviews, and professor dr.ir. L.J.M. Nieuwenhuis for sharing his vision on ODP-RM, QoS-control, fault-tolerance, etc.

I also should not forget to thank my fellow graduates. They made 'room A135/A133' a very pleasant environment to work in. Thanx Alard, Cano, Rob, Jeroen and Jurgen.

And finally I want to thank my friends (from the m38c) and my family for supporting me.

Kristian A. Helmholt

Groningen, May 1997

List of Abbreviations

CORBA	Common Object Request Broker Architecture
DAE	Distributed Application Environment
DPE	Distributed Processing Environment
ODE	Open Distributed Environment
ODP-RM	Open Distributed Processing Reference Model
QoS	Quality of Service
SLA	Service Level Agreement

1 Introduction

The telecommunications industry today is surrounded by rapid and uncertain changes in customer demands and technology. Deregulation, liberalization, and competition demand a lot from the industry: shorter development times, shorter time to market, higher degree of customization of services and a higher utilization of the network. To meet these demands, the industry needs a flexible and powerful (IT-)infrastructure.

At this time new IT-infrastructure standards are emerging. Examples are the Common Object Broker Request Architecture from the Object Management Group and DCOM from Microsoft. These architectures and their implementations are today's building blocks for geographically distributed telecommunication and computing platforms. These new computing platform offer the industry the possibility to develop application components that can support the business in a flexible way by re-use of components and standardized couplings between clients and service-providing objects. New services can be implemented faster than ever before.

The quality of a service can differ. For example a database could offer as a Quality of Service that the response-time of a search-request never exceeds 10 ms. In an ODE QoS can degrade because of non-deterministic causes: unknown amounts of End-Users, component loss and a changing configuration of available resources. Controlling QoS in an ODE is therefor not a trivial task. When QoS is not controlled can change due to the occurrence of the non-deterministic causes, the flexibility of an ODE becomes a major disadvantage because the service provider can no longer agree with the End-User on QoS. In this thesis the basic question throughout will therefor be: "How can QoS be controlled in an ODE?".

We will try to answer this question by examining QoS at different abstraction-levels by making use of the Open Distributed Processing Reference Model and by developing a mechanism for QoS-control in a ODE. This mechanism can be used for on-line migration of applications from host to host, replication of application-components, message order preserving multicasts, etc.

1.1 Scope and objectives of this thesis

In this thesis the focus is on several distinct points. First, the definition of QoS has not been uniquely established by the international scientific community, and also the notion of QoS in an ODE is unclear. We therefor need to define or model QoS in an ODE. Secondly, we want to control QoS by using existing mechanisms that can be found in an ODE. The mechanism must enable a QoS controlling application to operate properly in the presence of non-deterministic factors like End-User load, faults and variations in resource allocation. In short, we want to:

1. provide a model of Quality of Service in an Open Distributed Environment.
2. develop a mechanism (based on the use of redundancy) to control QoS in an Open Distributed Environment
3. develop a prototype QoS controlling application based on the proposed model and used mechanism to validate the model.

1.2 Approach and topics of this thesis

We will model QoS by looking at the concept from different levels of abstraction. Because the Reference Model for Open Distributed Computing encompasses the concept of describing a system at different abstraction levels, this reference model will be used. After having established a model of QoS in an ODE we will develop a model for controlling QoS: measurements, guarantees and realizing. We will not develop a model for controlling all kinds of QoS. For example, a topic like guaranteeing bandwidth on an connection in ODE will not be covered. Controlling QoS will be discussed in general and only in more detail where it concerns the effects of redundancy on QoS-level guarantees. We want to keep the examples we use as tangible as possible, and therefor we will make use of specific architectural frameworks such as TINA and OMG's OMA (including CORBA).

When we have modeled QoS in an ODE and provided an mechanism to control it, we develop a prototype QoS controlling application. This QoS controlling application is an addition to the project called the Universal Services Platform (UDP, i.e. in Dutch: Universeel Diensten Platform). The design trajectory of the prototype also shows how to make components of an existing ODE based application suitable for QoS Management, without having to redesign the architecture of the existing application.

1.3 Terminology

We define the concept of an Open Distributed Environment. We closely follow Leydekkers [17]:

> *The term 'Distributed Environment' in the thesis title is to be interpreted here as the infrastructure that provides the processing environment for distributed services. The infrastructure is the combination of computing nodes and a communication network that interconnects the computing nodes. [....] Open implies a heterogeneous environment comprising of a mixture of LAN, WAN and MAN networks and/or heterogeneous equipment.*

To give a more clear view on a typical distributed environment that is commonly used by the OMG and the TINA-C in Figure 1.. We see the two layers, the Distributed System and the Distributed Application Environment. On the distributed system layer we can see the hardware, the operating systems and the telecommunication network. On top of this hard- and software the middleware is located. This is software which makes it possible for an application to be geographically distributed overall several computing systems. Middleware can be considered as an extension to the traditional operating systems

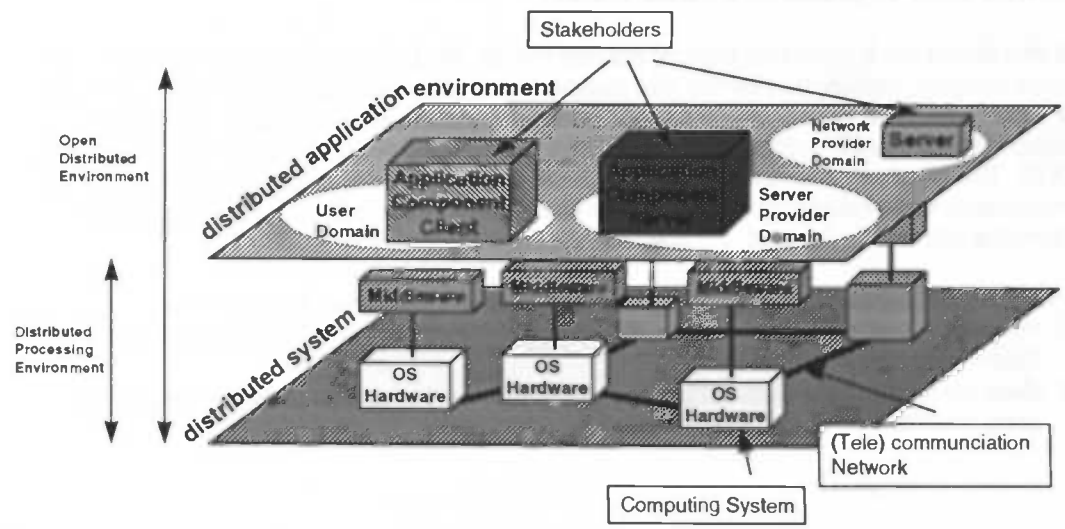


Figure 1 Terminology related to ODEs

interprocess communication system. It unifies the hard- and software components at the Distributed System layer into the Distributed Processing Environment. On top of this DPE the application components, like client and server, reside. This is the environment where *services* are provided to the *distributed applications*. The union of the two layers is called the Open Distributed Environment, which supports a particular *architecture*. It can be characterized by its *openness* and the *distribution transparencies* it offers. An important aspect of openness is *heterogeneity* in equipment, in operating systems and authority (e.g. cooperation between autonomous network operators).

1.4 Structure of this thesis

Figure 2 shows how the seven chapters of this thesis are interrelated:

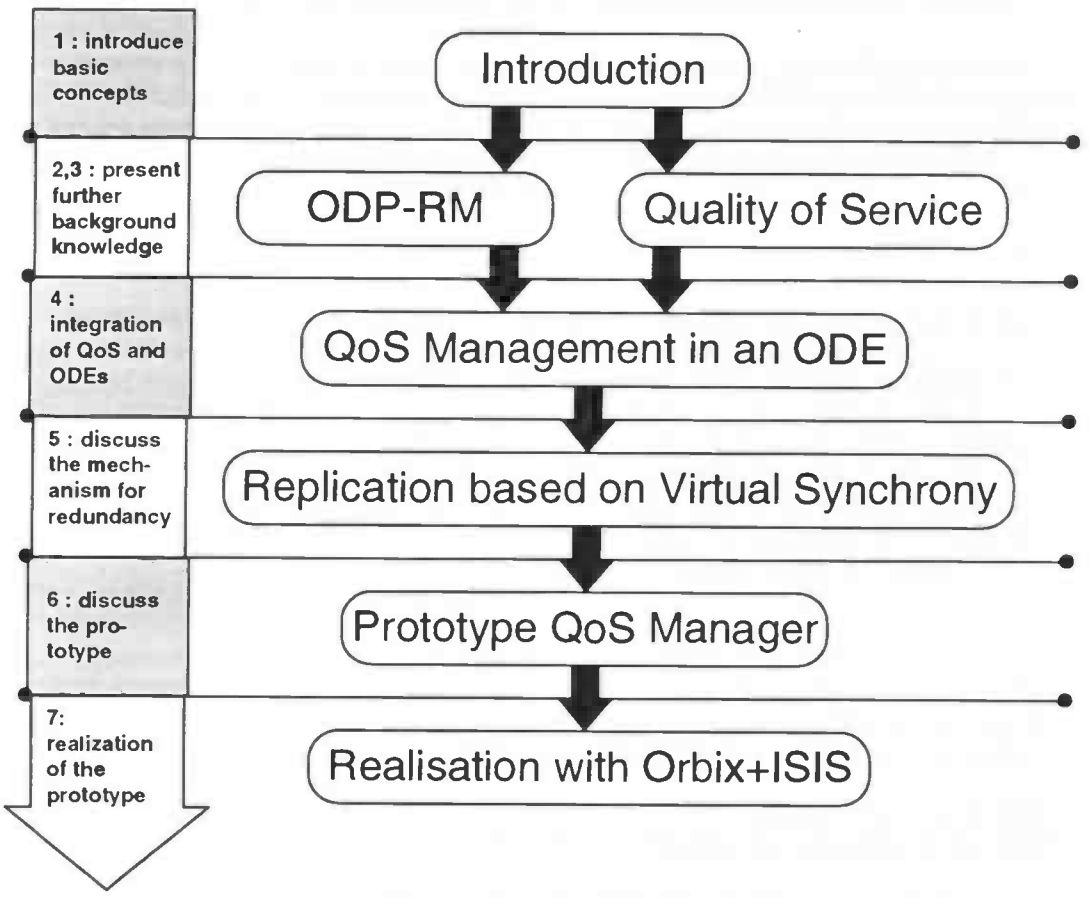


Figure 2 Relationship between the chapters

In Chapter 2 we discuss the ODP-Reference Model with respect to management and coordination of objects in an ODE, which will serve as a basis for controlling QoS in an ODE. In Chapter 3 we model Quality of Service in an ODE. These two chapters provide the material for developing a model for controlling QoS in Chapter 4. In Chapter 5 we will look at a replication-mechanism which is used in the model for controlling QoS. And in Chapter 6 we will look at the design of the prototype QoS controlling application. Finally, in Chapter 7 we will discuss the realization of the prototype application in the Orbix+ISIS environment.

2 Reference Model of Open Distributed Processing

In today's world different architectures of (open) distributed IT-infrastructures emerge. Although they differ in use of terminology and internal structure, they all have basic concepts in common. For example, geographically distributed computing parts that interoperate to provide a service are sometimes named *distributed objects* and sometimes *distributed components*. The concept is the same, the difference is in the name. We want to develop models that can be used in many architectures, not one.

If the models are to be used in more than one architecture, they should be presented using terminology that is universally applicable in modern day ODEs and ODEs to come. To this we state that the of complexity QoS in an ODE demands we discuss a model of this subject at different levels of abstraction. In this chapter we discuss a reference model (ODP-RM) which meets this demands.

2.1 Why ODP-RM?

To see why we use ODP-RM and why it is suitable for our task, we need to look at the goals of ODP-RM. The objective of ODP is to enable seamless interworking of distributed application components regardless of various forms of heterogeneity [17]. This can only be achieved by developing good standards. A prerequisite for this is having a reference model. ODP-RM provides such a reference model which supports modeling distribution, interworking, and portability. It is meant to cover all relevant aspects of distributed systems to coordinate the development of the various standards needed for open distributed systems. An additional reason for choosing ODP-RM is its use of *object orientation*. This aspect results in high modularity, which is a great advantage while modeling complex systems.

The Reference Model of Open Distributed Processing (ODP-RM), ITU-T recommendations X.901 to X.904 I ISO/IEC 10746, is based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture. There are four parts:

1. Overview and Guide to Use (ITU-T X.901), contains a motivational overview of ODP giving scope, justification and explanation of key concepts and an outline of the ODP architecture .
2. Descriptive Model (ITU-T X.902), contains the definition of the concepts and analytical framework and notation for normalized description of (arbitrary) distributed processing systems.
3. Prescriptive Model (ITU-T X.903) [11], contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards MUST conform.
4. Architectural Semantics (ITU-T X.904), this part contains a formalization of the ODP modeling concepts defined in the descriptive model.

We will not give a complete lecture on all four documents in this thesis. We will focus on those aspects of the Reference Model that concern the objectives of this thesis, so that the interested reader is not obliged to refer to the documents mentioned above.

2.2 Objectives of ODP-RM

In this section we describe the objectives of ODP-RM. The first thing ODP aims to achieve is collaboration of applications across heterogeneous platforms (see Figure 1). Secondly, it aims to achieve interworking between ODP systems, i.e. meaningful exchange of information and convenient use of functionality throughout the distributed environment. And thirdly it defines layers of abstraction through the definition of distribution transparency, i.e. hide the consequences of distribution from both the applications programmer and user. By describing what services (functions) the DPE should offer to the DAE, it provides concepts to describe the to the application component visible layer of middleware.

Before looking at how ODP-RM uses of abstraction to describe an entire system, we will look at some major features of object-orientation that relate to abstraction at the component or object-level in ODP-RM.

Encapsulation
The property that the information contained in an object is accessible only through interfaces supported by an object is called encapsulation. This effects of this property can be found throughout ODP-RM: the reference model does not speak of modules or units, but of objects, that have interfaces. They are the only means to access an objects.

Abstraction
The property that internal details of an object are not visible to other objects is called abstraction. This property is noticeably present at the different abstraction levels within the Reference Model where objects exist that only have 'external' properties. How they are implemented or defined from the inside is not mentioned (and should not be according to ODP-RM).

These OO principles help describing a complex system by decomposing it at abstraction levels, were internal details do not obstruct global overviews. Based on these principles, the Reference Model provides terminology to describe the concepts and roles of components in an Open Distributed Processing that organizes the pieces of an ODP system into a coherent whole. Note that it does NOT try to standardize the components of the system nor to influence the choice of technology. It is a reference model that must be adequate to describe most distributed systems available both today and in the future. So it is a very abstract model, that carefully describes its components without prescribing an implementation.

2.3 Abstraction based on viewpoints

In this section we will present the viewpoints of ODP-RM by using an example. We will look at a service from different points of view. In each viewpoint different aspects of a service are visible, like enterprise, computational and engineering aspects. Each of the five viewpoints in ODP-RM is accompanied by its own viewpoint language, that contains concepts and rules for specifying a system. By using each of the viewpoint languages ODP-RM allows a large and complex specification of an ODE to be separated into manageable pieces, each focused on the issues relevant to different viewpoints of abstraction. We will now cover these five viewpoints and present specific viewpoint languages.

2.3.1 Enterprise Viewpoint

A specification in the Enterprise Viewpoint describes the overall objectives of a system in terms of roles, actors and goals. In other words, it specifies the requirements of an ODP system from the viewpoint of an End-User. The concept of Service Level Agreements is also related to an Enterprise Viewpoint description. Note that distribution aspects that may be applicable to the system are not visible in this viewpoint, nor are specific hard- or software architectures.

An ODP system is represented in terms of interacting agents, working with a set of resources to achieve business objectives subject to the policies of controlling objects. Objects with a relation to a common controlling object can be grouped together in domains which form federations with each other in order to accomplish shared objectives. Such a federation, in which the members are mutually contracted to accomplish a common purpose is called a *community*.

The objects are mostly concerned with so called performative actions that change *policy*, such as creating an obligation or revoking permission. By defining an enterprise specification of an ODP application, policies are determined by the organization rather than imposed on the organization by technology (implementation) choices. Objects that are able to initiate actions have an *agent* role, whereas those that respond to such initiatives have *artifact* roles.

Example language

Several languages could be used to give a description of this viewpoint, as long as the language is able to reflect the relevant issues of the enterprise, that is the user requirements and policies. A good example of such a language is the so called 'Use Cases' , which is part of the Unified Modeling Language (UML). The basic concepts involved with Use Cases are *actors* and *systems*. A *system* is considered to be a regularly interacting or interdependent group of items forming a unified whole. An *actor* is an entity which is outside the system and communicates directly with the system. An actor can be a human, or another system. One human can represent several actors. From the systems point of view the actors are the representation of the outside world. The system only communicates with actors. So a typical Use Case is a description of a series of interactions between several actors and the system.

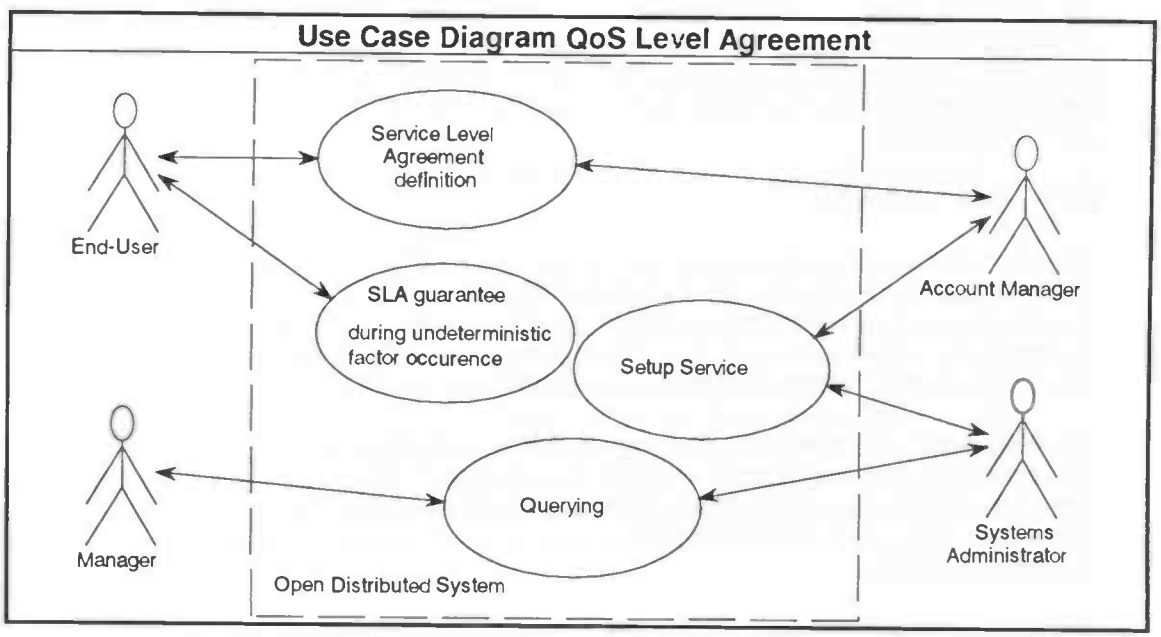


Figure 3 Use Case Diagram Example of QoS Level Agreement

In Figure 3 we provide an example of a set of Use Cases. We can observe four Use-Cases. They all concern the use of and interaction with a system that provides dependable open distributed processing. The collection of Use Cases clearly shows that there are several actors that can influence the system. There is the End-User, who wants to be provided with a set of services at a certain level. There is the Account Manager, who is able to translate the often 'vaguely expressed' wishes and demands of the End-User into unambiguous Service Level Agreements. There is the System Administrator who is able to inspect the internals of the system and can make changes to the system

(configuring hard- and software, installing new hard- and software). And there is the Manager, who is in fact a symbol for all people that need information about the 'dependability' of the system at different abstraction levels and from different viewpoints.

Use Case Catalogue Detail: 'Service Level Agreement definition'

Used by:
End-User, Account Manager

Description:
The End-User states his demands and the Account Manager translates them so that the Open Distributed System can understand the demands. If the demands are ambiguous the Account Manager asks the End-User again and again until the demands are unambiguous. When everything is clear, the Account Manager feeds the system with the demands. The system checks whether or not these standards can be met. If this is the case, the system reports at which cost this level of quality will be achieved. The End-User can now agree or can not agree. In the first case the standards are set. In the second case the End-User is asked to change his demands. This process of stating demands, feeding them to the system, evaluating the cost continues until the End-User agrees or when the End-User does no longer wants to define the SLAs.

Intent:
Making the system aware of what level of service the End-User wants it to provide.

preconditions:
There is an End-User available who has a clear vision of what kind of service level he/she expects. There is an Account Manager available that is able to translate the 'vague' demands of the End-User into definitions that the system 'understands'.

postconditions:
The systems knows what the level of dependability for a certain service should be.

Figure 4 A specific Use Case 'SLA definition'

In Figure 4 A specific Use Case 'SLA definition' we give a detailed description of one of the Use Cases from the collection in Figure 3 Use Case Diagram Example'. In the Used By section the actors are listed who participate in this case. Then a Description follows which explains what happens. In the Intent section, a brief statement is made what this Use Case should demonstrate. In the Pre Conditions we see how the actual situation before commencing the Use Case must be, in order to get to the situation as described in the Post Conditions.

2.3.2 Information Viewpoint

The information viewpoint is concerned with the information storage and processing in the system. It is used to describe the information required by an ODP application through the use of schemes, which describe the state and structure of an object. The main concern is the syntax and semantics of the information within the system. The flows of information and the information itself in the system are also located and identified.

The concepts in the information language enable the specification of information manipulated by and stored within an ODE. More complex information is represented as composite information objects expressing relationships over a set of constituent information objects. As mentioned before, three kinds of schema's are used to specify information objects:

1. **static** schema, captures the state and structure of an object at some particular instance;
2. **invariant** schema, restricts the state and structure of an object at all times; In database terms: in this schema one can find the constraints on attributes and on operations. Example: the call-duration-time is always greater than or equal to 1 second, because the setup-time takes 1 second.
3. **dynamic** schema, defines a permitted change in the status and structure of an object;

Schema's can be used to describe relationships or associations between objects; e.g. the static schema "owns phone-number" could associate each phone-number with an owner. Furthermore, a schema can be composed from other schema's to describe complex or

composite objects. For example a Telephone Directory consists of a set of clients, a set of phone-numbers and "owns phonenumber".

In addition to describing state changes, dynamic schema's can also create and delete component objects. This allows an entire information specification of an ODP system to be modeled as a single (composite) information object.

Example language
There are several languages/techniques which can cope with schema's, like OMT (for a more detailed of the Object Modeling Technique, see [33]) and Z. In the example in Figure 5 we have used OMT to describe an invariant schema of information on problems with an ODE.

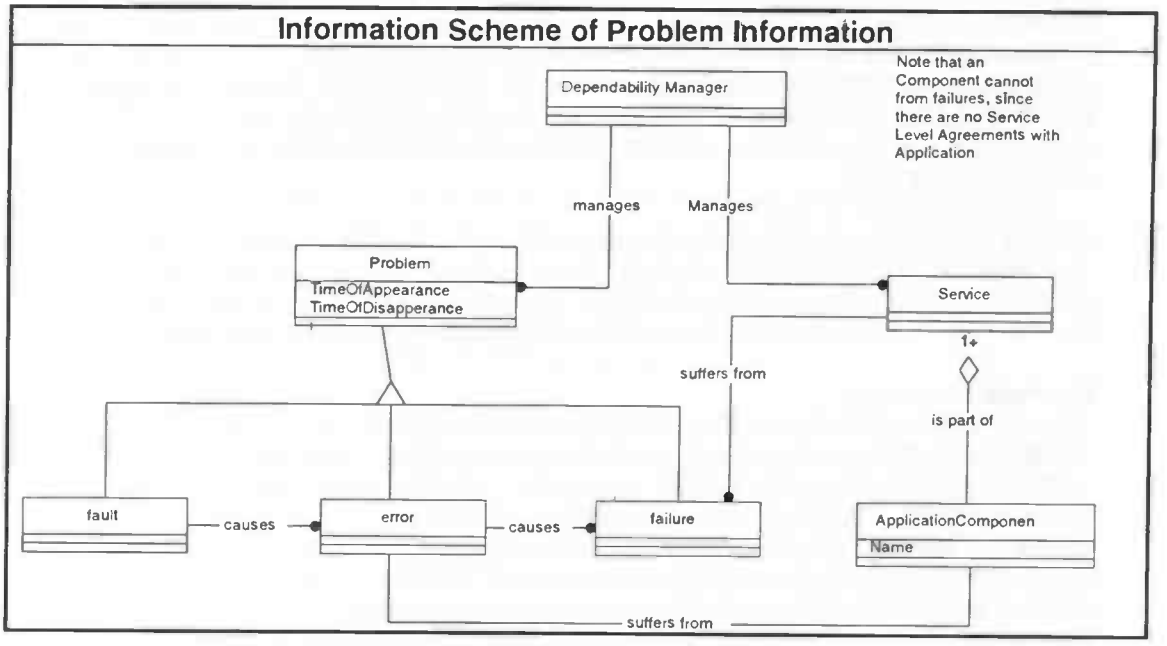


Figure 5 Information Scheme Example

The example shows a general object *Problem* which has the following information attributes: a time of appearance and a time of disappearance. From this general problem object three problem object specialization's can be derived: a fault, an error and a failure. These three objects have a 'causes' relation. The problem information is managed by a 'Dependability Manager' which also manages certain services. So the Dependability Manager needs information on problems in the system and information on the services in the system. A service can suffer from a failure, which is caused by an error. A service consists of separate application components which can suffer from errors.

2.3.3 Computational Viewpoint

The purpose of the computational viewpoint is to provide a functional decomposition of an ODP system. Application components are described as computational objects (for a graphical depiction see Figure 6 Computational objects manipulate the objects defined in the Information Viewpoint in order to achieve the objectives and requirements defined in the Enterprise Viewpoint. Such an object provides a set of capabilities that can be used by other computational objects, which enables them to interact with each other. A set of related capabilities is called a *computational interface.* A computational specification of a distributed application specifies the structure by which these interactions occur and specifies the semantics.

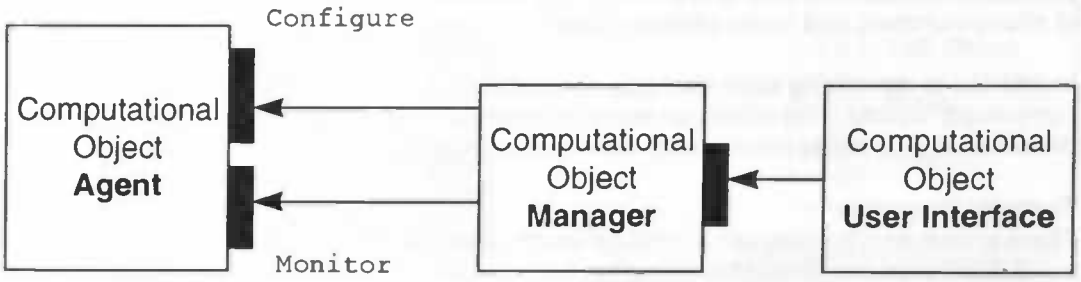


Figure 6 Graphical Computational Viewpoint Notation

In Figure 6 Graphical Computational Viewpoint Notation we provide a small example that shows three computational objects: A manager that uses an agent (actually several, but for simplicity's sake not depicted) to *configure* and *monitor* the system. In order for the agent to provide these two services it provides two interfaces: the *ConfigurationAgent* interface and the *MonitorAgent* interface (see the example language for a verbal description). The user interface is a separate object with functions for presenting information to a user.

Although the figure might suggest otherwise, the physical distribution is not a concern in this viewpoint. It just shows the structure of the distributed application. Each of the computational objects can be subdivided in other components (see the next section: the Engineering Viewpoint), that reside on different geographically distributed sites.

Example Language

The computational viewpoint is used to describe the structure and components of a distributed application. By making use of encapsulation and abstraction, the focus can be on the computational structure of the application. This requires defining the interfaces of the components. Several major ODE architectures have a language for describing interfaces. We call these languages interface definition languages and we state they can serve as a language to describe the Computational Viewpoint. The Common Object Request Broker Architecture (CORBA, part of the Object Management Architecture defined by the Object Management Group (OMG)) has defined the Interface Definition Language (OMG-IDL). The Telecommunications Information Networking Architecture defined by the TINA-Consortium has an Object Definition Language (ODL) and the Inter Language Unification project from Xerox PARC has an Interface Specification Language (ISL).

```
// Agent.idl
// IDL defintion of the interfaces of the Agent

#include "deptypes.idl"

interface ConfigurationAgent {

  // IDL operations
  short Kill(in ApplicationComponent AppComp);
  short Launch(in ApplicationComponent AppComp);

};

interface MonitorAgent {

  // IDL operations
  void Probe(inout SystemInfo SysInfo);

};
```

Figure 7 An example interface description in OMG-IDL

2.3.4 Engineering Viewpoint

In the Engineering Viewpoint the physical structure of the distributed system becomes visible, although the exact specifications (like the type of machines that are used) are irrelevant at this stage. To be more precise, the mechanisms used to provide the support for the Computational Objects are described. In this viewpoint the correlation between the distributed applications and the supporting DPE is described. We will now present the different engineering concepts for describing the infrastructure required to support distribution transparent interaction between objects. Figure 8 depicts the graphical engineering concepts.

We begin with the *nucleus*, which is the engineering abstraction of an operating system, which has its habitat on a *node* (a host machine). It coordinates processing, storage and communication functions used by other engineering objects within the same node. All basic engineering objects (BEOs) are bound to a nucleus.

Tied to the nucleus is the *capsule* (there can be more than one). It is a configuration of objects forming a single unit for the purpose of encapsulation of processing and storage. It is a subset of the resources of a node. If a capsule fails, only the objects inside the capsule are affected and the objects outside the capsule remain unaffected. An example of a capsule is a *process* in the UNIX operating system: if one process crashes, it does not immediately result in the crash of other processes.

In a capsule reside clusters. They are a configuration of basic engineering objects forming a single unit of deactivation, checkpointing, recovery and migration. The aspect of checkpointing (which is related to the subject of synchronization) and migration will be covered in more detail later on in this thesis. These mechanisms should however be described in this Viewpoint. And they should be supported by the environment in which the distributed applications are embedded.

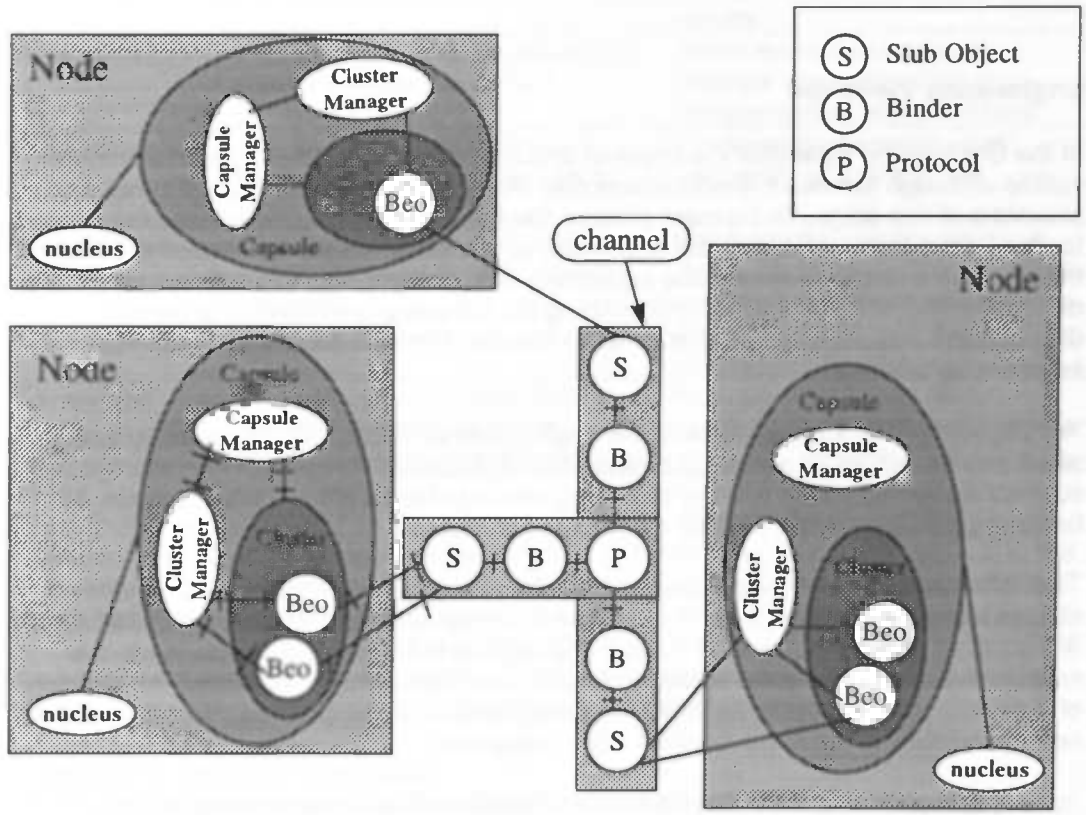


Figure 8 Engineering Viewpoint Graphical Notation

A *Basic Engineering Object* (BEO) is an engineering object that requires the support of a distributed infrastructure; it is the most elementary building block in this Viewpoint. BEOs are grouped together in a cluster and have an engineering interface which is either bound to another engineering object within the same cluster or channel. They are also always bound to the nucleus.

And finally there is the concept of a *channel*, which serves to support distribution transparent interaction of basic engineering objects. It can cross the boundary of a cluster, a capsule and even a node. It is defined as a configuration of *stub*, *binder*, *protocol* objects.

Stub
A stub-object provides conversion for data. It provides wrapping and coding functions for the parameters of an operation. In this way the parameters of an operation can be presented to the binder object as a sequence of bytes. The reverse is also possible, which means that the stub unwraps the sequence of bytes that come from the binder. This process of wrapping and coding is also known as *marshaling*.

Binder
A binder-object manages the end-to-end integrity of a channel. It takes care of directing the sequences of bytes to the correct target stub object.

Protocol
The protocol-object communications with other protocol objects to achieve interaction between engineering objects. According to ODP-RM they should be able to work together in their own communication domain (TCP/IP for example is not the same domain as SPX/IPX). In [24] Nankman defines a special group protocol-object. Later on in this thesis we will also define a same kind of group protocol-object, since we need it to describe the concept of group-communication through virtual synchrony.

Example language
Although we do not know of any major standard language that is able to an engineering object and the related mechanisms. We do know that CORBA supports the demands of this viewpoint to a fairly large extent. For example, in CORBA the stubs for several languages have been defined. These are called the language mappings and are available for C, C++, Smalltalk, etc. These stub definitions exactly describe how to approach another object in the system and how to transfer information between objects. See also section 7.2.

2.3.5 Technological Viewpoint

In the Technological Viewpoint we can see the system as a collection of physical components, both hardware and software. The viewpoint language makes use of technological objects, which must be names of standards which can be or are implemented. They can be components as operating systems, peripheral devices, or communication hardware. In this viewpoint the physical layout and mechanisms from the DPE become clearly visible. In fact, it can be used as a further refinement of the engineering viewpoint.

Example language
A lot of techniques and implementations are available for implementing an ODE, like Microsoft Winsock as an implementation for TCP/IP communication protocol and ILU, Chorus and Orbix+ISIS for Object Request Broker like implementations. However a generic viewpoint language does not exist, because a description of the system in the technological viewpoint depends on (and is written in terms of) the technology that is used.

2.4 Distribution Transparencies

Middleware transforms a heterogeneous set of computers and network links into a integrated computing platform. It makes the underlying network and computers transparent to an application (designer), with respect to the distribution of the application and the ODE. ODP-RM defines this as Distribution Transparency, which consists of a number of major categories (like Access, Location and Failure Transparency, see Table 1 for a complete listing). The distribution transparencies are realized by functions. These functions are in fact mechanisms which are visible in the Engineering Viewpoint. The results of applying these functions can be observed as distribution transparencies in the Computational Viewpoint. In other words, the Computational Model *assumes* transparencies, while the Engineering Model describes mechanism for *realization* of transparencies.

Table 1 Transparencies [17]

Transparency	Masks	Effect
Access	the difference in data representation and invocation mechanisms to enable interworking between objects.	Solves many of the problems of interworking between heterogeneous systems.
Failure	the failure and possible recovery of other objects (or itself) to enable fault-tolerance.	The designer can work in an idealized world in which the corresponding class of failures does not occur
Location	the distribution in space of interfaces. Location transparency for interfaces requires that interface identifiers do not reveal information about interface location.	Provides a logical view of naming, independent of the actual physical location.
Migration	the ability of a system to change the location of that object	Migration is often used to achieve load balancing and reduce

		latency.
Relocation	the relocation of an interface from other interfaces bound to it.	Allows system operation to continue when migration or replacement of objects occurs.
Replication	the use of a group of mutually behaviorally compatible objects to support an interface	Enhances performance and availability of applications.
Persistence	from the object the deactivation and reactivating of other objects (or itself).	Maintains the persistence of an object when the system is unable to provide it with processing, storage and communication functions continuously.
Transaction	the coordination of activities amongst a configuration of objects to achieve constancy.	Provides guarantees about interactions between applications.

In this thesis we use several transparencies to describe QoS control in an ODE. Failure-Transparency is used for describing the behavior of the DPE when controlling QoS in the presence of faults. Replication-Transparency is used for describing the behavior of the DPE when replicated software components are used to control QoS (e.g. more processing elements can enhance the QoS). Migration-Transparency is used to describe the behavior of the DPE when migration of objects is used to control QoS (e.g. moving objects from a heavily used computing node to a less used node can enhance the QoS).

Consistency between viewpoints
To ensure a correct mapping between the computational and the engineering viewpoint, i.e. no statements are made in both viewpoints that contradict each other, ODP-RM defines a set of rules that guarantees consistency from which we will now present the rules that are relevant to the subject of this thesis:

1. Each computational object which is not a binding object corresponds to a set of BEOs (and any channels which connect them). All the basic engineering object in the set correspond only to that computational object

2. Except for transparencies that require the replication of objects, each computational interface corresponds to one engineering interface, and that engineering interface corresponds only to that computational interface. Where replicated objects (and interfaces) are involved, each computational interface of the objects being replicated correspond to a set of engineering interfaces, one for each of the basic engineering objects resulting from the replication. These engineering interfaces each correspond only to the original computational interface.

3. Each computational binding corresponds to either an engineering local binding or an engineering channel. This engineering local binding or channel corresponds only to that computational binding.

2.5 ODP Functions

The functions that describe the behavior of objects in both viewpoints are supplied in several categories. We discuss those functions that concern the behavior and role of objects that (interoperate to) control QoS. This information is mostly taken directly from the Reference Model [11]. Figure 9 contains a organized collection of such functions.

2.5.1 Management Functions

When discussing QoS-control, we need to discuss the management functions of ODP-RM. They describe the behavior of objects in management situations.

Node management function
The node management function controls processing, storage and communication functions within a node. It is provided by each nucleus at one or more node management interfaces. Each capsule uses a node management interface distinct from the node management interfaces used by other capsules in the same node. It manages threads, accesses clocks and manages timers and creates channels and locates interfaces. Within the architecture defined by this Reference Model, the node management function is used by all other functions.

Object management function
The object management function checkpoints and deletes objects. When an object belongs to a cluster that can be deactivated, checkpoint or migrated, the object must have an object management interface in which it provides one or more of the following functions: checkpointing the object and deleting the object. The object management function is used by the cluster management function. This function is important for the management of QoS-level guarantees, because many techniques used in providing QoS-level guarantees in the presence of faults (i.e. provide fault-tolerance) are based on checkpointing, like freezing the state of an object or transferring the state of an object, as we will see later on in this thesis.

Cluster management function
The cluster management function checkpoints, recovers, migrates, deactivates or deletes clusters and is provided by each cluster manager at a cluster management interface, comprising one or more of the following functions with respect to the managed cluster:

- modifying cluster management policy (e.g., for the location of checkpoints of its cluster, for the use of the relocation function to trigger reactivation or recovery of the cluster);
- deactivating the cluster;
- checkpointing the cluster;
- replacing the cluster with a new one instantiated from a cluster checkpoint (i.e., deletion followed by recovery);
- migrating the cluster to another capsule (using the migration function);
- deleting cluster.

The behavior of a cluster manager is constrained by the management policy for its cluster. Cluster checkpointing and deactivation is only possible if all objects in the cluster have object management interfaces supporting the object checkpointing function. Deactivation and cluster deletion both require that the objects in the cluster support object deletion.

Within the architecture defined by this Reference Model, the cluster management function is used by the capsule management function, the deactivation and reactivation function, the checkpoint and recovery function, the migration function and the engineering interface reference management function; the cluster management function uses the storage function for keeping checkpoints.

Although this function might seem fit for modeling QoS control mechanisms like migration and replications, this is *not* the case. A cluster of objects must reside in one capsule on *one* node: this severely limits the use of this function for example when describing a group of identical objects that reside on several distinct nodes.

Capsule management function
The capsule management function instantiates clusters (including recovery and reactivation), checkpoints all the clusters in a capsule, deactivates all the clusters in a capsule and deletes capsules. It is provided by each capsule manager at a capsule management interface comprising one or more of the following functions with respect to the managed capsule:

- instantiation (within the capsule) of a cluster template; this includes reactivation and recovery.
- deactivation of the capsule by deactivating all the clusters within it (using the cluster management function);
- checkpointing the capsule by checkpointing all the clusters in the capsule (using the cluster management function);
- deleting the capsule, by deleting all the clusters within it, followed by deletion of the capsule manager for the capsule.

And for this function the same goes as for the cluster management function: clusters reside on *one* node, which severely limits the use in this thesis.

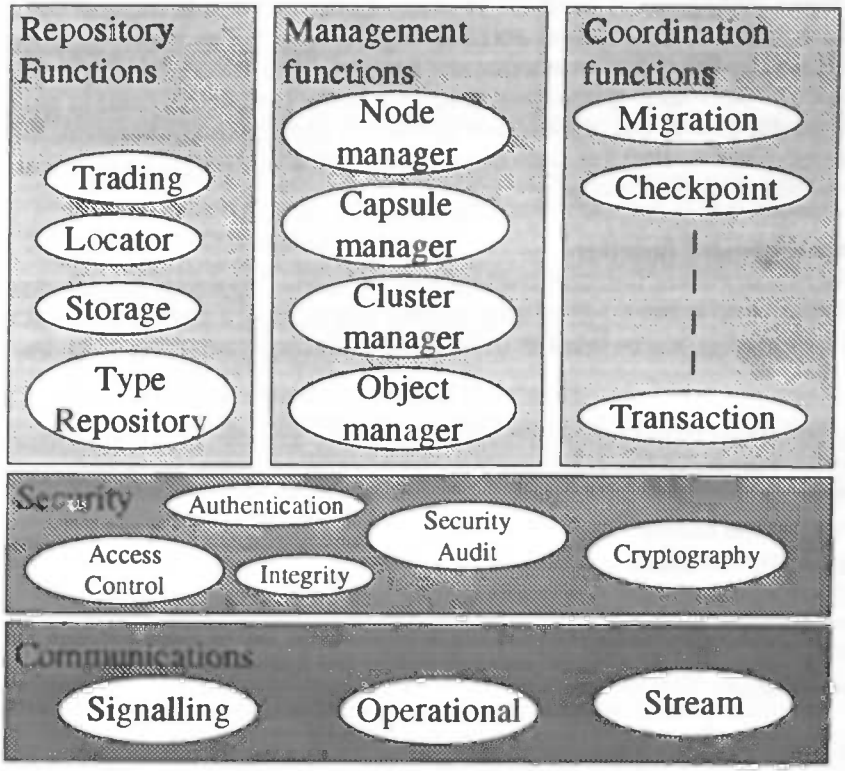


Figure 9 Some engineering functions

2.5.2 Coordination functions

All actions that are performed have to be coordinated by the ODE. In order to describe the coordination of many possible events and actions in an ODE, the ODP-RM provides coordination functions. This category of functions contains many important functions that enable us to describe the management of QoS-levels in an ODE.

Event notification function
The event notification function records and makes event histories available. This is very useful when the ODE has to notify an (interested) object (somewhere in the ODE), that an other object is no longer able to comply to the (guaranteed) QoS.

Checkpoint and recovery function
The checkpoint and recovery function coordinates the checkpointing and recovery of failed clusters. Although we think these functions lack the possibility of describing the checkpointing and recovery of objects that reside on different nodes, we do present a description, because we use this description when describing our extension of the group function.

The checkpoint and recovery function embodies policies governing

- when clusters should be checkpointed.
- when clusters should be recovered.
- where clusters should be recovered.
- where checkpoints should be stored.
- which checkpoint is recovered.

The checkpointing and recovery of clusters is subject to any security policy associated with those clusters, in particular, where the checkpoint is stored and where the checkpoint is recovered. Within the architecture defined by this Reference Model, the checkpoint and recovery function uses the cluster management function and the capsule management function.

Deactivation and reactivation
The deactivation and reactivation function coordinates the deactivation and reactivation of clusters. It embodies policies governing

- when clusters should be deactivated;
- where the checkpoint associated with a deactivation should be stored;
- when clusters should be reactivated;
- which checkpoint should be reactivated (e.g., the most recent);
- where clusters should be reactivated.

The deactivation and reactivation of clusters is subject to any security policy associated with those clusters, in particular. Within the architecture defined by this Reference Model, the deactivation and reactivation function uses the object management function, the cluster management function and the capsule management function. The deactivation and reactivation function are used by the migration function.

Group function
The group function provides the necessary mechanisms to coordinate the interactions of objects in multi-party binding. A interaction group is a subset of the objects participating in a binding managed by the group function. For each set of objects that is bound together in an interaction group, the group function manages:

- **interaction**: deciding which members of the group participate in which interactions, according to an interaction policy;
- **collation**: derivation of a consistent view of interactions (including failed interactions), according to a collation policy;
- **ordering**: ensuring that interactions between group members are correctly ordered with respect to an ordering policy;
- **membership**: dealing with member failure and recovery, and addition and removal of members according to a membership policy.

The behavior of the binding object, linking members of the group, determines how interaction is to be effected.

Replication function
The replication function is in fact a special case of the group function in which the members of a group are behaviorally compatible (e.g., because they are replicas from the same object template). The replication function ensures the group appears to other objects as if it were a single object by ensuring that all members participate in all interactions and that all members participate in all interactions in the same order.

The membership policy for a replica group can allow for the number of members in a replica group to be increased or decreased. Increasing the size of a replica group achieves the same effect as if a member of the group had been cloned and then added to the group in a single atomic action.

For the replication function to be applied to a cluster, the objects comprising the cluster are replicated and configured into a set of identical clusters. The corresponding objects in each such replicated cluster form replica groups. Thus a replicated cluster is a coordinated set of replica groups. The replication function is used by the migration function.

Migration function
The migration function coordinates the migration of a cluster from one capsule to another. It uses the cluster management function and the capsule management function and embodies policies governing when clusters should be migrated and where they can be located.

Two possible means of migration are

1. *replication*, migration of a cluster by use of the replication function comprises the following sequence of actions:
   a) the old cluster is treated as a cluster replica group of size one
   b) a copy of the original cluster is created in the destination capsule, together with a cluster manager
   c) the objects in both the two clusters are formed into replica groups (of size two)
   d) the objects in the old cluster are removed from the object groups (leaving groups of size one)
   e) the old cluster (and its manager) is deleted.
2. *deactivation in one capsule followed by reactivation in another*, Migration of a cluster by deactivation and reactivation is coordinated by the cluster's manager, and comprises deactivating the cluster at its old location, followed by reactivating the cluster at its new location.

2.6 Conclusions

In this chapter we have shown that the use of the ODP-RM enables us to describe models for QoS-control in an ODE from different levels of abstraction or different viewpoints. By describing the mechanism for QoS-control in an ODE in terms of the functions that are provided by ODP-RM we can model a mechanism for QoS-control in the Engineering Viewpoint which is distribution transparent in the Computational Viewpoint. Furthermore, by using ODP-RM the proposed models can be applied in several (open) distributed environments like CORBA and DCOM.

3 Quality of Service

We now posses an understanding of what an ODE is and we are able to describe concepts relating to controlling (e.g. management and coordination) objects in an ODE at different levels of abstraction. Since services are provided by (interoperating) objects and we are now able to describe the controlling of objects, we proceed with describing how the QoS of the service can be controlled. We begin with defining QoS and continue with decomposing it. After that we will discuss the delivery of QoS.

3.1 What is Quality of Service?

QoS has been defined in many ways. One could say it expresses the "goodness" of a service as the End-User perceives it. Since this is not a definition which can be used for abstract reasoning, we will decompose the concept of QoS into its components. We will also discuss reliability and availability of guarantees on QoS.

Quality is deeply intertwined with the concept of perception. The experience of quality can differ from person to person. Quality can be perceived subjectively and objectively. A number of attempts have been made to capture the concept of Quality of Service.

The definition as given by the RACE QOSMIC project partly deals with the difficulty of specifying QoS at the end-user level by introducing the concept subjective and objective QoS [22]:

> *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.*

Obviously the objective QoS depends on the type of service or user.

Quality of Service can be viewed in more than one way: Quality can be *user perceived* quality and can be *machine measured*. For example, a service that goes down once a year, but is perceived as slow (due to stalls) we can *measure* a high level of availability, e.g. a web-server on the Internet. However an End-User may qualify the service as bad, since he has to wait a lot. If the same service goes down every two months, but does operates at a faster level, the End-User may *perceive* a higher quality than the latter, because the End-User does not experience a lot of waiting time.

So to describe the quality of a service, it should be clear what the End-User perceives as a QoS. Quality at the End-User level is not described with the same vocabulary as quality on the machine level, although they are closely related. We will go into more detail on this subject in the next chapter on guaranteeing QoS in an ODE. With the respect to the definition of QoS, we will use the RACE QOSMIC definition.

3.2 Decomposition of QoS

QoS is not a single entity: the quality of a service has many attributes, like the speed of service delivery, the correctness of service delivery and the costs of service delivery. The decomposition of QoS in an ODE can also take place at the abstraction levels from which we view a service. To give an idea we present an example that concerns a video-link between an office in Johannesburg and an office in Alaska. This link is provided between Video-Terminals over TCP/IP connections and satellite communications.

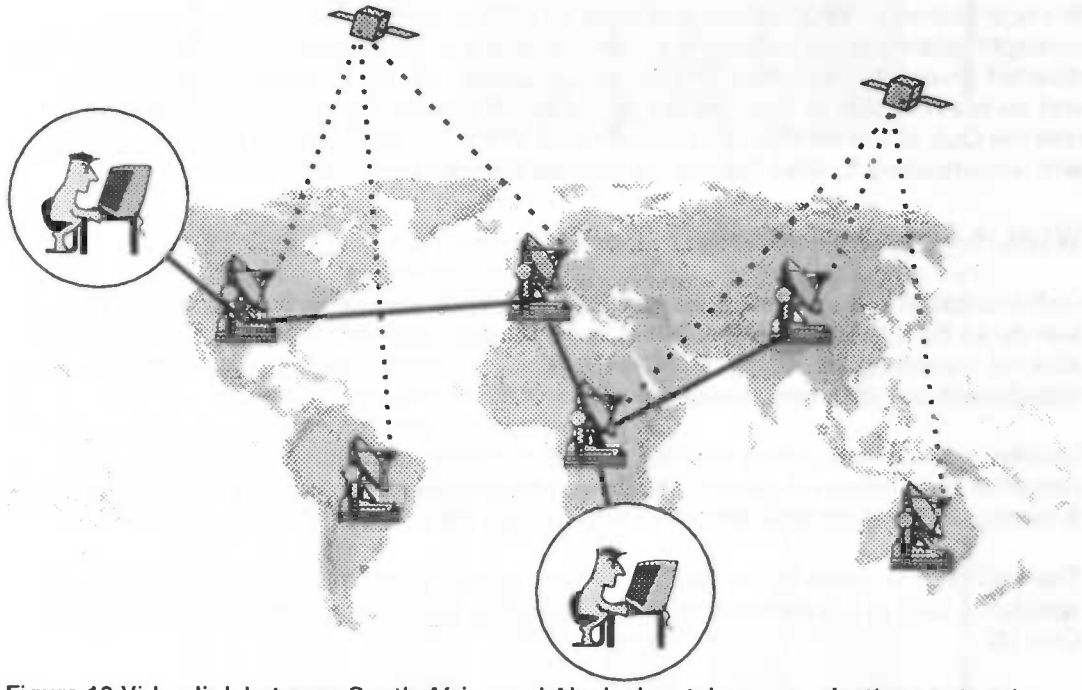


Figure 10 Video-link between South-Africa and Alaska in a telecommunications network

To guarantee a high level of QoS on the End-user level like "A stable and fluent video-link with our office in Alaska" we would need high QoS-levels of:

- a video-terminal
- a re-routable network
- a satellite link.

We have now decomposed the video-link into its physical components and see it at a lower abstraction level. We could now look at the QoS-levels of the separate components. When we use ODP-RM it becomes more clear: the Engineering Viewpoint is a further decomposition of the Computational Viewpoint *with respect to QoS*. In the Computational Viewpoint many QoS-attributes like transmission speed and bandwidth are not visible, because channels are not visible in this viewpoint. In the Engineering Viewpoint however we can see channels and their QoS-attributes. The QoS of objects in the Computational Viewpoint are determined by a composition of the QoS of objects in the Engineering Viewpoint.

QoS Abstraction level decomposition depends heavily on the type of system/architecture under consideration. Consider the video link example again: in the Computational Viewpoint we might see this:

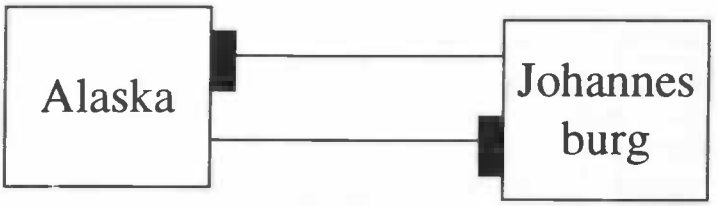


Figure 11 Video-Link in the Computational Viewpoint

There is a Video Terminal on the left in Alaska and there is a Video Terminal on the right in Johannesburg. The interfaces are the same and they might have QoS-parameters as:

- refresh-rate of the video image
- resolution of the displayed video image
- color depth of the video image

Note that this decomposition above in the Computational Viewpoint is a decomposition of QoS on the basis of attributes. In the Engineering Viewpoint we can see the channel between the two terminals. We have simplified the TCP/IP-satellite connection to a simple Stub-Binder-Protocol-Binder-Stub channel, but this does show the that in the Engineering Viewpoint we can also observe QoS-parameters that concern the channel like transmission speed and volume throughput:

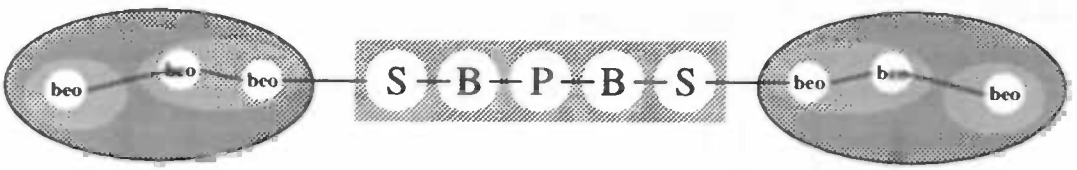


Figure 12 Video-Link in the Engineering Viewpoint

Literature on QoS tends to decompose QoS on the basis of attributes. Hutchinson et al. decompose QoS into major *categories* [10]. Every category contains so called *dimensions*, which are in fact measures that can be used to quantify a certain QoS attribute. We see a QoS category as a set of QoS attributes which are closely interrelated:

- *Timeliness*, contains dimensions relating to then end-to-end delay of control and media packets in a flow. Examples of such dimensions are latency, measured in milliseconds and defined as the time taken from the generation of a media frame to its eventual display, and jitter, also measured in milliseconds and defined as the variations in overall nominal latency suffered by individual packets on the same flow.
- *Volume*, contains dimensions that refer to the throughput of data in a flow. At the level of end-to-end flows, an appropriate QoS dimension may be video frames delivered per second.
- *Quality of Perception*, concerned with dimensions such as screen resolution or sound quality.
- *Logical Time*, concerned with the degree to which all nodes in a distributed system see the same events in an identical order.
- *Cost*, contains dimensions such as the rental cost of a network link per month, the cost of transmitting a single media frame in a flow, or the cost of a multiparty, multimedia conference call.

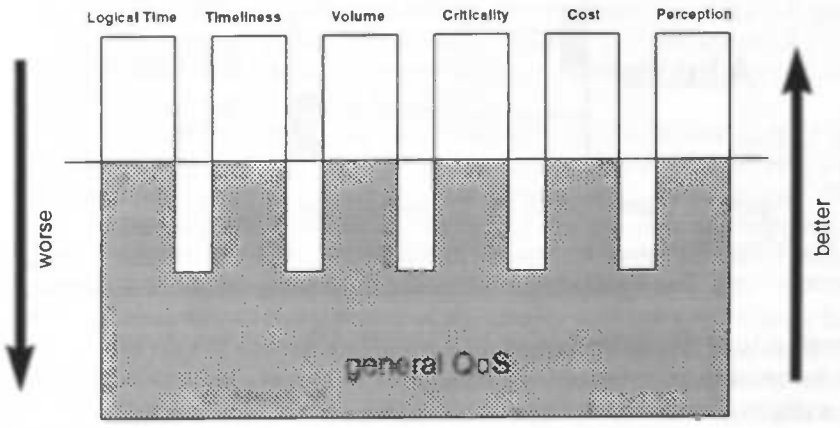


Figure 13 QoS levels of a service

Different QoS-attributes cannot be seen as independent attributes: increasing the level of separate QoS-attributes, while keeping the computational effort on the same level, results in a decrease of other QoS attributes. An metaphoric example is presented In Figure 13 and Figure 14. In the first figure we see the QoS-levels set a specific level. In the next figure, we keep the computational effort, with respect to resources and computing time, the same and increase the Volume and Perception category. This cause the Cost and Logical Time category to 'worsen'.

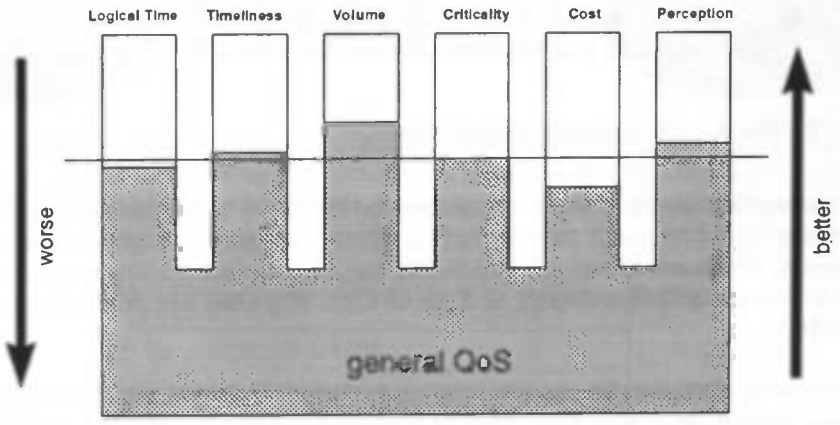


Figure 14 QoS levels of a service

We have followed Hutchinson et al. in their decomposition of QoS, but make a major exception: they model the *dependability* (the property to justifiably place reliance) of a service as a separate QoS attribute. We do not; in an ODE we can offer services whose (non-functional) behavior can be described by Quality of Service Level Agreements (QoSLA). They should be an exact specification (see also [16]), that is consistent, complete and authoritative, and can be applied as an effective test in all circumstances to determine whether the behavior of the system is accepted by the End-User. In this thesis we want to control the levels of QoS-attributes in general. Therefore the extent to which we can *depend* or *rely* on a guarantee is not a *quality* of a service, but a *property* of the ODE that controls the QoS-levels.

3.3 QoS Delivery

QoS can not only be perceived, it also must be delivered by a provider. We define three major categories (based on Franken [7]):

- **Guaranteed**: the required QoS must be guaranteed so that the requested QoS will be met, including "rare" events such as equipment failure. This implies that the service will not be initiated unless it can be maintained within the specified limits. In other

words, the resources that are needed to comply to a QoS are reserved for that service and not shared.

- **Compulsory**: the achieved QoS must be monitored by the service provider and the service will be aborted if it degrades below the compulsory level; The resources needed to comply to a QoS are not reserved, but shared. Compare this to the Dutch telephone system.
- **As soon as possible** or **best effort**: the weakest agreement. There is no guarantee that the required QoS will be provided. But the provider has an obligation to do the best he can.

Faults in the ODE and too much service requests may cause a provider not to provide the service with the requested QoS. We define faults, errors and failures with respect to QoS delivery (see Nieuwenhuis in [25]):

- A service *failure* is a deviation from a Quality of Service Level Agreement
- An *error* is that part of the system state which is liable to lead to a service failure.
- A *fault* is phenomenological cause of an error.

In other words, an error is the manifestation of a fault *in* the system and a service failure is the effect of an error *on the service*. We can observe recurrence in this definition; for example, consider a computer that provides computing services. The hard-disk controller of this computer breaks down (due to a fault) and causes virtual memory-problems (an error), the computer can no longer provide its computing services (a failure). However, if we isolate the hard-disk controller, we can see it as a system that offers 'controller'-services and the fault was in a chip somewhere on the card. But if we discover a chip on the controller that was responsible for the loss of 'controlling'-services, we can isolate the chip and say it offers computing facilities and...etc.

We propose the use of measures from the field of dependability to express *how long* or *when* the delivered QoS complied to the requested QoS. Again this does *not* mean that dependability is *a quality of a service*. We list three major (interrelated) categories of measure together with their metrics:

1. *Reliability*, which is a measure of the continuous delivery of a proper service at the request level of QoS from an initial reference point of time. A general notion is presented in [16]: a function $R(t)$ which expresses the probability that the system will conform to its specification throughout a period of duration t. However, the nature of this definition is such that $R(t)$ cannot be known for any system; at best the use of reliability modeling techniques will enable the form of $R(t)$ to be predicted and estimates made of the relevant parameters. A precise characterization of the *operational reliability* of a system can be given as a record of the occurrences of failure over a particular period of time, according to Littlewood[20].
   a) Rate of occurrence of failures (ROCOF): appropriate in a system which actively controls some potentially dangerous process, such as a chemical plant. (It has been used as the means of expressing the reliability goal for the certification of flight-critical civil avionics, and in particular the manufacturers of the A320 fly-by-wire system are on record as stating that the reliability requirement for this system was a ROCOF no greater than 10^{-9} failures per hour.)
   b) Probability of failure on demand, this might be suitable for a system which is only called upon to act when another system gets into a potentially unsafe condition. An example is an emergency shut-down system for a nuclear reactor. In the UK, the power industry rule of thumb is that such a probability of failure on demand can never be expected to better 10^{-5} for a system susceptible to failure resulting from design faults, particularly software faults.
   c) Probability of failure-free survival of mission. In circumstances where there is a 'natural' mission time, such as in certain military applications, it makes sense to ask for the probability of the system operating surviving

the mission without QoS-level failure. For example the most dangerous parts of a airplane flight are take-off and landing. If we would use failures/time as a metric for expressing the reliability of a flight, a longer flight would be safer than a short flight.

d) One could also focus on the (financial) effects of a failure: cost of failure, mentioned by Kant in [12], if a system has a certain ROCOF and the costs of failure are immense (nuclear power plant), then the ROCOF should be very low. Vice versa, if the ROCOF is relatively high, but the cost of failure is relatively low, then the system could still be considered dependable, since no catastrophes are happening.

2. *Availability* is a measure of the delivery of the proper service with respect to alternation of delivery of proper and improper service. This could be used in circumstances where the amount of loss incurred as a result of system failure depends on the length of time the system is unavailable. The combination of the next two metrics gives a fairly good estimation of how available the system is.
   a) mean time to repair (MTTR), which can also be viewed as the mean time between the last proper service delivery and starting proper service delivery again.
   b) mean time to failure (MTTF), which can also be viewed as the mean time to the next drop in proper service delivery, i.e. a service failure.
   c) mean time between failure (MTBF). Kant defines it in [12] as the MTTR+MTTF. Note a very high mean time of MTBF does not mean that the system is always operating at requested QoS-levels. If the MTTR is very high also, the system might only be QoSLA-compliant for a small amount of time.
3. With *Safety*, proper and improper service that do not cause catastrophes are combined; it is a measure of the continuos delivery of a non-catastrophic service.

These measures are obviously interdependent, and the selection of a particular one, or more, is a matter of judgment as said in [20]. In any case, these characteristics should accompany the settings of QoS-attributes in a Quality of Service Level Agreement.

Quality of Service Level Agreement
SearchTime <= 10 ms
• *reliability* = 50%
StorageCapacity = 4 Gb
• *availability* = 90%
SortingSpeed p/second = 1000 records
• *reliability* = 60%

Figure 15 QoSLA of a database service

In the example QoSLA in Figure 15 we can see that the search-time should be below 10 ms in 50 percent of all search-calls. In 90 percent of the time the database should be able to store 4 Gigabyte and in 60 percent of the sorting-calls the database should be able to sort 1000 records in 40ms. If the database complies to this QoSLA we can rely on it. We could ask ourselves why 100 percent reliability and availability are not requested? This is because of the fact a (near) 100 percent reliability or availability level is difficult to guarantee and not every End-User is interested in near 100 percent levels. Based on what they are willing to 'pay' or what they need, we propose to offer them different levels of reliability and availability.

In [12] Kant lists the following application domains that influence what types of reliability and availability are needed:

1. **high availability**, these systems are designed for transaction processing environments like banks, airlines, telephone databases, or switching systems. Also, data corruption or destruction is unacceptable.
2. **mission oriented**, these systems require extremely high levels of reliability over a short period, called the mission time. Little or no repair/tuning is possible during the mission.
3. **long-life**, systems that need long life without provision for manual diagnostics and repairs. There should also be considerable intelligence built in to do diagnostics and repair automatically. This category contains for example satellites.

3.4 Conclusion

In this chapter we have chosen a QoS definition and decomposed it on the basis of abstraction and on the basis of type categories (attributes and dimensions). In contrast to many other authors of QoS-related work, we have not defined *dependability* as a separate QoS, since it is an *property* of all QoS-levels. Instead we proposed the use of measures from the field of dependability for quantifying (failures in) QoS delivery.

4 Controlling QoS in an ODE

In the preceding chapters we described (using ODP-RM) Open Distributed Environments and Quality of Service. We now discuss controlling QoS in an ODE. Two conditions have to be met to control QoS: the first are abundant resources with smart resource allocation mechanisms. The second condition is that the system is tolerant for 'unexpected' events like break-down of components (e.g. a faulty processor, a broken connector).

ODEs can meet these conditions if configured properly and built redundantly enough. The first condition can be met by the combination of computing nodes in a network: resources can be shared and computing nodes can *interoperate* in parallel. And the second condition can be met by reconfiguring the ODE: if a link between two nodes goes down or a computing node crashes, connections can be re-routed and QoS-levels demands can be still be met.

In this chapter we discuss three issues concerning controlling QoS (in an ODE): measurement, guarantees and realization. We need to measure QoS levels in order to determine whether or not the service complies to the QoSLA. We need to know how to configure the ODE to deliver QoS. And we want to realize a mechanism that controls QoS by configuring the ODE.

4.1 Measurement

Measuring Quality of Service in an ODE can be done from different points of view. For example we can measure the QoS of a link by inspecting the throughput of bits per second, but we could also ask the End-User if the video-communication application is 'fast' enough (see also the ODP-RM viewpoints).

Measuring can also performed by different observers. Each observer could measure a different QoS. For, when two people judge a *complex* distributed application (i.e. with many components), they might both say that it is performing well and that it is offering a high QoS. But one person might refer to the way data is being transferred, while the other might refer to the way the application interacts with the user. This is an example of subjective qualification. In this case both people make a judgment without using the same criterion (i.e. the same QoS-attribute).

When the same QoS-attribute is used as a criterion, there still is a difference between objective and subjective judgment. If a processor is running on 155 MHz, two persons might judge it to be both 'good' as well as 'medium'. The first person is used to a processor running at 140 MHz and the other is used to a 170 MHz processor. They both use the same objective metric (MHz) concerning the same type of quality (processor speed measure), to express a subjective judgment. This is because every person has it's *own reference framework*. The moment they both use the same framework, they will reach the same conclusion. In other words they will make the same qualitative remark based on the same quantitative measure/metric.

This example clearly shows the difference between objectively and subjectively expressing quality. Therefore we define: the difference between objective and subjective QoS-measurement in complex systems is based on:

1. the QoS-attribute under consideration and the quantitative metric used
2. the reference framework (of judgment values) used by the perceiver (the End-User)

We also state that the more a system is viewed as a whole (by the End-User), the more subjective the judgment *tends to be*, since there is a higher probability that the perceived QoS is dominated by the QoS of a particular component of the system. We also define that the End-User is the provider over the reference framework

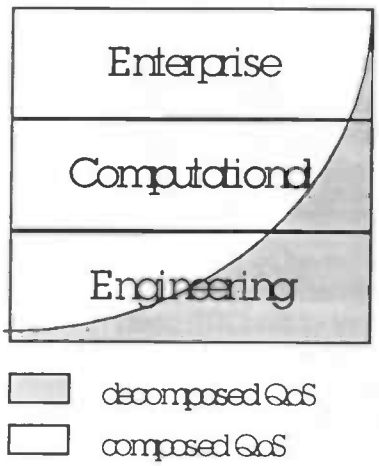


Figure 16 level of decomposition of QoS in ODP viewpoints

When we focus on QoS and we look at an ODE in the viewpoints Enterprise, Computational and Engineering (in that specific order), we can observe that an ODE is more and more dissected into its components. At each decomposition step it becomes more and more obvious which (combination of) QoS-attribute determines the by the End-User perceived QoS. This is because in a more detailed decomposition, components have more distinct QoS-attributes. For example, a distributed database system can provide fast data access as a QoS. This QoS can be decomposed into the QoS of a fast TCP/IP connection, a fast datastorage unit and a fast datastorage-manager, which all posses more distinct QoS attributes (see also Nankman in [24]). Figure 16 shows this decomposition with respect to QoS graphically.

In Engineering Viewpoint components have distinct QoS attributes. In the Computational Viewpoint the QoS of an object is determined by a combining the QoS of engineering objects. Therefor it becomes more difficult to make objective quantitative statements on the QoS of a computational object, because defining the algorithm or formula for combining the QoS of the engineering objects into the QoS of the computational objects, already involves subjectivity. The same goes for defining QoS in the Enterprise viewpoint.

Mapping between End-User QoS and system-configurations
The End-User in the real world states his QoS-demands with an Enterprise Viewpoint language which does not concern how a service should be provided (which mechanism should be used), but that it should be provided and what quality the service should. The provider of a service should offer the service and comply to the QoS demands. To know whether or not the service is delivered according to the requested QoS, the provider needs to measure the QoS. The provider can measure QoS of the devices in the ODE, which is Technological Viewpoint QoS information. The provider cannot directly measure how the QoS is perceived, which is Enterprise Viewpoint QoS information. We now develop a model which translates machine-measured QoS information from the technological viewpoint into user-perceived QoS information from the Enterprise Viewpoint.

4.1.1 Technological Viewpoint

In this viewpoint the system consists of physical hardware devices, interconnected by switches, routers and of course cables. The system has many separate clearly defined QoS attributes that can be measured:

1. *network* QoS levels: what is the packet loss, what is the transfer speed, what is the latency, what is the average jam-time, etc.
2. *processor* QoS levels: at what speed does the processor work, how heavily is the processor used, how available is the processor available, etc..
3. *memory* QoS levels: how fast can data be retrieved from memory, how much time is spent swapping memory from volatile storage to permanent storage
4. *storage* QoS levels, how many read/write errors occur, how many time-outs happen.
5. (graphical user) *interface* QoS levels: how good does the interface accurately mirror the state, is no false information given to the user, does the terminal produce no

garble (for example think of a X-window system, that crashes, the distributed system core could be functioning without any errors, but the interface is dead, so no communications can occur).

These are attributes that can be measured and expressed in objective terms.

4.1.2 Engineering Viewpoint

In this viewpoint the system consists of interconnected basic engineering objects, combined in clusters, that live in capsules, located on a node. An engineering object is a structural description of a mechanism, and can be mapped to devices that are visible in the Technological Viewpoint.

We define the that the QoS of an engineering object is determined by the QoS of the technological components it is mapped to. When a technological component does no longer provide the required QoS, a QoS-controlling component of the ODE could re-map the engineering object to another technological component. An example of this is using a backup hard-disk in case of disk-failure: the desired QoS-levels can remain at the demanded level, but the technological device does change. Note that although the speed of the exchange largely determines how the End-User perceives the temporary drop in the QoS. If the hard-disk failure is detected fast enough and the switch takes place within several milliseconds, the End-User will probably not perceive this temporarily drop in QoS and will thus perceive the QoS as reliable and available.

4.1.3 Computational Viewpoint

In this viewpoint the system is described in terms of objects interoperating with each other. Objects interact with each other without using channels and the location of objects is not visible. The QoS-attributes are therefor less technological oriented. There are no such QoS-attributes as packet-loss or network delay, but attributes like access-time to data, response-time of a service-providing object or quality of video-images.

QoS-attributes like response-time can be deduced from the Engineering Viewpoint description in which every Computational Object is decomposed into engineering objects. For example, the average response-time of a database-object can be calculated from the network delay, the processor speed and the storage speed of the mechanisms that provide a database in the Engineering Viewpoint.

4.1.4 Information Viewpoint

In this viewpoint the Quality of Service Level Agreements are visible since they contain the information which a QoS controlling component of the ODE needs for establishing which service is to be provided with what QoS. Such an information description of the system is concerned with defining the type and structure of information and it will therefore cover requirements for precision which would typically be derived from QoS statements in other viewpoints (for example, color and resolution of video.

The QoS at which information is transferred or maintained is not described in this viewpoint: these are QoS-attributes of Computational and Engineering objects.

4.1.5 Enterprise Viewpoint

In this viewpoint there is an IT-system that provides certain services at different quality of service levels. A service is accompanied by a collection of desired settings of QoS-attribute levels. The QoS of a service is determined from the QoS of the computational objects. With respect to the personal judgment framework of the End-User we propose

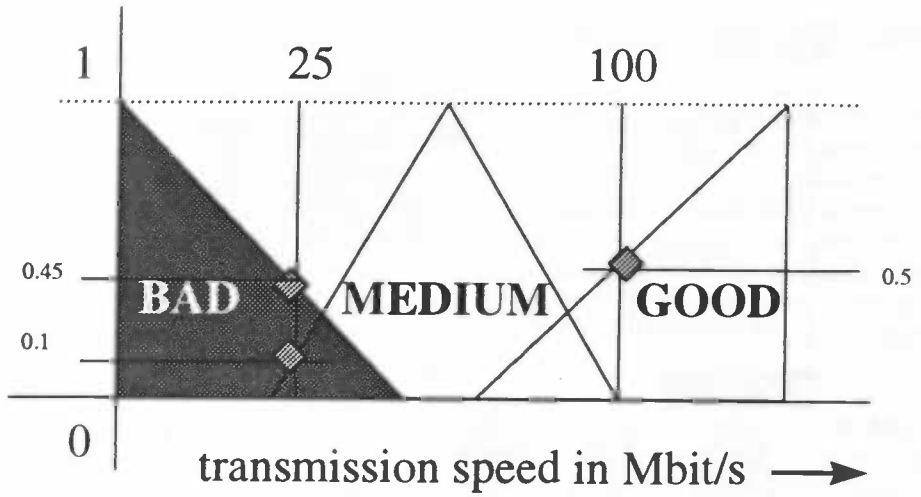


Figure 17 Translation of numeric values concerning transmission speed into verbal values

the use of Fuzzy Sets to translate numeric values into verbal quality statements. We will not provide an introduction to Fuzzy Sets (see BOBO), but do provide an example in Figure 17. On the horizontal ax we can observe the transmission speed. On the vertical ax we can observe the values from 0 to 1. There are three QoS areas. The 'BAD', the 'MEDIUM' and the 'GOOD' QoS area. The vertical coordinate (the height) of a point on a slope of the areas determines to what extent the horizontal coordinate (the transmission speed) belongs to the area (or set in mathematical terms). For example, a transmission speed of 100 Mbit/s belongs for about 0.5, i.e. 50% to the area of 'GOOD' QoS. Since it does not belong to the 'BAD' or 'MEDIUM' sets, 100 Mbit/s is experienced as a 'GOOD' QoS. If we look at 25 Mbit/s we can see that it belongs for 10% to the 'MEDIUM' QoS set and for 45% to the 'BAD' set. In other words 'rather bad and barely medium'. A transmission speed that is both considered as bad and as medium may seem awkward, but in the real world people can also often not define a precise border between good and bad QoS. A precise border between the sets can be created by removing overlap between the areas.

This example shows how the personal reference frame of an End-User can be inserted into the measurement model: by determining the levels of QoS (i.e. good, medium, worse, bad, etc.) and setting the slopes of the lines that contain the areas the machine-measured QoS information can be transformed into verbal End-User QoS statements.

When we add the descriptions of QoS in the different viewpoints we can observe the

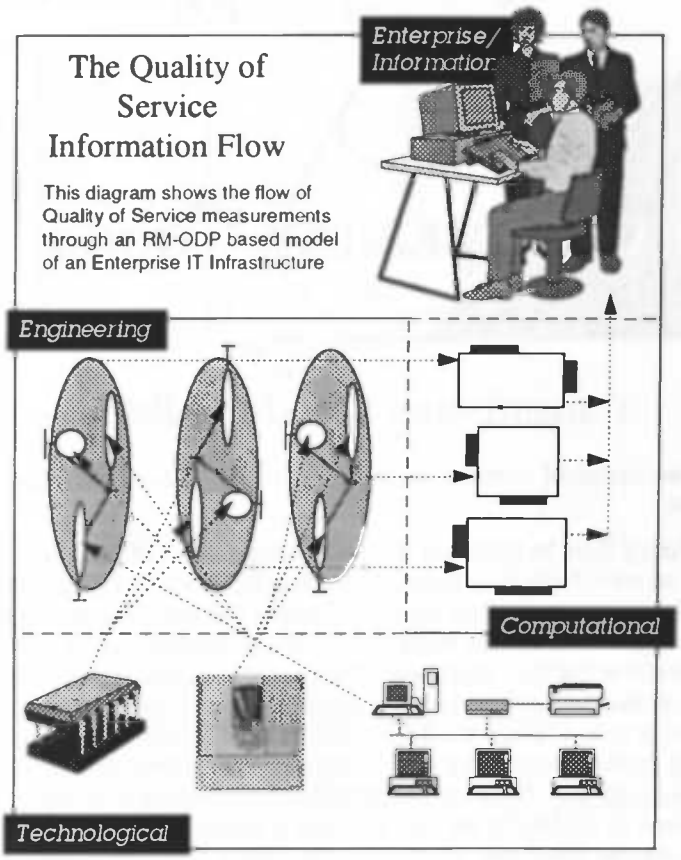


Figure 18 Quality of Service information flow

following pattern emerge: the QoS of technological devices is measured and aggregated (per capsule or per node) This information is then again aggregated (per computational object). Finally the QoS of the computational objects is used to evaluate the QoS the End-User perceives. In Figure 18 the flow of QoS-attribute information from abstraction layer to abstraction layer is visualized. It can be seen that the information is in fact collected in a tree-like fashion: at each abstraction layer, the QoS of objects is determined and this information is aggregated to determine the QoS of objects at a higher abstraction layer. The leaves are the QoS of the services provided by technological objects. The next levels of branches is a formed by the engineering and computational objects. And finally the root is the service.

Conclusion
In this section we have shown how we look at QoS information in the ODP viewpoints. Based on the way information is collected and aggregated we now conclude that it is important to know the relationship between the actual service-providing technological objects (computers, links, etc.) and the service in the Enterprise level. When we know this relationship we can determine which objects in the ODE comply to the requested QoS and which do not. By collecting and storing the information in a tree-like fashion we can deduce this relationship.

4.2 Configuring

We are now able to measure the QoS from the technological devices and translate it to verbal End-User QoS statements. We can now determine how much the delivered service deviates from the requested QoS. But this is only a part of what we want. We also want to know how to configure the ODE in order to provide the requested QoS. If we do not have the possibility to configure the ODE in such a way, we cannot guarantee the QoS. Intuitively we would now design a model which would compute an ODE

configuration based on a requested QoS. We will not do so, because we think controlling QoS in an ODE, based on a ODE configuration model, is not a solution.

To see why, we first show the similarity between *Markov* models and QoS behavior models. *Markov* models give an estimate on the probability that an entire system is operational or not, based on the stochastical information on the operational behavior of its components. In the previous chapter we have shown the QoS of a service depends on the QoS of the underlying (ODE-)components. We now adapt the question "is the component operational?" to "is the component operating at the requested QoS level". It now models the probability of a system providing service according to the requested QoS. This is a stronger demand, it not only implies components are operational, it also implies components deliver the requested QoS.

We now turn to Friedman and Voas [8], who state that the problem with Markov models is that we have to show independence between the components in order to apply them. In an Open Distributed Environment this is not easy, not to say impossible. When we use the ODP-RM viewpoints we can observe that services consist of computational objects, which in their turn consist out of engineering objects. And these engineering objects might be located at the same nodes in the network and might use the same connections. If Markov models have problems 'reflecting reality', the QoS-behavior models will certainly have those problems also. This makes constructing a Markov-like model rather complicated, not to say impossible.

To show how complex the situation actually is we provide an example. In Figure 19 we can see a ODE configuration with two services: the 5 pointed star service and the multi pointed star service. The objects that provide the service are redundantly available, for service there are identical copies of a servers at both site A and site B. The services are located on a node, a circle in the figure. The nodes are linked by lines and hubs. This has direct implications on the QoS. There is a major difference between delivering QoS to a client (or End-User) at site C and to a client at site D. QoS-attributes of the 5-point-star service at sites A and B might look the same in the computational viewpoint, in the engineering viewpoint we can immediately see the QoS is delivered by other components to a client at site C than to a client at site D: site C is connected by a single link, site D is connection by many more links and nodes.

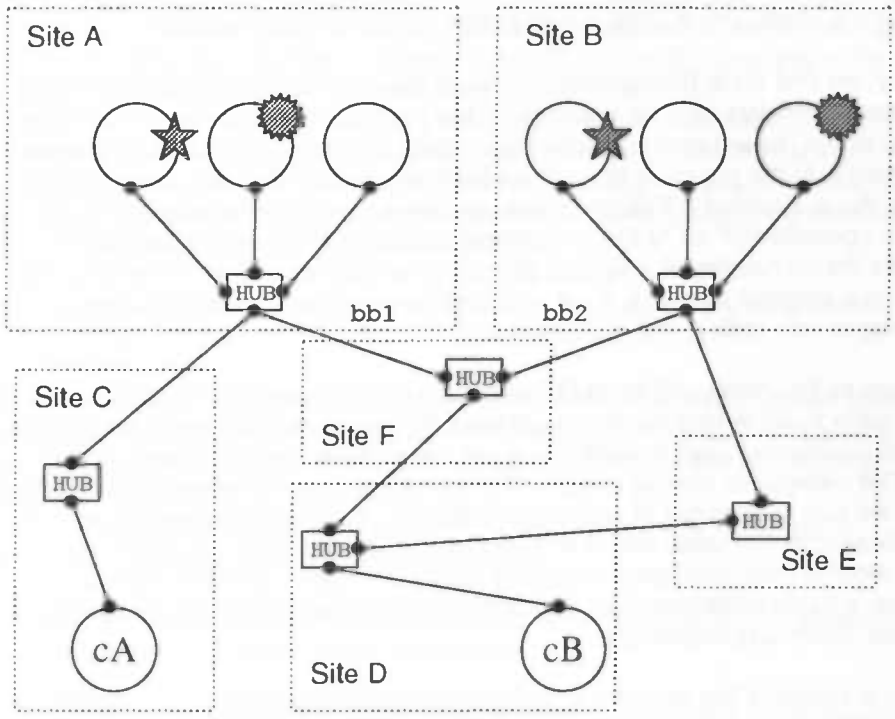


Figure 19 Layout of a possible client-replicated server connection

This example shows that the reliability and availability of QoS can often only be computed per client and becomes a complicated task when many configurations (routations and allocations) are possible.

In the case of such complicated configuration all information concerning the network and the location of clients and servers could be incorporated into a model, but as Friedman and Voas say in [8] with respect to (only!) faults, such models require an integrated hardware/software testability theory, which to their knowledge does not exist. Even if such models would exist, we would have to collect statistical data about power-loss, the amount of cable-breaches in a certain environment, stochastical usage distributions, because ODEs are often geographically distributed and used by many End-Users (at randomly distributed times).

Another example of designing models of (complex) software is the difficulty of quantifying the probability of failure of software. Practical methods to prove the non-existence of design errors, or methods to predict the change of an fault occurring have not been developed yet. In [4] Butler and Finelli explore the inherent difficulty of accurately modeling software reliability. They show that it is not reasonable to apply the classical hardware models for guaranteeing ultra-reliability (< 10^{-7} failures per hour) for modeling software and hardware design flaws. If it is difficult to predict the reliability of software based on a model, predicting whether or not software will deliver the requested QoS in a highly dynamic environment on the basis of a configuration-model will also prove to be difficult.

Conclusion

The inability of Markov-like models and the difficulty of for example quantifying the probability of software-failure provides evidence that controlling QoS in a complex environment like an ODE should not be performed on the basis of a configuration model. A realistic algorithm that deduces ODE parameter-settings on the basis of a requested QoS in a QoSLA would not be useful because of:

1. the amount of parameters that can be set in the configuration (e.g. all parameters at the host and at the network) make the computation of a appropriate configuration lengthy and time-consuming.

2. because of the interdependence of software components, the computation of configurations will pose some mathematical problems which require complex solutions.
3. an ODE is so dynamical that the time to compute a new configuration would probably make the outcome of the computation outdated.
4. because resource allocations in an ODE continuously change configurations become rapidly outdated and new configurations have to be computed, which make the ODE continuously reconfiguring itself.
5. in the case of software failure the current models can not carefully predicted the occurrence of failure, which makes a configuration vulnerable.

Therefor we propose not configure the ODE on the basis of a configuration model, but on the basis of 'a need for adjustment'. This means that a service is initiated with parameters set based on an estimates that will probably deliver the requested QoS. The estimates can be determined on the basis of previous usage statistics. If the perceived QoS (as measured by the model in the previous section) meets the requested QoS in the QoSLA, nothing is changed. As soon as the perceived QoS does no longer meet the requested QoS, the should adjust the ODE not by computing a complete new configuration, but by adapting. For example adding extra ODE elements that increase the QoS (which of course means computing a part of the configuration), exploits the available redundancy and makes it possible to share computing resources.

4.3 Realizing

Enabling the system to adapt on changes in the ODE which threaten the requested QoS, can be done in several ways: by supplying abundant resources and making systems robust, so component failure will not directly endanger QoS.

4.3.1 Abundant resources

Supplying a system with more resources than the total amount of End-Users (probably) will use is a possible solution for reliably guaranteeing QoS-levels. A good example of this is the Dutch telephone system. If one takes the phone of the hook, one almost always hears a dial-tone. In general all End-Users of the phone-system are satisfied, when they need to phone, there is connection at the guaranteed level of QoS. However, if all End-Users would pick up the phone at the same time (i.e. millions of End-Users), then the telephone exchange would be overloaded and some End-Users would not hear a dial-tone. However, unless all End-Users agree on a malicious attack, the system has been designed in such a way this situation only occurs with very low probability. The design is based on stochastic usage distributions. In this case, there are (stochastically seen) abundant resources because of the switching technology used in telephone exchanges.

Another example is adding more computational power to a system when needed. Consider again the distributed database example. This database guarantees a minimal access-time of 10 ms as a QoS-attribute for a searching-service. When the amount of End-Users of the search-service becomes so high that the requests cannot be handled within the 10 ms period, extra search-engines working in parallel at other hosts in the ODE could be added, based on the number of End-Users.

4.3.2 Robustness

Service failure due to not complying to desired QoS-levels can also be caused by component loss. In this thesis, we consider component loss to be caused by faults in hardware and software. Faults can be separated into several categories [25]:

1. *Physical* faults, which are introduced by physical phenomena, either inside or outside the components of the system. If the system is decomposed in a hardware and a software component then physical faults will not occur in software. Within this category there is a distinction between:

a) *Internal* causes, for example a CPU that is built out of contaminated silicium which results in faults, or for example a loose connector inside a computer.
b) *External* cause, like loss of power, a breach in one of the cables of the network due to construction on the road, magnetic solar storms.

2. *Human-made* faults, which are either *design-flaw* generated faults or *interaction* generated faults. The design-flaw generated faults are introduced during the design or during modifications. Violation of operation or maintenance procedure can be considered as interaction generated faults. These faults can occur both in hardware and in software.

And there is a second categorization of faults, that is important for time-sensitive QoS-level guarantees:

- *Transient* faults are faults that are present in a system for only a limited period of time after which it disappears spontaneously from the system. Note that however in a OO distributed system, a transient fault within an object may result in a wave of other transient faults within other objects.
- *Permanent* faults are faults that remain in a system until it is repaired
- *Intermittent* faults are recurring transient faults.

We propose to include a separation of faults on the basis of the responsibility of the underlying DPE and the applications in the DAE:

1. *Distributed Processing Environment* generated faults, these are caused by the underlying infrastructure (the DPE) and cannot be influenced by the distributed application that is running in the DAE. That is, no matter how correct and safe the DAE is, these faults can still occur and are the responsibility of the DPE.
2. *Distributed Application* generated faults, these are caused by logical design errors in the distributed application software itself (i.e. application errors). They cannot be influenced by the designer(s) of the DPE and only occur in software.

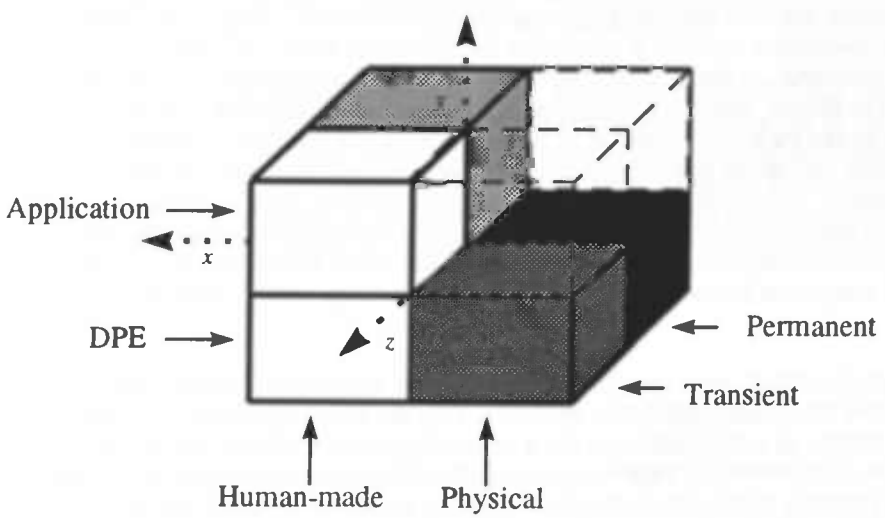


Figure 20 Relationship between fault-categories

In Figure 20 we can see the relationship between these categories. We have modeled the set (or space) of all possible faults as a cube. The origin the coordinates in this cube space is in the middle of the cube. Every fault to the left of the plane x=0 belongs to the Human-Made faults. Every fault above the plane y=0 belongs to the distributed application generated faults. Note that a physical application fault does *not* exist. Every fault beyond z=0 belongs to the permanent faults. We have included the intermittent faults in the transient fault space. Note that since this we do not quantify faults, the axes cannot serve as a measure.

Dealing with faults
The fault-cube shows and the sub-cube 'generated physical faults' shows the vulnerability of an ODE to externally generated faults. To prevent faults resulting in errors, which result in failure to comply to QoSLAs, there are several techniques:

1. *fault-avoidance*, One can use formal methods and design verification techniques aimed at ensuring the absence of faults, both in hardware and software.
2. *fault-tolerance,* One can also assume that certain types of faults (which result in a drop of QoS and is a failure) do occur and develop strategies to cope with them. This is know as tolerating faults by design. We present some techniques to provide such robustness (in software), as found in [30],[25],[16]:
   a) *Backward Error Recovery* , is based on the idea of checkpointing: establish a checkpoint, where the current system *state* (i.e. the collection of parameter settings that is essential for the system to continue further *correct* operation) is 'frozen' and 'stored', perform some actions and if a fault occurs, go back to the last checkpoint and take the necessary actions to avoid generating the same fault. There are some variations to this theme, like Transaction Processing (where a set of operations can be made atomic) and Recovery Block Scheme from Horning that was developed in 1974.
   b) N-modular redundancy, encompassed the idea of having separately located copies of a module in the network available. In case of a crash of one of the copies (or replicas), the others still continue and for most QoS-attributes no drop in service-level is noticed by the End-User. There are again variations on this them like pure replication where the copies are exactly the same and N-version programming where the only interfaces of the modules are the same. The 'insides' have been programmed and designed by different teams of programmers, which should result in a lower probability of total error. However the improvements are less than expected presumably because design faults of independent teams are not statistically independent [25].

4.3.3 Preferred mechanism

From the section 4.3.1 and section 4.3.2 we conclude that in both cases redundancy is a preferable means to deliver the requested QoS. This is in accordance with the statements we made about the power of an ODE being in the redundancy and flexibility. This makes *N*-modular redundancy (with every component/object on a separate machine) a logical choice:

1. we can provide extra computation power on a need-to-compute-basis; for example in the case of the database search-time example, we can add an extra search-replica at another host in the distributed database.
2. we are protected from total service-failure by a computers that crash, since more than one copy of a service providing component exists on several hosts. The probability of all hosts crashing at the same time is far less that the probability that a single host crashes.
3. we can enhance other QoS-attributes like correctness QoS-levels, by adding a voting-mechanism to replica groups to satisfy *correctness* QoS-levels.

If each replica object is implemented differently by only replicating the interface, we might increase the use of redundancy: different implementations might process a request in a faster or more efficient way, depending on the situation. This would result in higher levels for the QoS concerning speed, access-time, etc. We call this *extended* replication.

Redundancy can be provided by replicating hardware components and software that makes use of replicated software-components (replica-objects). When a computer has replicated hardware components, it can still operate while some of the replicated components have broken down. Software replication means that service-providing software components are located on different machines, which allows break-down of a machine while the service continuos to be provided. When to use hardware replication

and when to use software replication, depends on the situation where reliable guarantees are needed. We now present a comparison between hardware and software replication in the following table.

Comparing replication	in hardware	in software
purchase costs	high, because the hardware consists of a (single) machine with hot-swap-able components.	low, because cheap networks of interlinked PCs can be used.
maintenance costs	high, because special equipment requires special personnel	low, because standard components requires standard personnel
computational speed	fast, because hot-swap-able components run in parallel 'without' much communicational overhead	slow, because software protocol require much communication across 'slow' lines.
software flexibility	software can be run only on this machine	software can be recompiled to use on a different machine
ease of development	normal	difficult

With ease of development we mean, the amount of effort the programmer has to do in order to build applications. Later on in this thesis, the reader will see why. We do state that software replication provides a relatively cheap way of exploiting the redundancy that is available in the ODE. Hardware replication does not exploit this redundancy.

Use of replication techniques in an ODE results gives rise to the use of a special metric, which concerns the level of fault (or failure) -transparency in an ODE. We name this metric the fault-transparency *capacity*, and define it as:

$$\text{Failure Per Fault} = \frac{DAE\ Service\ Failures}{DPE\ Faults}$$

It is defined as the ratio of the faults that occur in components within the DPE that are responsible for providing the DPE services and the failures that occur at the application service level. A low FPF means that the fault-transparency (or Failure-Transparency in ODP-RM) layer is performing well: Although there several hosts crash or links break-down, the distributed application does not have to suffer from these faults. A high FPF means that almost every fault in the ODE, results in a failure at the application service level.

4.3.4 QoS behavior

Until now we have not provided much detail on the behavior of the system in the case of an event that threatens the delivery of the requested QoS. We now state there are three major categories of system behavior with respect to such an event. These three categories are closely related to the levels of fault-tolerant behavior [25]:

- *Persistent QoS-delivery*, the system continues to deliver QoS at the guaranteed level as described in the QoSLA in the presence of (unexpected) events (like faults) that affect resources or components that are responsible for the demanded QoS. For example: because too much bandwidth is demanded from an ATM-connection, the system automatically provides a second connection, so no drop in QoS-level is experienced.
- *Graceful QoS degradation*, the system continues to operate in the presence of events that affect resources or components that are responsible for the demanded QoS, but there is a partial degradation of functionality and QoS. For example: because too much bandwidth is demanded from an ATM-connection, transmission requests are routed across a secondary TCP/IP connection and End-Users experience a drop in transmission QoS.
- *Paused QoS-delivery*, the system keeps delivering the demanded QoS, but after a temporary halt in its operations.

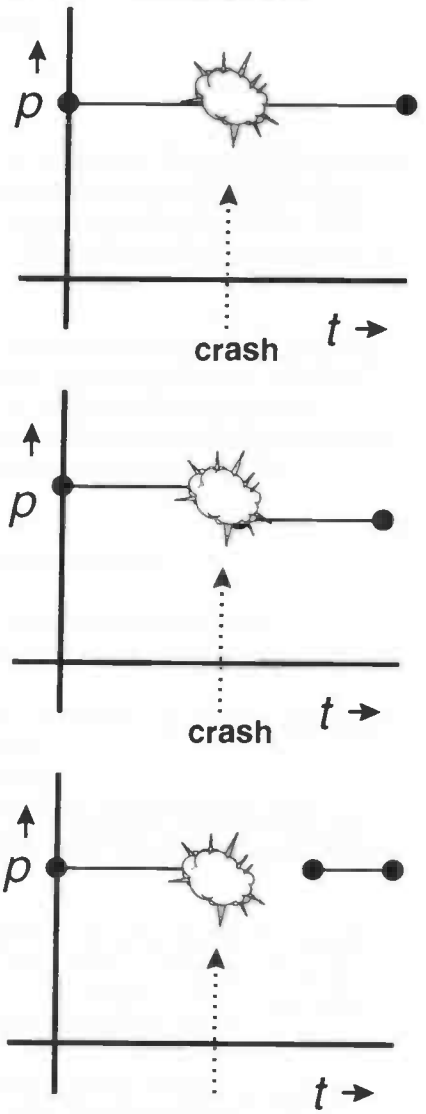


Figure 21 QoS delivery behavior of a service

In Figure 21 we can see the behavior model as a graph of the performance of a system through time. In the case of persistent QoS-delivery, the performance (p) does not change at all. In the case of graceful degradation the performance drops to a lower level, etc.

Redundancy can ensure both persistent QoS-delivery and graceful QoS-delivery. The first type of delivery can be provided, because a service that is based on replicated components is not directly sensitive to the crash of a computer in the network and extra computer power can be provided when needed. The second type of delivery will occur, when no more replicas can be added to compensate or when the use of redundancy does not solve the problems in meeting the QoS demands. For example, how redundant a connection may be, the speed of the serial transfer of data on an ATM-connection has a maximum speed.

4.4 Conclusions

In this chapter we provided a model of measuring QoS in an ODE based on viewpoints. We have shown that mathematical models of ODE configurations, that compute parameter settings of a configuration based on requested QoS, are complicated and difficult to compute. Furthermore, configurations that can be computed are quickly outdated, because of changes in resource allocation due to the dynamics of an ODE. This would result in an unstable ODE that is continuously reconfiguring itself. Therefor we propose to use a mechanism that adjusts the ODE partially 'on demand'. This

mechanism will be based on replication, since it exploits abundant resources and makes an ODE robust (e.g. protection against faults).

5 Replication based on virtual synchrony

In the previous chapters we have defined the concept of QoS in an ODE and proposed a mechanism for measuring. We also suggested replication as a means for controlling QoS in an ODE. In this chapter we show the details of software replication in an ODE. Discussing the theory behind software replication in an ODE will enable us to determine the impact of replication on QoS.

The replication-mechanism we are going to discuss is based on message-ordering. By ordering the messages (the requests and replies) between client-objects and service providing objects in an ODE we can assure correct behavior of replicated service-providing objects. The protocol (or mechanism) which guarantees the ordering is called *virtual synchrony*. It has been developed by Birman et alius and because of its complexity it will be defined in section 5.2.2.

The replication-mechanism should comply to the following demands:

- the computational model of the application must be replication transparent;
- the engineering model based on replication must be generic, i.e. independent of the application under development;
- the engineering model must not include a single point of failure, i.e. failure of a single engineering object may not cause overall system failure;
- dynamic creation of a replicated object must be possible;
- automatic dynamic complying to a service level agreement should be possible, i.e. maintain a certain size of a group of replicated objects under the presence of crashing group members.

The last demand means that if a replicated object disappears (e.g. because of a crash) and the situation demands that a new replicated object should be constructed in order to comply to the requested QoS, a new replica-object should be created. Also when more computational resources are needed, a new replica-objects should be created.

We will discuss the replication-mechanism by first defining (replica-)object groups and then state the (replication) group membership problem. This *communication* and *synchronization* related problem makes it difficult to guarantee the *correctness* of an application when components are replicated. After presenting the problem we will provide a possible solution. At the end of this chapter we deduce, based on the properties of virtual synchrony, what the impact of our replication-mechanism is on QoS.

5.1 Object groups

We start with defining object groups and related concepts like the membership of object groups. We define a set of objects with the same computational *state* and *behavior* as a object *group*. Every object in this group is called a *member* of the group. The computational *state* is defined as the set of variables (or attributes) which determine the behavior of the objects when it receives requests. In other words:

- The request can be chosen from the entire spectrum of all possible requests according to the specification of the interfaces of the objects.
- Two objects have the same state when both display the *same behavior* after receiving the same request with the same parameter settings.

- After the displaying the exact same behavior, which we define as a *state transition*, the objects should have the exact same state again.

We adopt a dynamic view, which means that objects can join a group and leave a group. The size of a group can vary dynamically while objects are handling requests. After joining a group the joining object receives the same state through a state transfer, i.e. a state transition occurs in the joining object. This view is called the *dynamic membership model*. Leaving a group can be by normal termination or by failure (like a cable connector disconnection).

One of the most important aspects concerning object groups, is state synchronization. If the states of all objects in the group are not the same, i.e. not synchronized, they are no longer replicas. Their behavior on receiving a request may substantially differ, which causes unspecified and probably unwanted effects. Synchronization is complicated due to the fact that replicated objects in an ODE will probably reside on different hosts. The delay between sending and receiving messages may cause havoc when synchronizing state. To see why, imagine a process group consisting out of server objects A,B and C (this is shown in Figure 22). We will use a naïve way of communication, i.e. each client sends a copy of the request to each replicated object A, B and C. There are two clients. The first one sends a request m1 to the group and the second one sends request m_2. Without further measures, messages may arrive in different order. In this case the request m_1 arrives at a later point in time at C than m_2 does. If we assume that a request results in a state transfer, it is easily observed that the order m_1,m_2 and m_2,m_1 results in different states of server-objects B and C respectively.

A possible solution would be to use a timestamp in every request, so the receiving servers can order the messages.

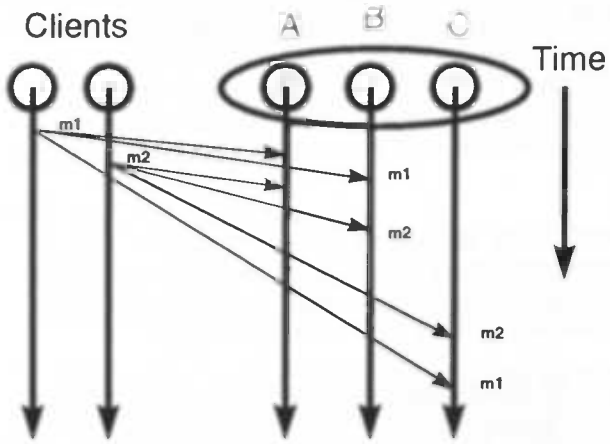


Figure 22 Communication between clients and a process group

However, the clocks in every host have limited accuracy and can drift. Therefor time in distributed systems concerning process groups is an issue. Furthermore, an object can crash (e.g. because his host does). It is easily observed that this gives rise to yet another spectrum of problems during state synchronization. These problems are well know in literature (interactive consistency, Byzantine Agreement, Distributed Clock Synchronization, etc.).

Solutions for the synchronization problem(s) above have been developed for different failure models concerning the transmission of messages and the crashes of objects. We use 'message-passing' as the mechanism for sending and handling requests:

- *Crash* failures. Within this model it is assumed that a crashed object remains silent forever. It does not produce any 'faulty' messages. So messages sent by this object are always *correct*. If the object fails by not sending a message when it should, it will never send a message again.

- *Omission* failures. Within the omission failure model it is assumed that objects can and will omit messages. So messages sent by the objects are not always correct.
- *Malicious (or Byzantine) failures.* Within this model (almost) nothing is assumed about the objects. Objects may send any (in)correct message. They may be malicious in the sense that they may "conspire in order to achieve overall system failure".

We will discuss a solution for the crash-failure model: a set of objects cooperates by exchanging messages over communication links that are arbitrarily slow and balky. The assumption here is that the messages sent on the links eventually get through (in theory the amount of time can be infinite), but that there is no meaningful way to measure progress except by the reception of messages. In this model we are pessimistic about the ability to measure time or to predict the amount of time actions will take. We use this model because in real distributed computing systems there is a lack of closely synchronizing clocks. Furthermore the systems are unable to distinguish network partitioning failures from host failures.

When this *purely* 'timeless' computing model is used, a classical result limits all possible solutions. In 1985, Fisher, Lynch and Paterson proved that the (asynchronous) consensus problem (similar to the Byzantine problem, but now posed in an asynchronous setting) is impossible if even a single process can fail. They have demonstrated that any protocol guaranteed to produce only correct outcomes in an (asynchronous) system can be indefinitely delayed by a complex pattern of network partitioning failures. Although practitioners commonly conclude that every day scenarios in distributed systems do not correspond to the scenarios in the FLP-proof, it is therefor common to make use of time (as in time-outs or polling).

5.2 Group Membership Problem

If the ordering of the messages remains the same, a crash of an object between sending message m_1 and m_2 can result in incorrect behavior. This is because when one member of a group does not receive a message that the other members do receive, it is no longer guaranteed that all states are synchronized. We can deduce that it is important to know which members are still in the group and which members have failed, to prevent loss of state synchrony. This problem is also known as the *Group Membership Problem.*

In [2] Birman proposes the use of a so called group membership service (GMS), which would be a possible solution for the GMP. Although Birman reasons about processes and we reason about objects, the GMS-theory still applies to objects. He adopts a model in which processes wishing to join do so by first contacting the GMS, which updates the list of system members and then grants the request. Once admitted to the system, a process may interact with other system members. If the process terminates, the GMS will update the list of members and can notify (if needed) the other members. The GMS problem (or a model of such a service that provides enough detail to implement it) is difficult to formalize. Birman says "*there exists an attempt to formalize the GMS problem, and there are some known and serious problems with the proposed formalization. However, the proposed protocols are generally recognized to work correctly, and the concerns that have been raised revolve around relatively subtle theoretical issues having to do with the way the formal specifications of the problem is expressed in temporal logic*".

Operation	Function	Failure Handling
join (process-ID, callback) returns (time, GMS list)	Calling process is added to membership list of system, returns logical time of the join event and a list giving the membership of the GMs service. The callback function is invoked whenever the core membership of the GMS changes.	Idempotent. can be reissued to any GMS process with same outcome
leave(process-ID) returns void	Can be issued by any member of the system. GMS drops the specified process from the membership list and issues notification to all members of the system. If the process in question is really operational, it must rejoin under a new process-ID.	Idempotent: fails only if the GMS process that was the target is dropped from the GMS membership list
monitor(process-ID, callback) returns callback-ID	Can be issued by any member of the system. GMS registers a callback and will invoke callback (process-ID) later if the designated process fails.	Idempotent: as for leave

Figure 23 Table with GMS Operations

In Figure 23 we a table shows which operations or services a GMS should offer to replica-objects. The GMS will need to be highly available. Hence it will probably be implemented by a set of processes (on physically separated computers) that cooperate to provide the GMS abstraction. At first sight it seems that the same group membership problem now reoccurs, i.e. a GMS server needs to solve the GMS problem on its own behalf. To do so, it uses a *group membership protocol* (GMP). In short, the GMP is responsible for the membership of a small service, while the GMS is responsible for the rest of the groups.

5.2.1 Group Membership Service

The protocol that we are now going to describe is based on the one that was developed as part of the Isis system in 1987, but was substantially extended by Ricciardi in 1991.

The GMS is implemented in such a way that all operations that are invoked use a modified RPC protocol. Processes are allowed to issue requests to any member of the GMS server group with which it is able to establish contact. The join operations are idempotent. If a joining processes times out or otherwise fails to receive a reply, it can reissue its request, perhaps to a different server. Clients that detect apparent failure, report them to the GMS. The GMS is responsible for failure notification. Actions that would normally be triggered by time-outs (such as reissuing an RPC or breaking a stream connection) are triggered in this protocol by a GMS callback notifying the process doing the RPC or maintaining the stream that the party it is contacting has failed.

Costs

We will now try to establish an estimate on the complexity of the GMS (and later on of the communication protocols which make use of the GMS), in order to be able to determine the costs of the GMS and the proposed communication protocols. These costs can be measured:

1. in terms of the latency before the delivery of a message;
2. the amount of messages that each individual has to process (message load);
3. the number of messages placed on the network as a function of group size;
4. the overhead required to represent protocol-specific headers.

We will be expressing the complexity and costs using mostly the third criterion.

We will only cover those details of the GMS which are important for an estimate on the complexity of the algorithms that are used, for details see Birman in [2]. The first part of estimate will be on the protocol to track the core membership of the GMS service itself, i.e. the processes responsible for implementing the GMS abstraction, but not their clients. The assumption is that processes watch each other using a network-level ping operation and detect failures by time-out. Apparently failed members are handled by the GMS coordinator, which is the GMS member that has been operational for the longest period of time. If a process believes the GMS coordinator has failed, it treats the next highest ranked process (perhaps itself) as the new coordinator.

A process that is suspected of having failed is *shunned* by system members that learn of the system failure.

Upon detection of an apparent failure, a GMS process does the following:

1. it immediately ceases to accept communication the failed process.
2. It sends a message to every other GMS process with which it is communicating

When the shunned process is still operational, it will find out that it is shunned by its next attempt to communicate with the GMS. The only way it can rejoin the group is by re-joining with a new process-ID.

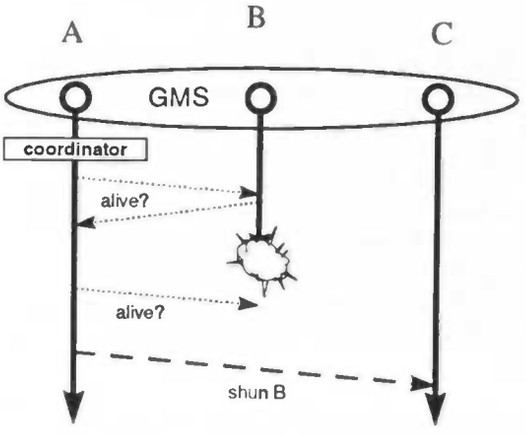


Figure 24 Failure notification within the GMS

When a failure has occurred or when there is an addition request, the GMS coordinator starts updating the membership list, which will then be replicated among all other GMS processes. If the processes that are being added or deleted do not include the old GMS coordinator, the coordinator performs a two phase protocol:

1. in the *first* round the list of add and delete events are send to the participants, including the coordinator itself.
2. all participants acknowledge receipts. The coordinator waits for as may replies as possible. It requires that majority of the current GMS member processes respond. If less than a majority of processes are reachable the coordinator waits until communication is restored, before it continues again. If processes have failed and only a minority of the GMS member processes are available, a special protocol is executed.
3. After reception of the acknowledgments the GMS coordinator commits the update in a *second* round, which also carries with it notifications of any failures that were detected during the first round. The second-round protocol can even be compacted with the first round of a new instance of the deletion protocol.
4. The GMS members update their membership view upon reception of the second-round protocol messages.

When we define the number of GMS member processes to be n and we define the complexity of the algorithm based in terms of send operations, we get the following table:

Round	amount
First round (temporary updates)	n
First round (acknowledgment)	n
Second round (commits)	n
Total	$3n$

The complexity of in this case is $O(n)$, if no intermediate failures occur. The more intermediate failures occur during the execution the more the protocol has to be executed. In any case, the more members a group contains, the greater the chance a failure occurs and the more often the protocol has to be executed.

When the current coordinator is suspected of having failed and some other coordinator must take over, a three-phase protocol is needed:

The new coordinator starts by informing at least a majority of the GMS processes listed in the current membership that the coordinator has failed and then collect their acknowledgments and current membership information.
At the end of this first phase, the new coordinator may have learned of pending add or delete events that were initiated by the prior coordinator before it was suspected off having failed. The first-round protocol also has the effect of ensuring that a majority of GMS processes will start to shun the old coordinator. The second and third rounds of the protocol are exactly as for the normal case.

In the three-phase case, an extra $2n$ send operations have to be performed, so the complexity remains $O(n)$.

Ricciardi has provided detailed proof that the above protocol results in a single, ordered sequence of process add and leave events for the GMS and that it is immune to partitioning (see Ricciardi [1992]).

The second part of the estimate concerns the case where the GMS is used by client to add and join a group. They do so by invoking the *join*, *monitor* and *leave* operations on the GMS interface. The process that wishes to join locates an operational GMS member (for example by inspecting a environment variable on the hostcomputer of the process) It issues a join RPC to that GMS process. If it times out, the RPC can be reissued to some other member. When the join succeeds, it learns its ranking (the time at which the join took place) and the current membership of the GMS service. Similarly, a process wishing to report a failure can invoke the leave operation in any operational GMS member. If that member fails before confirming that the operation has been successful, the caller can detect this by receiving a callback reporting the failure of the GMS member itself and then can reissue the request.

The GMS offers brings power to several protocols, when the GMS is used to ensure system-wide agreement on failure and join events, the illusion of a fail-stop computing environment is created. Under conditions when the GMS can make progress, such protocols like the three-phase and data replication can make progress and will maintain their consistency properties continuously, at least when permission to initiate new actions is limited to the primary component in the event that the system experiences as a partitioning failure.

5.2.2 Group Communication

We will now focus on the ordering of messages in a group of replicated processes. We define a communication primitive as a method for sending a message to a set of processes that can be addressed without knowledge of the current membership of the set. In this context a multi-cast is a protocol that sends a message from one sender process to multiple destination processes. We define receiving a message as the arrival of a message from a sender into a message queue, where messages reside for possible re-ordering. When all necessary re-ordering has taken place, the message is *delivered* to a process. This means the message is removed from the queue and presented to the application code.

When a process wants to do a multi-cast, it needs to know to which processes (the members of a group) he has to send a message. We now define a non-changing particular group of processes, where no members leave or join the group, as a *view*. As soon as a join or leave event occurs, the view becomes a *different* view. The GMS is

responsible for updating the views in accordance to reality and should offer the latest views on process groups for multi-casting. We can now precisely state the definition of virtual synchrony (as defined by Joseph and Birman in 1985) is: A view-synchronous communication system ensures that any multicasts initiated in a given view of some process group will be failure-atomic with respect to that view and will be terminated before a new view of the process group is installed. Failure-atomic is defined as the property that if the multicast has been performed successfully all members of the view have received (and delivered) the message. If there was a failure of one of the members, no member delivered the message.

There are several types of multicasts. We are interested in a failure-atomic multicast, which guarantees that for a specified class of failures, the multicast will either reach all of its destinations or none of them. Depending on the circumstance, there are really two forms of failure atomicity that might be interesting:

1. *dynamically uniform*, it guarantees that if one process delivers the multicast, then all processes that remain operational will do so, regardless of whether or not the initial recipient remains operational subsequent to delivering the message.
2. *standard multicast*, the only guarantee is that if one waits long enough, one will find either that all the destinations that remained operational delivered the message or that none did so.

The key difference concerns the obligation incurred by the first delivery event. From the perspective of a recipient process *p*, if *m* is sent using a protocol that provides dynamic uniformity, then when *p* delivers *m* it also knows that any further execution of the system in which a set of processes remains operational will also guarantee the delivery of *m* within its remaining destinations among that set of processes. On the other hand, if process *p* receives a non-uniform multicast *m*, *p* knows that if both the sender of *m* and *p* crash or are excluded from the system membership, *m* may not reach its other destinations.

Although dynamic uniformity is a costly property to provide, it sometimes is very desirable to provide, if actions upon receiving *m* will leave some externally visible trace that the system must know about, such as redirecting an airplane or issuing money from an automatic teller.

Ordering

The ordering of messages directly influences the consistency the state of the replicated processes. The type of ordering also determines the (computational) cost of group communication primitives.

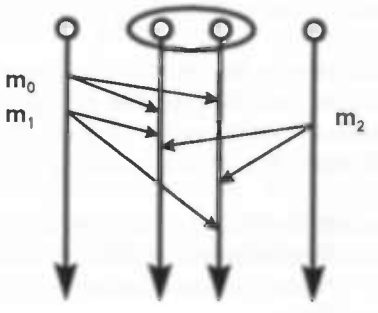


Figure 25 Sender Order Multicast

In the beginning of this chapter we presented the unordered multicast, which does not provide guarantees concerning consistency (see Figure 22). There are other multicasts which have certain properties that are sometimes desired by client/server applications. We will now present these multicast types:

- *Sender* order, or FIFO order: if the same process sends m_0 and m_1, then m_0 will be delivered before m_1 at any destinations they have in common
- *Causal* order, if the message m_0 is sent before the message m_1 (by any sending process), m_0 will be delivered before m_1 at any destinations they have in common.

- *Totally*, *Agreed* or *Atomic* ordered, any processes that receive the same two messages receive them in the same order.

In Figure 26 we can see a causally ordered multicast. At the left server (in the ellipse) the message arrive in the sequence m_0, m_1, m_2. At the right server the sequence is m_0, m_2, m_1. Because the m_0 arrives before m_1 at each server the ordering is causal. In Figure 27 we can see a totally ordered multicast. At both servers the messages arrive in the order m_0,m_1,m_2. This ordering is also causal, but if message m_0 would suffer from latency at both servers, and they would both receive the messages in the order m_1,m_0,m_2 the ordering would still be total, but no longer causal. The atomic multicast is often needed in client/server applications. For example, imagine a service that supplies a unique number to a client. The service (which in this case should be highly available) is replicated by means of a process group. If two clients request a unique number and the order of incoming requests differs from server process to server process, the states of the replicated servers are no longer synchronized and each server returns a different number to the clients.

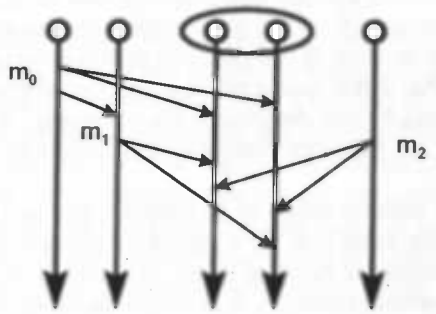


Figure 26 Causally Ordered Multicast

For each type of ordering we can make yet another discrimination between weakness and strength:

1. a *weak* total ordering property would be one guaranteed to hold only at correct processes, namely those remaining operational until the protocol terminates.
2. a *strong* total ordering property would hold even at faulty processes, namely those failing after delivering messages but before the protocol as a whole has terminated.

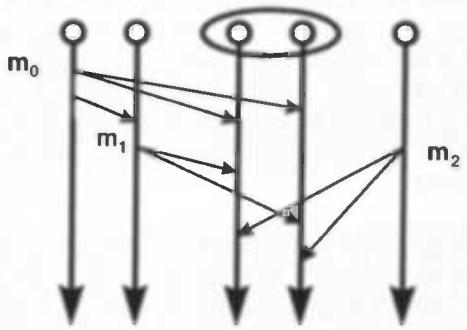


Figure 27 Atomic Multicast

One can observe that in real life missions the strong total ordering property is often needed, because the ordering of messages often have externally visible consequences. These consequences could be noticed by an external observer interacting with a process that later fails, and then interacting with some other process that remained operational.

So we conclude that to guarantee a reliable state synchronization of all replica objects in an ODE in the presence of communication and host failures, we need a strong totally causally ordered communication primitive, or else the risk exists of loosing complete state synchrony.

Costs

We now provide a course outline on the costs of several group communication primitives (sender order, causal order and total order multicasts).

1. *Sender ordered* multicast, this multicast is sometimes called the *fbcast*, with two variants: a non-uniform and a dynamic uniform (or safe) *fbcast*. These are relatively inexpensive broadcasts, since the protocol requires only one round of communication, i.e. $O(n)$. The latency of this protocol is equal to the latency of the network. The safe *fbcast* is costlier because of an extra round of communication to ensure all members have received a copy; around twice the network latency. This protocol can however not be used in cases where a state-transfer occurs.

2. *Causal ordered* multicast, or sometimes called the *cbcast*. A protocol which presents an $O(n^2)$ upperbound is as follows. A sender (the coordinator) transmits a message by means of a reliable stream-style channel to all other members of the group. The channel is only broken, if the GMS reports the departure of an end point. When a process receives a message it delivers it immediately and resends it to the remaining destinations. Every member of the group will now receive one copy of the message from the original sender and copies of the other participants. A proof of why this is non-uniform failure-atomic can be found in Birman [2]. This simple protocol has a $O(n^2)$ complexity if we base the complexity on the amount of sending processes. Making use of hardware broadcasts and compressing rounds of the protocol by tapping into information in the lower levels of a communication system, we could reduce the complexity below $O(n^2)$ but still above $O(n)$. If we would extend the protocol to be dynamically uniform, we would have to assure that no process delivers the message until it is known the processes in the destination group all have a copy. This can be realized by adding extra rounds to the protocol, i.e. the processes should receive the message, store it and *not* deliver it. The should first send an "OK to deliver" message to the coordinator, which should then send a message to every process stating it that all have seen the message and that they may commit, etc. Without going into details the reader can observe that these round are of a linear nature. Further piggy-backing of messages during communications will result in a better performance, but no smaller complexity. There are also less message costly protocols (close to $O(n)$), but which require more overhead due to the use of time-stamps in the messages.
3. *Total ordered* multicast, or *abcast*, there are several protocols for this multicast, like the one developed by Chang and Maxemchuk, which is quite simple and has not much overhead. It is token-based: A sequencer process (a distinguished process) which publishes a an ordering on the messages that it has received; all other group members buffer received messages until the ordering is known and then deliver them in the appropriate order. Provided that the group only has a single token, the token ordering results in a total ordering for multicasts within the group. There is another algorithm which uses a two-phase protocol (see Lamport [July 1978] and later adapted to group communication settings by Skeen). In the first phase of communication, the originator of the multicast (the coordinator) sends the message to the members of the destination group. They receive the message, but do not deliver it. Each process then proposes a delivery time for the message using a logical clock, which is made unique by appending the process-ID. The coordinator collects the proposed delivery times, sorts the vector, and designates the maximum time as the committed delivery time. Its sends this time back to the participants. They update their logical clocks (and hence will never propose a smaller time) and reorder the messages in their pending queue. If a pending message has a committed delivery time, and this time is smallest among the proposed an committed times for other messages, it can be delivered to the application layer. This protocol can be made fault-tolerant by electing a new coordinator if the original sender fails. This protocol however has a non-uniform property and making it dynamically uniform (see the example of cost estimation of a Non-uniform Failure-Atomic Group Multicast above) would be more costlier (i.e. linear).

This list does not encompass all complexity issues, for example communication with non-group members is not discussed. This is not necessary since this summation already shows the basic costs of different communication styles. We can conclude that the costs for a strong totally causally ordered communication primitive vary from above $O(n)$ to $O(n^2)$, based on the amount of overhead in messages that is excepted.

5.3 Conclusions

We have showed that the costs of using virtual synchrony as a function of the amount of replica objects in the ODE varies between linear and quadratic complexity (under the assumption of the crash-failure model). In other words, in the worst case the costs of this kind of replication grows quadratically with respect to the amount of messages that have to be send, when adding an extra replica object. However the factor which is usually

neglected, i.e. the amount of rounds of communication (e.g. $O(2n)$ or $O(3n)$) becomes very important in for example packet-switched networks were latency can become quite high when network traffic is high.

When latency is high, a single round of communication may take some time. Every extra round will cause noticeable delay. So while the costs of the protocols may be relatively small, the actually delay can become high due to network latencies. This can degrade the experienced performance with respect to time-sensitive QoS-attributes dramatically Furthermore, when there is a lot of packet-loss, the protocols which depend on the Group Membership Service become costly protocols, since packets have to be retransmitted and when to much packets are lost the groupsize and the membership have to be reestablished. Therefor careful choice of group communication primitive is therefore necessary, depending on the exact demands of the distributed application under development and the assumed failure-model.

We have also seen that replication based on virtual synchrony, can guarantee state synchronization in highly dynamic environment like an ODE, but does this at the cost of using extra communication bandwidth and computing time. To establish how replication through virtual synchrony influences QoS we assume the following situation: there is an ODE that provides QoS according to agreements in the QoSLAs. We now want to deliver QoS that is described in the QoSLAs. We want to guarantee the delivery in the presence of non-deterministic factors like failures and End-User load, by using replication and without adding extra links or other equipment to the ODE. In this case:

- *Timeliness* related QoS for all services in general will degrade. Only the services that get allocated more resources (due to the replication) the QoS will remain the same or can increase. This is because replication uses more bandwidth of the links: when one QoS like the search-time on a database is guaranteed by adding a second search-engine object during peak-time, the network traffic increases, which directly influences the timeliness QoS of other applications.
- *Volume* related QoS in general does not directly increase and tends to degraded in general. First, the throughput of data through a link cannot be enhanced by adding replicated (software) objects to the system. Secondly the communication used for virtual synchrony takes up more bandwidth. However, End-Users in general may experience a faster communication speed when for example contacting a replicated web-server. Although the amount of volume through the links is not increased, the request can be handled by (through load-balancing) by the replica server objects.
- *Quality of Perception* can be greatly enhanced, due to using extra resources. By replicating objects at different hosts that handle End-User requests, more requests can be handled and thus more data can be transmitted if the links to the End-Users differ enough. Note that the level of enhancement totally depends at how much of the path between (replicated) servers and clients (End-Users) is shared between the clients.
- *Logical Time* related QoS increase significantly, since the logical ordering of messages in time is one of major features of virtual synchrony.
- some of the *Cost* related QoS like the costs of loss of continuity can be highly reduced, because replication provides a higher chance at full-continuous operation. The costs of transmitting data will not be influenced directly by using replication.

6 A prototype QoS controlling application

In the previous chapter we have looked at some of the details of replication based on virtual synchrony. We showed the services a GMS can offer. We now design a prototype QoS controller application for an ODE that is able to exploit the services of the GMS, while assuming a GMS is already available in the ODE. Before begin to discus the actual design in section 6.2, we first present the design process.

6.1 Design

In this thesis we use a specific set of methods to formulate the user requirements and transform them into a real working system. We have to design a prototype QoS-manager which can manage the ODE in such a way that guarantees on QoS-levels are 'arbitrarily' reliable and available. We have split the design process in a sequence of design steps, where in each step only a limited set of design decisions is considered. This method is known as *stepwise refinement* or *top-down* design method: each design step produces a more refined version of the design, that eventually leads to a real working system. The sequence of these steps is called the design trajectory, which is defined in Figure 28.

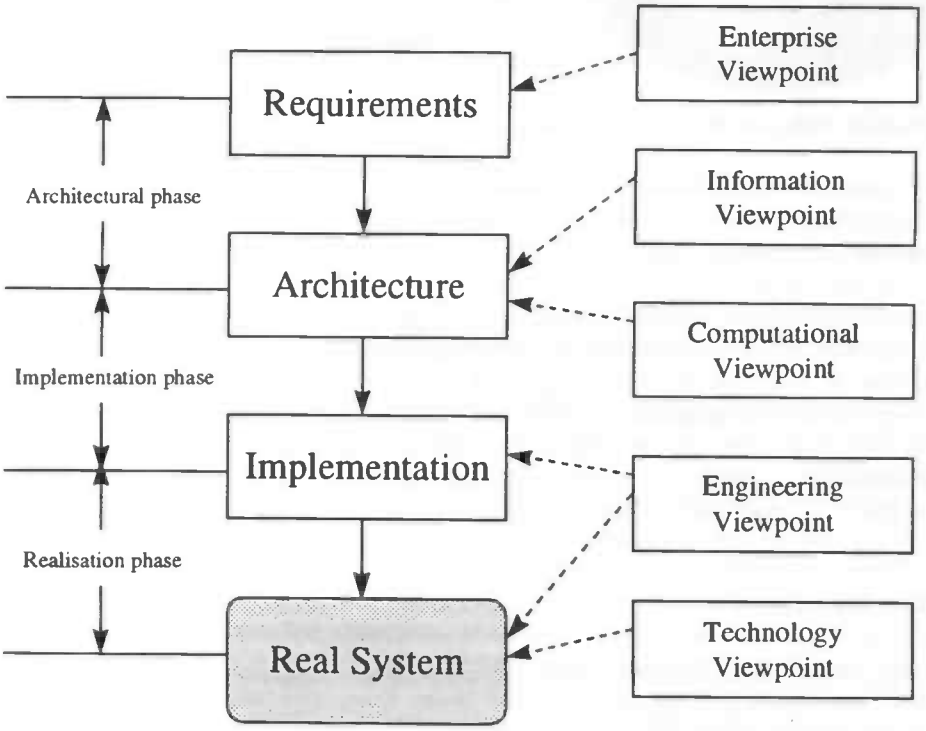


Figure 28 The simplified design trajectory

It starts with User Needs (in our case the desire to manage dependability). From these needs the *requirements* are derived. After the requirements have been established, we design an *architecture* which describes *what* properties the system should posses, in order to comply to the requirements. Then we design an *implementation* that describes *how* the properties are achieved. And finally we use specific hard- and software to build the system and realize it.

We will use the ODP-RM models during various phases of the design: we base the refinement steps on the viewpoints. When we look at the system in the viewpoints in the order Enterprise, Information, Computational, Engineering and Technology, we can see the refinement steps emerge. The requirements are based on the End-User needs and therefor they can be derived from the Enterprise Viewpoint. The Information and Computational viewpoint describe what properties a system should posses, and not how these properties should be achieved. These viewpoints are therefor used to define the Architecture. The Engineering Viewpoint is used for Implementation and Realization and the Technology Viewpoint is used for Realization.

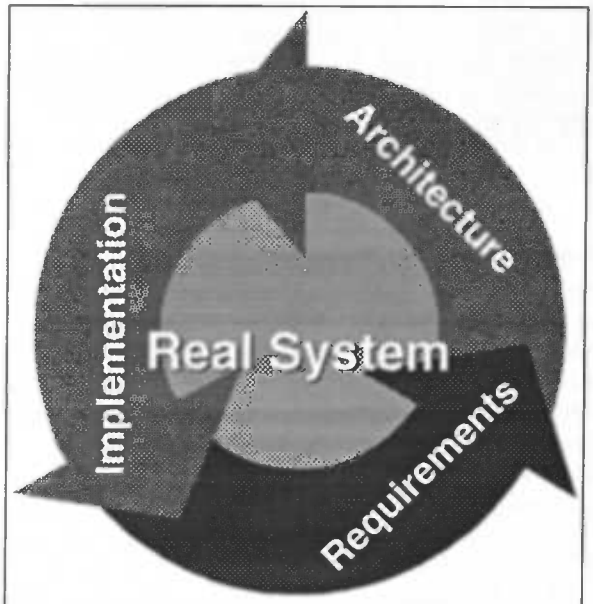


Figure 29 Graphical notation of cyclic designing

Cyclic Trajectory

The trajectory in Figure 28 is rather rigid and would suggest that every possible difficulty or problem during the Implementation phase has been accounted for in during the Architectural phase. This is of course not true. While designing and developing the prototype we encountered many problems and subtle points, that sometimes made us change the original architecture, by re-establishing the requirements, re-designing the architecture and implementing things in another way. This process is also known as the cyclic design (top-down with bottom-up corrections). It can be seen in Figure 29. However to describe the system and understand why it has been built in the way we have done it, we use the simplified rigid design process.

6.2 Requirements

We begin by assessing the needs of an End-User. In our case this is the End-User of the service which QoS has to be controlled. We want the application to manage the ODE in such a way that is capable of controlling QoS at different levels of reliability and availability (e.g. the QoS is available in 50% of all the requests). In accordance to the viewpoint analysis as present in the chapter on ODP-RM we start with the 'Use Cases' method. We describe situations in which actors interact with the system as would like them too interact. These Use Cases implicitly describe the End-Users needs and demands.

6.2.1 The Enterprise Viewpoint

The Use Cases and Actors all come from the Enterprise Viewpoint and we will now define the main Use Cases and Actors.

The End-User

The *End-User* is the only actor who sets the standards: He decides how often the agreed QoS of the service should be available at what costs. We define the End-User to be absolutely ignorant of ODP systems. He is a user that wants QoS delivered at a certain price. This service can be a banking application, an air-line ticket reservation system, but it can also be some kind of distributed database. The End-User does not know how this service works; he just knows that it works and how to operate it. The technical details are not known to him. This ignorance stands in the way when he has to tell the system how reliable and available the QoS-levels should be. Therefore we introduce the concept of

an account-manager (AM). This is someone who is capable of translating the End-Users demands concerning QoS-level guarantees into parameter settings for the ODE. To be more precise the AM is capable of identifying the personal reference framework concerning QoS from the End-User. To give an example. If the End-User says: "I want my database-application to be able to offer a search-time QoS of 10ms, with a probability of 10^{-x} ". The AM should then be able for example to deduce from the geographic structure of the ODE that the client wants his application replicated in at least two computer-centers that are separated by a sufficient amount of kilometers. In case of an earth-quake in the location where one of the computer nodes is located. These persons are in fact the only two human actors. Of course there are the programmers and system-administrators, but these are not specific for the QoS-manager and are always necessary to build and maintain an IT-infrastructure. We are now ready for our first typical Use Case.

Use Case "Establishing the availability of the QoS"
The End-User states that he wants his application to be able to withstand the magnetic disturbances of a solar flare. The SEC deduces that the End-User wants his application to be replicated geographically on a continental scale. The AM then asks the QoS management system if those demands concerning a specific application can be met, i.e. can be guaranteed. The QoS-manager should then reply with an answer. It should say whether or not it is possible to comply to the demands of the End-User, and if possible the system should state at which cost the demands can be met, be that financial cost, degradation of performance or loss of other QoS. If the demanded level can not be achieved, the AM should report this to the End-User. Based on the desire of the End-User the process repeats itself until the system says it is possible and the End-User agrees with the cost level. If a choice has been made, the choices will be stored as a Service Level Agreement and from that moment on the system will maintain the level of dependability according to the way they have been described in the Service Level Agreements for that service.

Use Case "Installing and configuring a new service"
We introduce a third actor: The *Systems Administrator*. This person is capable of making changes to the systems configuration and can install or uninstall applications. When an End-User has expressed his wishes concerning the installation of a new service or the removal of an old one to the AM, the AM should go to the SA and should tell the SA how to set up the distributed application, in order to provide the new service. The SA should then install or remove the software and make the necessary (re)-configurations.

Use-Case "Using a service in the presence of a QoS-level threatening event"
This is a special Use-case. That is, there should be as little as interaction between the QoS-manager and the End-User as possible. If the replication-mechanism can provide enough protection, there should be no interaction at all. We now assume that the End-User is in the middle of using a service and somewhere in the system an event occurs that threatens the guarantee on the QoS-level, like a fault or too many users. A message should be sent to every actor that should be interested in this information. Such an actor could be the SA or the AM. The QoS-manager should notify the SA that a QoS threat has occurred. The QoS-manager should add more replicas or reconfigure the ODE to re-establish the QoS. The amount of effort that is spent and the amount of resources that are used by the QoS-manager in re-establishing the QoS-level depend on the levels of reliability and availability the End-User has specified in the QoSLA.

6.3 Architecture

The architecture is largely based on the system view from the Information and Computational Viewpoint. Therefor we now need to concentrate on the information and computational aspects of the system. But before will we describe the system in these viewpoints, we will first make some remarks on management, to justify our decisions.

6.3.1 Systems Management

Distributed Object Management in an Open Distributed Environment is a subject which concerns many different levels of systems management, like network, computer or storage management.

In the Telecommunications industry the Telecommunications Management Network (TMN) model still grows to be one of the major standards on the area of network management. But there are more like the OSI network management model, *SNMP*, the Simple Network Management Protocol, which is intended to address short-term solutions. *CMOT*, the Common Management Information Services and Protocol over TCP/IP. These two standards come from the Internet Activities Board. As a finally we like to mention the Institute of Electrical and Electronics Engineers (IEEE) which has defined network management standards for local and metropolitan area networks (LAN and MANs). They can be found in the IEEE 802.1 LAN/MAN Management standards. These standards are know as the Common Management Information Protocol (CMIP). For more details see [3].

Although all different standards, it is commonly accepted today that management involves the planning, organizing, monitoring, accounting, and controlling of activities and resources. The OSI and the Internet network management structures are focused principally on monitoring, accounting, and controlling network activities and resources. Planning and organizing are not involved in the OSI/Internet scheme. This is a drawback, since any amount of monitoring, accounting and controlling is futile, if the network is not planned and organized properly.

These standards are mostly partially or completely orientated on the underlying network. Our system is located on the border between top of the DPE and the bottom of the DAE. Although the system should be able to use information concerning guarantees on QoS-levels, it does not manage the network (it might do so in the future, but momentarily we have designed the system not to do so). We still think the OSI management standards are useful because they are organized around object-oriented design techniques.

General concepts
The OSI (and also Internet and IEEE) networks management standards define the responsibility for a *managing process* and a *managing agent*. A network management system, consists out of protocols that convey information about network elements (managed objects) back and forth between various agents in the system and the managing process. The Management Information Base (MIB) (which is a kind of database) is vital to a network management system. All three protocols do not require that the managing process, managing agent and the MIB are to be placed at a particular location in the network. We can recognize location transparency. In practice, the agent software is usually placed in components such as servers, gateways, bridges, and routers. The MIB is usually also located there. The part of the MIB that is of concern to the agent is located at the agent too.

Managed objects are resources that are supervised and controlled by network management. Hardware such as switches, workstations, PBXs and multiplexers can be identified as managed objects. But also software, such as queuing programs, routing algorithms, and buffer management routines, can also be treated as managed objects. Every model has a different understanding of the concept of a managed object (see [3] for more details).

In the OSI model a managed object is defined by four aspects of network management:

1. *Attributes*, also known as the interface or visual boundary. They describe the characteristics, current state and conditions of the operation of the managed objects. Associated with the attributes are attribute values. For example, an object as a PBX line card may have an attribute called status with a value of "operational". Attributes cannot be created, deleted or named during the existence of an instance of a managed objects.

2. *Operations* that may be performed on it.
3. *Notifications* that it is allowed to make. Managed objects are permitted to send reports (notifications) about events that occur on it.
4. *Behavior* that is exhibited in response to operations performed on it. The characteristics of this behavior include how the object reacts to operations performed on it and the constraints placed on its behavior

We have chosen to use these ideas and incorporate them into the Quality of Service Level Agreements. The QoSLAs contain attributes that describe the level of service that should be provided by the service in the Enterprise Viewpoint. For example, look again at the QoSLA example of a set of operations on a database in Figure 30.

SearchTime	= 10 ms
• *reliability* = 50%	
StorageCapacity	= 4 Gb
• *availability* = 90%	
SortingSpeed	= 40 ms
• *reliability* = 60%	

Figure 30 A Quality of Service Level Agreement

The functional descriptions of services are not included in the QoSLA. A functional description of a service belongs in a specification of a service which would contain statements like the kind of operations which can be used an the type of data the service can handle. The behavior of QoS-delivery in the presence of the QoS threatening non-deterministic factors like the User load are indirectly expressed by the reliability and availability settings for every QoS-attribute.

A managed object is managed by a managing process. The managing process is defined as part of an application process that is responsible for management activities. Between that process and the object is the agent process, which performs the management functions on the managed objects at the request of the managing process.

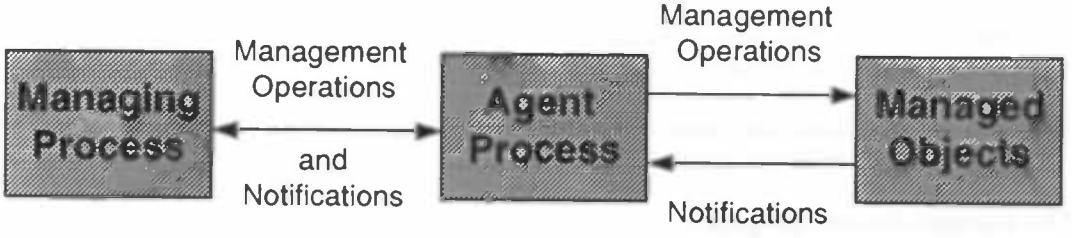


Figure 31 OSI Management Processes

We have chosen to use the same structure in our QoS controlling application: on each node an Agent component resides which collects data and aggregates it, before sending it to the managing component. This is especially necessary when the managing component is replicated. If all information had to be collected without an intermediate Agent, the system would drown in its own information. Structuring a management application like this, has proven to result in good performance (see also Friedrich and Rolia in [9]).

6.3.2 The Information Viewpoint

From previous sections we deduce that the following information that is important for a QoS controlling application:

1. Quality of Service Level Agreements
2. Events that threat the guarantee on a QoS-level

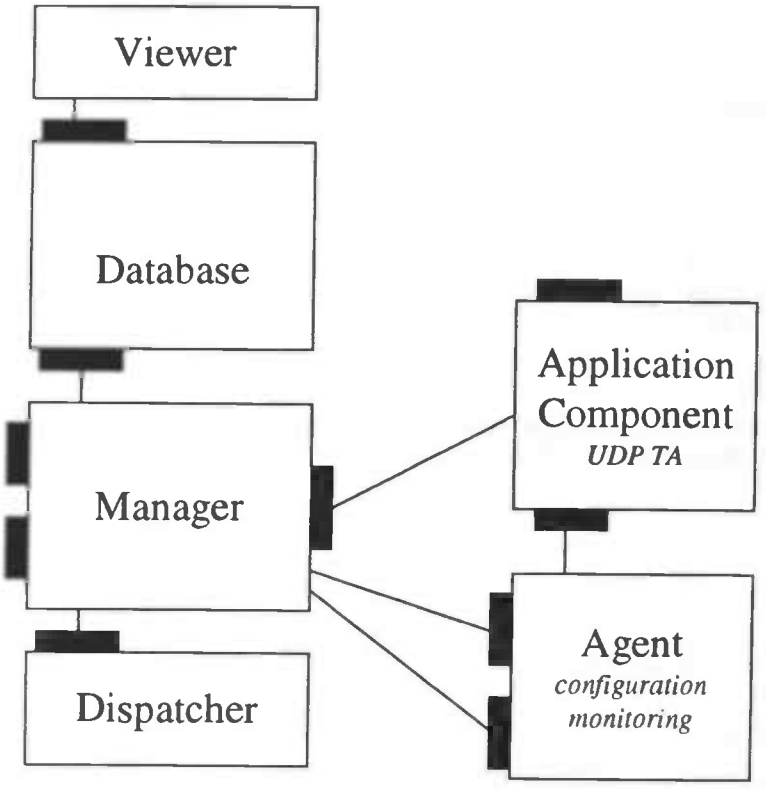
We now present the datastructures that are used by the application to store and transfer information:

```
/*
 * Description: IDL types
 */

// An ApplicationComponent has a name and vendor. This struct
// could contain other information
struct ApplicationComponent {
  string Name;
  string Vendor;
};

// A Service Level Agreement contains the name of the
// ApplicationComponent for which the service levels apply
// The reliability and availability are examples of the
// information which could be contained in an SLA

// The reliability field is used by the Manager to determine
// how much replicas should be placed on the host
typedef short Reliability_info ;
typedef short Availability_info ;

struct ServiceLevelAgreement {
  string Name;
  Reliability_info Reliability;
  Availability_info Availability;
};

// The information on a host
typedef long SystemInfo;
```

Note that the SLA is not the same as the QoSLA example in Figure 15. The fields in the IDL-types description above are a simplified version.

6.3.3 The Computational Viewpoint

We will now present the computational objects that are able to manipulate the information in the Information Viewpoint according to the demands of the Enterprise Viewpoint. We adopt the same structure that can been seen in Figure 31 OSI Management Processes.



In the Computational View on the QoS-manager we can see the managing object 'Manager'. It manages the Agents, which are responsible of configuration and monitoring the Application Component, in this case the Terminal Agent from a application platform called UDP. The Manager provides information to the a Dispatcher, which is responsible for sending QoS-related information to interested actors like Account Managers and End-Users. The Manager uses a Database where it stores the Quality of Service Level Agreements. This is separated in a separate computational object, because storage and retrieval techniques of the QoSLA should be the task of a separate database, since the manager object will be replicated and should not contain much data. The viewer provides a view on the stored QoSLA and performance data.

Figure 32 QoS controlling application in the Computational Viewpoint

6.4 Implementation

We present the interface of the Manager, the Database and the Agent. The interface are defined in OMG-IDL. We begin with three interfaces of the Manager component:, the Configuration-Manager, the MonitorManager and the Evaluator. The included `types.idl` can be found in section 6.3.2 where the Information Viewpoint is discussed.

```
/* Description: IDL interfaces of the manager */

#include "types.idl"

interface ConfigurationManager {

  // IDL operations
  void Launch(in ApplicationComponent pApplicationComponent);


  // Database operations

  // A database can ask the manager for a suitable database
  // for synchronization
  void GetDBHost(out string pHostName);

  // A database can tell the manager that he is ready to
  // be used for synchronization.
  void SetDBHost(in string pHostName);

  // Set a Service Level Agreement
  void SetSLA(in ApplicationComponent pApplicationComponent,
              in ServiceLevelAgreement pServiceLevelAgreement);

  // The Manager is initialized by launching 1 replica and the
  // 'ignite' it by invoking the 'Ignite' method. The Manager will
  // then start replicating itself on the host
  void Ignite();
};
```

```
interface MonitorManager {

  // Probe for information
  void Probe(inout SystemInfo pSystemInfo);
};

interface Evaluator {

  // Service Level Agreement Breach Of Contract, is invoked by
  // the managed and replicated Applications when a groupmember
  // has left the group of replicated ApplicationComponents
  void SLABoC(in AppCInfo pAppCInfo);

};
```

The Database interface provides methods for storage and retrieval. There are also methods for browing the SLAs in the database. These methods are used for replicating the data in the databases.

```
/* Description: IDL interface of the database */

#include "types.idl"

// when all SLA need to be extracted from the SLA database
// a sla_iterator is used for 'browsing'
typedef short sla_iterator;

interface Database {

  // Every ApplicationComponent has a SLA of its own which
  // can be retrieved and stored
  void RetrieveSLA(in ApplicationComponent AComp,
                   inout ServiceLevelAgreement NSLA);
  void StoreSLA    (in ApplicationComponent AComp,
                    in ServiceLevelAgreement SLA);

  // a SLA-iterator can be requested for by invoking this method
  sla_iterator GetSLAIterator();

  // the SLA database can be browsed by invoking the NextSLA
  // method with the requested SLA-iterator
  void NextSLA(in sla_iterator SLAit,
               inout ServiceLevelAgreement NSLA);
};
```

The last interface belongs to the Agent. It provides methods for activating (launch) and de-activating (kill) ApplicationComponents on the hosts. It also contains a Probe method for extracting information from the host:

```
/* Description: IDL interface of an Agent */

#include "types.idl"

interface ConfigurationAgent {

  // IDL operations
  short Kill(in ApplicationComponent AppComp);
  short Launch(in ApplicationComponent AppComp);

};

interface MonitorAgent {

  // IDL operations
  void Probe(inout SystemInfo SysInfo);

};
```

We now have the architecture that specifies the design of the QoS controlling application in different viewpoints. We now focus on the basic mechanism of the QoS-controller. In section 4.4 we concluded that in order to control QoS we had to develop a mechanism that would partially adjust the ODE in the case of a demand for extra QoS support. We will now provide a possible implementation of such a mechanism in an ODE. This mechanism is visible in the Engineering Viewpoint.

6.4.1 Engineering Viewpoint

We have modeled the mechanism as a back feedback loop in Figure 33: the achieved QoS is measured in the ODE by a monitor. This information is evaluated after aggregation and the achieved levels are compared to the demanded levels in the QoSLA by using the measurement model from section 4.1. When a drop in QoS is detected, the QoS controller 'renegotiates' with agents at hosts for resources in the ODE. This renegotation means that the QoS-controller tries to establish whether or not there is room for a replica-object at a host. When the controller has allocated enough resources it can commit the reconfiguration. The QoS is then measured again by the monitor and the loop continues.

Monitoring and configuring is done locally on the separate hosts by the Agents. Evaluation is done by the Manager (see Figure 32), just as managing the renegotiating and the reconfiguring. We will now discuss each of these tasks.

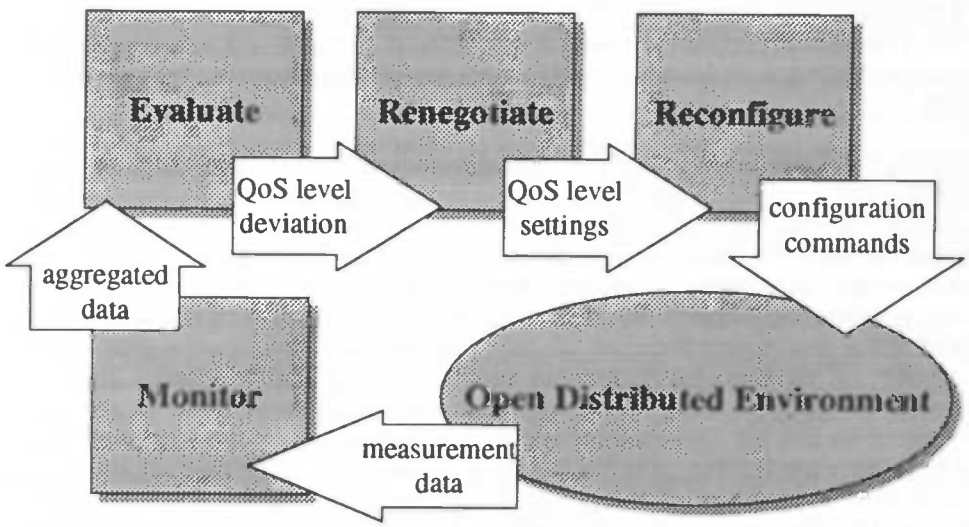


Figure 33 a management feedback loop

Monitoring

A monitor ideally is an external observer that does not interfere with the system. When a monitor is constructed in hardware, it can often be done in such a way, that it does not directly interfere with the normal system operations. A software monitor always interferes with a ODE by using resources, like memory, computing time and network facilities. This is called the Heisenberg effect for software, modeled after The Heisenberg Uncertainty Principle:

> *In 1927 Heisenberg formulated the uncertainty relation:*
>
> $$\Delta x \, \Delta p \geq \hbar/2$$
>
> *This means that the uncertainty in the position (x) times the uncertainty in the momentum (p) is greater than or equal to a number called h-bar divided by two. We cannot measure either position or momentum exactly. The more precisely we*

determine one, the less we can know about the other. This is a fundamental property of quantum mechanics.

Sending measurement data on a need to know basis (in an aggregated form), reduces the amount of communication. Less network traffic reduces the Heisenberg effect. Therefor we have incorporated this into the design and the mechanisms. Besides the standard problems concerning monitoring in a non-distributed system, ODP systems have a number of extra problems concerning monitoring, as Schroeder enumerates in [29]:

1. delays in transferring information mean this information may be out of date.
2. variable delays in transferring information result in events arriving out of order.
3. the number of objects generating monitoring information in a large system can easily swamp monitors.
4. in the likely event that the distributed system is heterogeneous, a canonical form is needed to encode messages passed between heterogeneous machines.

Our prototype can deal with the third point, as we mentioned before: information is collected, aggregated and sent on a need to know basis from Agent to Manager. Also point four does not pose any real problem, since we have used CORBA for implementing the system. CORBA offers an Interface Definition Language which can be used to communicate data between heterogeneous computers. The first problem remains a problem if the transmission speed of connections is not high enough and cannot be easily solve. The second problem however is also difficult to tackle, but as we have seen the communication protocols we have used that are based on virtual synchrony solve this problem .

Not all measurement data can be easily extracted from the ODE. For example, when a host has crashed or it can no longer communicate through the network, the Agent can no longer send his measurement data to the Manager. To determine whether or not a host can no longer be used, the Manager uses two strategies. The first strategy is to let agents send monitor information at relatively short intervals to the Manager. This information contains a time-stamp, which is stored as the most recent time-stamp from a specific host. If the time-stamp becomes too 'old', the machine is declared suspicious and it is 'pinged', i.e. the Manager determines whether or not it is still available. The second strategy involves inspecting information that comes from a replica from which one of the members has left. The members of such a replica-group can send information about the hosts they are located on to the Manager. By comparing the new situation with the old (i.e. before the member left the group), the Manager can deduce which host(s) are suspicious. The Manager can then exclude the host from the list of operational hosts (resources) until a 'ping' of the host(s) has proven the host is available again.

Evaluation of achieved and guaranteed QoS
This is done on the basis of measurement model we developed in Chapter 4. The monitored (machine-measured) QoS data is translated to the End-User perceived QoS. This is compared to the statements in the QoS Level Agreement. We define the QoS to be no longer available when the perceived QoS is not equal to or above the QoS as described in the QoSLA. If the QoS is no longer available, this should be logged in order to compute the reliability and availability of the QoS-delivery.

A special case is determining what the probability is of preserving the state of a service-providing component (and thus the continuous operation of a service) in the case of a break-down of the underlying hardware (e.g. the crash of a host). Continuous operation is not directly a QoS, but computing its probability is necessary to determine if the QoS controlling application should migrate the service-providing object to another host or add extra replicas. There are several algorithms in literature which can compute this probability based on the ODE configuration. For example FARE, as designed by Kumar et. al. in [15]. They discuss an algorithm for describing the reliability of a computer communication network under program execution constraints. It is also based on the idea that a given a certain configuration of executable modules, files and nodes in a network,

there exists a number of possible configurations that can be used to execute a program on a certain node. A shortcoming of this algorithm is that it is not applicable for distributed components running on more than one node. FREA, the Fast Reliability Evaluation Algorithm (by Lin et. al. [19]), employs a different concept for computing reliability. It is based on the generalized factoring theorem with several incorporated reliability-preserving reductions to speedup the computation. More complicated algorithms like the A* algorithm (see Franken in [7]) also deal with routation problems. But FARE, FREA and certainly the A* algorithm (and its heuristic version) are complicated and in section 4.4 we objected against the use complicated time-consuming algorithms that perform configuration-based calculations. We therefor will not discuss them any further, but provide a simple algorithm (presented in the following part on reconfiguring) that provides an estimate on the probability of service-continuation in the presence of the break-down of hosts.

Renegotiation
When the QoS deviation has been established and a drop in QoS has occurred, the QoS controlling application can perform several actions based on what type of QoS-delivery was demanded. If the QoS was provided (see section 3.3):

1. *guaranteed*, then the controller should try to *reserve* more resources to deliver the demanded QoS. If no new resources can be found, the QoSLA is violated and the QoS should be marked as unavailable for as long as the requested QoS cannot be re-established. Note that if the service is not being used, the resources should still be reserved for this service, i.e. remain allocated to the components that provide the service.
2. *compulsory*, then the controller should perform the same actions as in the case of guaranteed QoS delivery, but if a renegotiation does not lead to re-establishing the requested QoS, the service should no longer be provided. Note that the resources are not claimed. If the service is no longer used, the resources are no longer claimed.
3. *best effort*, the controller should try to allocate as much resources as possible to give the End-User the highest possible level of QoS, while not violating the QoSLAs of the other services.

In section 4.1.5 we proposed the use of Fuzzy Sets (or Fuzzy Logic) to translate numerical QoS values into verbal descriptions, like *good*, *medium* or *bad* QoS. We could also make use of Fuzzy Logic when the controller decides what it should do based on the difference between perceived QoS and demanded QoS. The theory of Fuzzy Logic speaks of so called rule bases which state what a system should do, based on the contents of a 'fuzzy' variable (like the translated QoS) As an example we provide such a simplified rule-base in Table 2 (see also Figure 17).

Table 2 Rule base for renegotiating and reconfiguring

IF	TRANSMISSION SPEED	EQUALS	BAD	THEN	REROUTE TRANSMISSION PATH
IF	TRANSMISSION SPEED	EQUALS	MEDIUM	THEN	ALLOCATE MORE BANDWIDTH
IF	TRANSMISSION SPEED	EQUALS	GOOD	THEN	DO NOTHING

The advantage of using such a general applicable rule base is that for example allocation policies can be determined by system-administrators, without knowledge of the QoS demands of every End-User. An extra advantage is the fact that fuzzy logic based control systems often tend to control a dynamic system with more stability and accuracy than conventional controllers like PID-controllers.

Determining which components should get more resources or which components should be replicated should determined by looking at the QoS measurement tree (see section 4.1). This tree shows which technological objects (the leaves of the tree) are responsible for the total QoS. Because there is usually more than one component to choose from, a choice should be made. This choice should be based on what resources are still

available and what objects can be replicated. For example, in the case of the database-search operation the QoS controller could try to migrate the database-search operation to a faster host or could add another replica to the system. Which choice should be made can be determined by a rule-base or other heuristics.

Reconfiguring
Reserving and reallocating resources is done in the reconfiguring phase. In our mechanism the Manager uses the Agents at the hosts to activate the service providing components at the host and to allocate the resources. When using the Agents the Manager makes use of distribution transparency: each Agent has CORBA-compliant interface. No matter what machine, CPU or operating system (Wintel, DEC/Alpha, Motorola, UNIX), a service providing component can be activated at a host. The Agent is encapsulated by the OMG-IDL compliant interface. The Manager can invoke the launch calls without knowing what type of computer is the Agent is located on.

Note that Manager and Agents reconfigure the ODE while it is *on-line*. This means that services are *not* shut down. Migration and reactivation of service-providing objects is done, while request for service keep coming in and have to be handled. This implies reconfiguring must be done carefully, or else the ODE might become unstable, because of bouncing requests and resource allocation problems. We state that it is the responsibility of the DPE (enhanced with a GMS service) to handle the blocking of calls while migrating and adding service-providing objects.

Example
We now present an example of an algorithm that implements (parts of) the evaluation, renegotiation and reconfiguring phase. The input of the algorithm consists of a requested probability that the service will continue to operate in the presence of break-down of hosts. When the algorithm has finished and there were ample resources, replica-objects have been placed at hosts in such a way, that the provided service will continue to be delivered in the presence of break-down of hosts. If there were not enough resources (hosts) the probability will be as high as possible. The design of the algorithm is based on the theory of structure functions or Reliability Block Diagrams (which can be found in [1] and [12] respectively).

We define a system to be an interacting set of n components. x_i denotes the state of the i-th component. If x_i equals 0 then the component is operating, if x_i equals 1 the component has failed. This is accordance with the crash-failure module we use throughout the designs in this thesis.

The state of the entire system is determined from the vector of component states $x = (x_1, x_2, \ldots, x_n)$ by the function $S(x)$. If $S(x)$ equals 1 then the system is operating (and thus the services are being provided). If $S(x)$ equals 0 the system is not operating and has failed providing all services. $S(x)$ is called the *structure* function. It is easily seen that a large ODE will have many components and that the vector x will be very large. Therefor we suggest a vector x per service in this example.

We propose that the vector can be derived from the viewpoint descriptions. First the computational viewpoint is used to derive which computational objects are needed to provide the service. Then, via the mapping to the Engineering Viewpoint, we can derive which Engineering Objects are needed. Every Engineering Object (BEO, Binder, Capsule etc.) is mapped to a technological component. In this example we do not see every technological component as an x_i, only the nodes are.

The number of components which are operating, when the state of the system is defined by x, is called the size, $s(x)$, which is computed as follows: $s(x) = x_1+x_2+\ldots+x_n$.

A vector for which $S(x)$ equals 1 is called a *path*. A vector for which $S(x)$ equals 0 is called a cut. A vector x is a minimal path if $S(x)$ equals 1 but fore every $y < x$ (which means the vector x contains more 1s than y does) then $S(y)$ equals 0. This means that if any one of the components which were operating for the path x were to fail, then the

system must fail. So as longs as one of the minimal paths of a system operates the service can be provided. In an analogue fashion (dual to be precise) the *minimal cut* can be defined: the minimum amount of components that have to fail to reach the point of service failure.

Using the concepts of paths and cuts we define two system categories which are sufficient to model all component dependencies.

A series system is a system in which all components must operate for the system to operate. For example in Figure 34 both A and B must not fail in order to have an operational system. In formal notation the structure function becomes $S(x) = x_1 * x_2 * .. * x_n$.

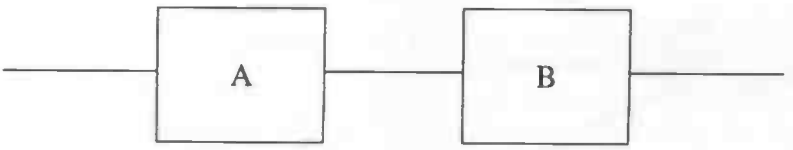


Figure 34 Two-component series system

A parallel system is a system in which only one component needs to operate for the system to operate. (see the two-component parallel system in Figure 35) . A set of *n* replicated servers is in fact a *n*-component parallel system if we assume the crash-failure model, when we use no voting mechanism. In formal notation the structure function becomes $S(x) = 1 - ((1-x_1) * (1-x_2) * \ldots * (1-x_n))$.

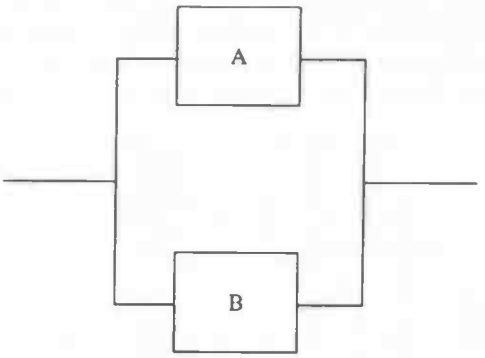


Figure 35 Two-component parallel system

When we make combinations of series and parallel systems, we can model almost any system that consists out of components that interoperate to deliver a service. It is possible to draw such a system, but we also use *algebra*. Without going into details, we present a few rules for computing structure functions. If two systems with structure functions $S_1(x_1)$ and $S_2(x_2)$ are put in series then there structure function becomes $S(x)= S_1(x_1) * S_2(x_2)$. And if they are put in parallel then the structure function of the resulting system is given by $S(x)=1- (1-S_1(x_1)) *(1- S_2(x_2))$. Here, $x = (x_1,x_2)$.

We can now calculate the probability of a cut in the Reliability Block Diagram (and thus a service failure) by replacing x_i with p_i, the probability that an object is operational. The reliability of a series system then becomes:

$$R(p) = p_1 * p_2 * \ldots * p_n$$

and for a parallel system:

$$R(p) = 1 - (1-p_1) * (1-p_2) * \ldots * (1-p_n)$$

Why this is possible and correct, can be checked in [1] and [12], we only present it here. We could go on, presenting more rules and define systems with a higher level of complexity, but that would be a contradiction to the previous sections. To see why,

remember that we would use this model in the Engineering Viewpoint and remember the statistical independencies that have to exist when multiplying probabilities. In an ODE it can be possible (and most of the time it is the case) that Engineering Objects reside on the same host, therefor simple multiplication does not suffice here, because of stochastic interdependencies between the engineering objects. Instead, we only want to use this model to compute the reliability of sets of *n*-replicated servers, that are located on different hosts. When located on different hosts that are geographically separated we

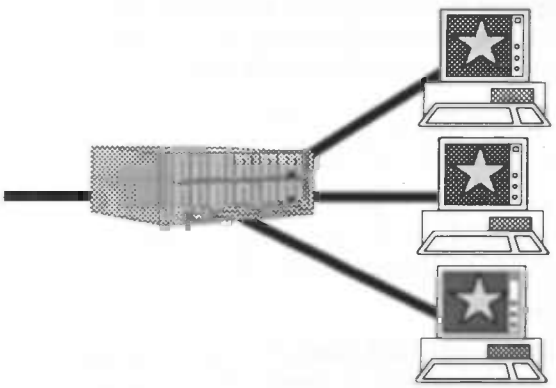


Figure 36 Triple replicated 'star'-service on a simple sub network

can assume a certain amount (enough in this case) of statistical dependency.
As an extra example we will look at a replicated server object. The groups exists of three members and is interlinked by an Ethernet cable and a hub. We have modeled the reliability of a server as the reliability of the host on which it resides, i.e. p_s. We model the Ethernet cable and the Ethernet card as one system that has a certain chance of failing, i.e. $(1\text{-}p_n)$. We model the hub as a single system that can fail, i.e. p_h.

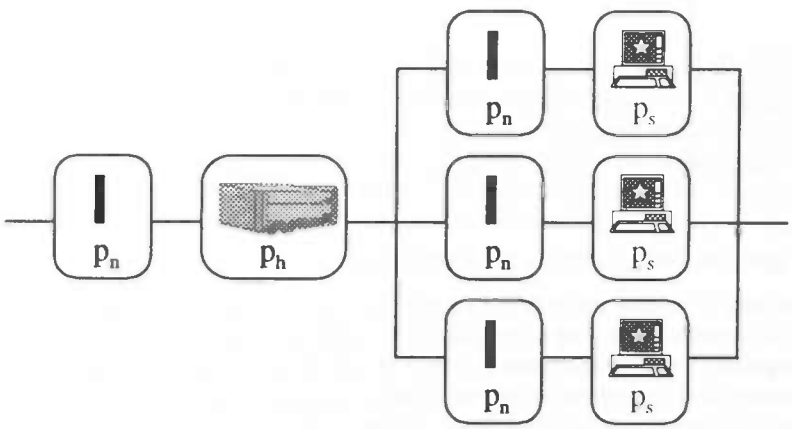


Figure 37 Reliability Block Diagram of the triple replicated 'star'-service

The reliability of the entire service now is: $(p_n \cdot p_h) \cdot (1 - \prod_{1}^{3} (1 - (p_n \cdot p_s)))$. This formula directly shows that the more replicas are used, the less the total probability of failure becomes. We now present the algorithm:

1. Start with the basic level of reliability that is independent of the amount of replicas, i.e. the hub reliability (p_h) and the reliability of network connection of the hub (p_n). These components build a series system, so their probabilities should be multiplied. Initialize the variable `reliability` with this value.
2. While the contents of variable `reliability` is not equal or greater than the desired reliability and not at least two objects have been placed at different hosts go to step 3, else go to step 4.
3. temporarily allocate a host (maybe through transaction processing) that offers a certain level of reliability. Add the reliability factors (i.e. p_n and p_s) of the host itself

and its network connection to the formula and recompute the `reliability`. At every loop the reliability will now increase. Go to step 2.

4. If no more hosts can be allocated then the requested level of reliability can not be guaranteed, based on the desired dynamic QoS behavior (guaranteed, compulsory and best effort), the controller can decide to make the allocation of hosts permanent and start the service or abort the allocation of hosts and roll-back to the original situation.

This mechanism can also be used when a replica in the group crashes and a new replica has to be added if the QoSLA are be breached. In this case, the reliability of the remaining members is computed and new replicas are added according to the algorithm above.

This phase of the management feedback loop has been implemented in a very simple way: the Manager tries to find possible hosts on which it can place new replicas. It does so by looking in the Database to find out, what the QoS-levels of services at the different hosts are. The information in the Database on QoS-levels is supplied by the Agents on those hosts. It is at this place were a *load-balancing* could be implemented: by placing the replicas on selected hosts in a smart work-load "spreading" fashion, more performance can be acquired from the ODE. We now present the actuall C++ code which implements this algorithm. This code is taken from the core of Manager and implements the Launch method of the Manager:

```
void ManagerCore::Launch (const ApplicationComponent&
pApplicationComponent){

  // Retrieve the Service Level Agreement from the database

  .....

  // Create an SLA
  ServiceLevelAgreement aServiceLevelAgreement;

  // Let the Database fill in the SLA
  aDepDBIF->RetrieveSLA(pApplicationComponent,
                        aServiceLevelAgreement);

  short vHostIndex = 0;
  short vReliability = 0;

  // continue to loop as long as the list of hosts with Agents
  // has not been completely traversed and as long as the
  // reliability (the probability that at least one of the
  // replicas is operational) is smaller than the reliability in
  // the Service Level Agreement

  while ((vHostIndex < aAmountOfAgentHosts) &&
         (vReliability < aServiceLevelAgreement.Reliability)) {

    // the code in this while loop assumes the Database and the
    // Agents are available. A more robust version of this loop
    // would include code to check on this

    // Use an ApplicationComponent to retrieve stochastial
    // information on a host. The name of the ApplicationComponent
    // is the name of the host and the reliability field contains
    // the reliability of the host

    ApplicationComponent vHostSpecifications;
    // declare an SLA struct
    ServiceLevelAgreement vHostServiceLevelAgreement;
    // insert a name
    vHostSpecifications.Name = new
    char[strlen(aAgentHostNames[vHostIndex])];
    strcpy(vHostSpecifications.Name,aAgentHostNames[vHostIndex]);
    // and retrieve its SLA from the database
    aDepDBIF->RetrieveSLA(vHostSpecifications,
                          vHostServiceLevelAgreement);
```

```
    if (aConfigurationAgentIFs[vHostIndex]!=NULL) {

     // there is a pointer to the Agent on the host with
     // index 'vHostIndex'.

      // Now invoke the launch method on the Agent
      short succes =
        aConfigurationAgentIFs[vHostIndex]->
          Launch(pApplicationComponent);

      if (succes>=0) {

        // the launch was a succes. The reliability of the
        // computational object may be now multiplied by the
        // reliability of the host on which the replicated
        // object resides

        double a = (100.0 - (double)
          vHostServiceLevelAgreement.Reliability) / 100.0;
        double b = (100.0 - (double) vReliability) / 100.0;
        double c= a*b;
        vReliability = 100 - (100*c);
      }
    }
    vHostIndex++;
  }
};
```

This exceprt of the Manager code clearly shows how Manager, Database and Agent interoperate on different hosts by making use of the location transparency that CORBA offers.

6.5 Conclusions

In this chapter we designed a prototype QoS controlling application which contains a mechanism based on a feedback-loop for controlling QoS in an ODE. We briefly showed how fuzzy logic can be of use in controlling a dynamic system. As an example we provided a simple algorithm which can be used by the application to reconfigure the ODE. The purpose of the algorithm is configuring the ODE in such a way that service-delivery continues (with a certain probability) in the presence of the break-down of hosts.

7 Realization with Orbix+ISIS

In the previous chapters we have provided a model of Quality of Service in an Open Distributed Environment. We have developed a mechanism (based on the use of redundancy) to control QoS in an Open Distributed Environment and we have shown the designs of a prototype QoS controlling application based on the proposed model and used mechanism. At this point we have reached all our goals, except realizing the prototype in a real open distributed environment. In this chapter we will show how we have used the commercially available product Orbix+ISIS to use replication based on virtual synchrony in an ODE. In this chapter we will also discuss the mapping of ODP-RM concepts Orbix+ISIS and prototype realization details.

7.1 Orbix+ISIS

In order to implement the prototype in an open distributed environment, we decided to use the Common Object Request Broker Architecture from the Object Management Group. Because the design of our prototype implied the availability of a GMS in the DPE, we have chosen to use the commercially available Orbix+ISIS product. It is a merger of Orbix from IONA technologies, which provides CORBA support, and the Isis Toolkit from ISIS, which provides a GMS (and replication based virtual synchrony)

7.1.1 Background

The Isis Toolkit was developed by K. Birman and colleagues between 1985 and 1990. It was the first process group communication system to use the virtual synchrony model. As its name suggests, Isis is a collection of procedural tools that are linked directly to the application program, providing it with functionality for creating and joining process groups dynamically, multicasting to process groups with various ordering guarantees, replicating data and synchronizing the actions of group members as they access that data, performing operations in a load-balanced or fault-tolerant manner, etc.

Isis introduced the (primary partition) virtual synchrony model and the *cbcast* primitive. This enabled it to support a variety of reliable programming tools, which was unusual for process group systems at the time Isis was developed. Late in the life cycle of the system, it was one of the first (along with Ladin and Liskov's Harp system) to use vector timestamps to enforce causal ordering.

7.1.2 Employment

A number of applications have been developed using Isis. The toolkit became widely used through a public software distribution. These developments led to the commercialization of Isis through a company, which today operates as a wholly owned subsidiary of Stratus Computer, Inc. The company continues to extend and sell the Isis Toolkit itself.

Successful applications of (Orbix+)ISIS include components of the New York and Swiss stock exchanges; distributed control in AMD's FAB-25 VLSI fabrication facility; distributed financial databases such as one developed by the World Bank; an number of telecommunication applications involving mobility, distributed switch management, and control; billing and fraud detection; several applications in air traffic control and space data collection; and many others. The major markets to which the technology is currently sold are financial, telecommunication, and factory automation.

7.2 Engineering support of Orbix+ISIS

We will now map the concepts of an engineering object, cluster, capsule, node, nucleus and channel onto Orbix+ISIS components. This will enable us to determine to what extent Orbix+ISIS provides support for the ODP-RM management and coordination functions.

Object: engineering objects cannot be clearly and easily observed within the Orbix+ISIS environment. This is because this implementation of the CORBA 1.3 specification is realized by making use of software libraries.

Capsule: a capsule in Orbix+ISIS is a process (or 'application') containing the service-providing objects, which are in fact the basic engineering objects.

Cluster: it is not possible to identify the notion of a cluster in Orbix+ISIS. Replicated objects reside in different capsules and a cluster resides in one capsule. Therefor Orbix+ISIS provides no activation, deactivation and migration of clusters of engineering objects.

Node: a node in Orbix+ISIS is a computer or a node in the ODE.

Nucleus: in ODP terms the nucleus is the operating system on a computing node. There are versions of Orbix+ISIS for Windows NT, UNIX, VMS, etc. Because the code which determines the behavior of an object is encapsulated by OMG-IDL, all operating system functions can be addressed from the code just as in other non CORBA-compliant software.

There are some extra functions provided by Orbix+ISIS, which are in fact an extension of the normal operating system facilities:

- multi-threading, the application programmer can set a variable, which determines whether or not every request should be carried out by a separate thread of execution. To be precise, the details of thread-programming are taken over by Orbix+ISIS.
- reliable messaging, the application programmer can send messages across a reliable connection that provides ordering of messages.

Channel: as mentioned in the section on the nucleus, the channel is a very important concept in Orbix+ISIS. This comes from the fact that Orbix+ISIS is, as mentioned, a modified version of Orbix, placed on top of the Isis toolkit. The designers of Isis have put much effort in providing reliable communication, with an emphasis an group communication. They have built a reliable communication protocol on top of the unreliable UDP protocol.

The stubs are derived from the IDL description of an object by feeding the description to an IDL-compiler, which generates stub- and skeleton-code. The stub code consists out of (proxy-) stub C++ classes. The binders are from and the protocol is UDP. This software is contained in libraries that are linked with client and server objects.

7.2.1 Basic mechanisms

The mechanism for communication interoperability of objects in Orbix+ISIS (and in CORBA) deserves some extra attention. In the Common Object Request Broker Architecture client- and server-objects do not directly communicate with each other but use an Object Request Broker. In Figure 38 we can see communication in both the Computational and the Engineering Viewpoint.

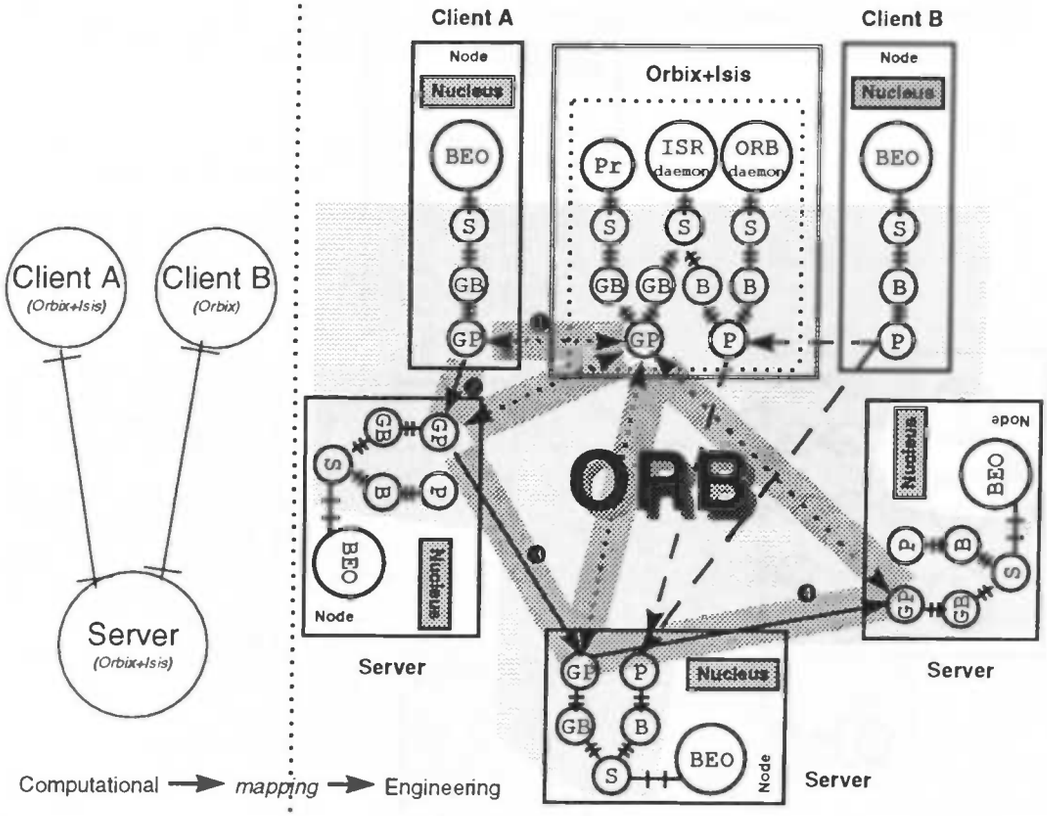


Figure 38 Computational and Engineering specification of Orbix+ISIS and Orbix

We see two cases; a client which has been linked with pure Orbix library code and client which have been linked with Orbix+ISIS library code. In the last case the situation is as follows: when communication between clients and server begins the client sends a binding request to an ISR daemon (`isrd`) at an host (which can be specified if the service is not on the local host or the service cannot be automatically located by a locator service), which determines whether the request is meant for a Orbix+ISIS server or for an Orbix-only server. This request goes through the Isis communication lines. If the request was meant for a Orbix+ISIS server the request is sent through Isis communication lines using the communication primitives described in section 5.2.2. If the request is for a standard Orbix-only server the request is passed on to the standard Orbix daemon, which arranges a connection between the client and server. In both cases after the request for binding has been handled, there is a direct connection between client and server. In the Orbix+ISIS case there is a so called `protos` daemon which is an implementation of the Group Membership Service (see section 5.2.1).

we can see this in graphical notation. Note the fact that the protocol- and binding- objects are different for Orbix+ISIS and *pure* Orbix objects. Note that the Orbix and Isis daemons have to be present on each node. For reasons of simplicity we have left them out. The protos daemon only needs to be available on several hosts (see for details).

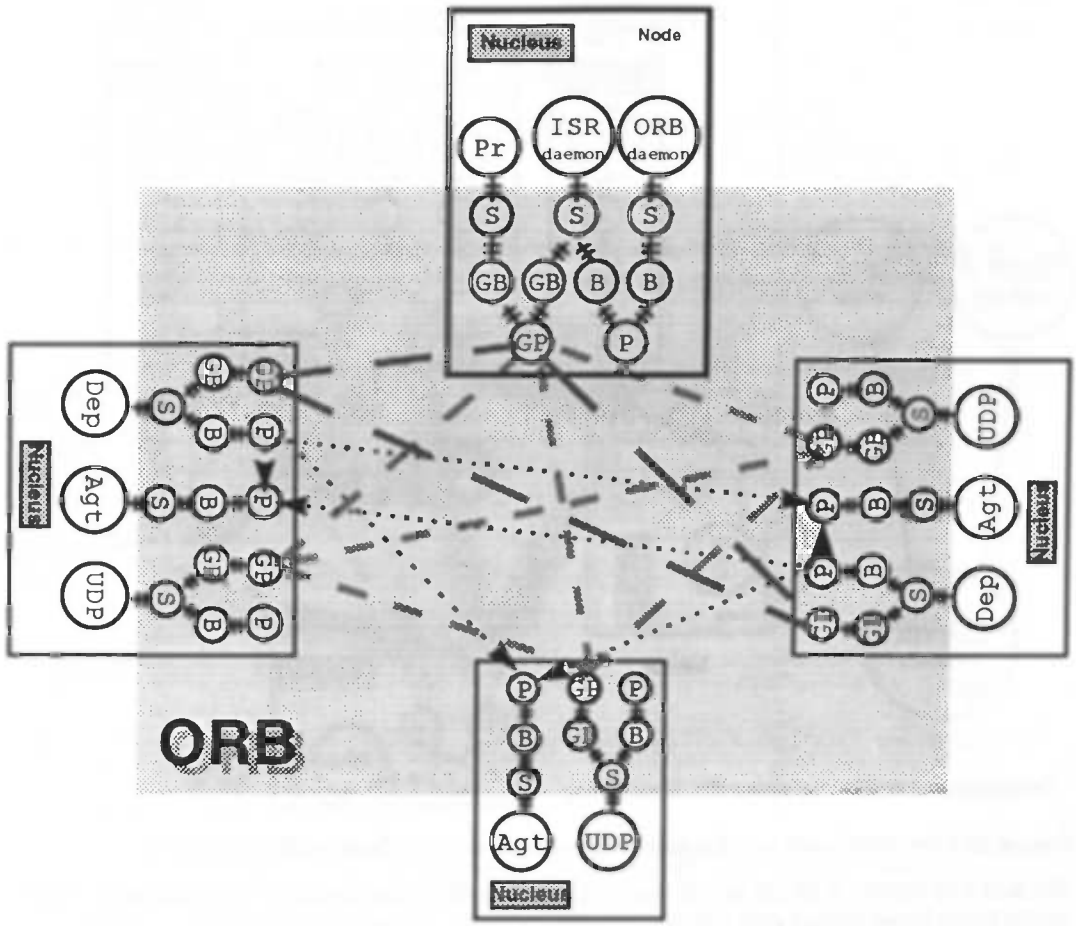


Figure 39 The prototype QoS Manager

In Figure 39 we can observe the a simplified 'Engineering View' on the prototype QoS controlling application and the surrounding ODE. On four nodes we can see several engineering objects like the Manager (`Dep`), the Agents (`Agt`) and the application (`UDP`) which QoS has to be controlled. The striped and dotted lines are communication patterns between the different objects. Again we have not drawn the `isrd` and `orbd` objects at every node. There is also one `protos` daemon. The Manager objects are replicated on the West and East node. The Agents are **not** replicated but placed at every node except the North node. The `UDP` application is replicated at the West, South and East node. Note that the binder-objects and the protocol-objects are different when the ISIS based group protocol is used.

7.2.2 Implementation details

We want to present some details on the implementation of the distributed application, that show how Orbix+ISIS supports the replication of objects. We begin with the inheritance of replication. We present the definition of a C++ object-class which is one of the interfaces of the Manager.

```
/* Description: Implementation of the ConfigurationManager Interface
 */

#include "manager.hh"
#include "ManagerCore.h"

class ConfigurationManagerImpl :
  public virtual ConfigurationManagerBOAImpl,
  public virtual IsO::ActiveReplica {
```

The implementation of the ConfigurationManager interface is capable of replication by inheriting replication-functionality from the `IsO::ActiveReplica` class.

```
public:

  // pointer to the Core
  ManagerCore* aCore;

  // temporary buffer for state transfers
  char* aStateBuffer;

  // constructor
  ConfigurationManagerImpl(const char* pCOName, ManagerCore* pCore);

  // destructor
  ~ConfigurationManagerImpl();

  virtual void Launch (const ApplicationComponent
                       &pApplicationComponent, CORBA_Environment
                       &IT_env=CORBA_default_environment) ;

  virtual void GetDBHost (char *& pHostName,
                          CORBA_Environment
                            &IT_env=CORBA_default_environment);

  virtual void SetDBHost (const char * pHostName,
                          CORBA_Environment
                            &IT_env=CORBA_default_environment);

  virtual void SetSLA (const ApplicationComponent&
                       pApplicationComponent,
                       const ServiceLevelAgreement&
                       pServiceLevelAgreement,
                       CORBA_Environment
                       &IT_env=CORBA_default_environment);

  virtual void Ignite (CORBA_Environment
                       &IT_env=CORBA_default_environment);
```

The C++ functions that implement the methods listed in the IDL interface (see section 6.3.3) look very similar to the IDL interface. To complete the definition, we present the methods that are needed for state synchronization.

```
// Orbix+Isis methods needed for replication
virtual void _receiveState(CORBA_Request &state,
                           CORBA_Environment &IT_env=
                           CORBA_default_environment);

virtual void _sendState(CORBA_Request &state,
                        CORBA_Environment &IT_env=
                        CORBA_default_environment);

virtual void _memberLeft(CORBA_Object* member,
                         MemberLeftReason reason,
                         const IsO::Groupview* newView,
                         CORBA_Environment &IT_env=
                         CORBA_default_environment);
};
```

The last function is invoked by Orbix+ISIS when a replica object has left the group. This means that if one of the Manager replica leaves the group, this method is invoked at all the others replica Managers.

These last three functions are interesting with respect to the support of Orbix+ISIS, and therefor we present the implementation code of these methods:

```
void ConfigurationManagerImpl::_receiveState(CORBA_Request &state,

CORBA_Environment &IT_env) {
```

```
  aStateBuffer = new char[DEFAULT_STRING_SIZE];

  // receive the state from the group
  state >> aStateBuffer;
  // and set the state of the core
  aCore->setCoreState(aStateBuffer);

};
```

The state is retrieved from a stream-like variable `state`. This variable is supplied by Orbix+ISIS. Note that the state is not set by the interface, but by the core of the Manager.

```
void ConfigurationManagerImpl::_sendState(CORBA_Request &state,
CORBA_Environment &IT_env) {

  // get the state from the Core
  strcpy(aStateBuffer,aCore->getCoreState());
  // and send it
  state << aStateBuffer;

};
```

The state is sent to the stream-like variable `state`. Note that the state retrieved from the core of the Manager as a string. This string contains the contents of the state-variables. And finally the method which is invoked when a replica leaves the group:

```
void ConfigurationManagerImpl::_memberLeft(CORBA_Object* member,
  MemberLeftReason reason,
  const IsO::Groupview* newView,
  CORBA_Environment &IT_env) {

  // Orbix+ISIS supplies a message containing the reason why a
  // member left

  if (reason==memberLeft) {
      DB("A Manager Replica was released");
  } else if (reason==processFailed) {
      DB("A Manager Replica has failed");
  }

  // Inspect which members are still there
  IsO::GroupviewIterator lGroupviewIterator(_getGroupview(IT_env));
  CORBA_Object* lMemberObject = lGroupviewIterator();
```

By using the `IsO::GroupviewIterator` an object can determine which other objects are still in the group. The Manager uses this information to determine the difference between the old groupview and the new groupview. The hosts of objects that were present in the old groupview but not in the new groupview are marked as suspicious.

```
  char lHostsThatSurvived[DEFAULT_STRING_SIZE];

  while (lMemberObject!=NULL) {;

    // determine the host which are still available
      ostrstream
        lHostsThatSurvivedStream(lHostsThatSurvived,
                                 DEFAULT_STRING_SIZE);

      lHostsThatSurvivedStream << lMemberObject->_host();
      lMemberObject = lGroupviewIterator();
  };

  aCore->determineSuspiciousHost("Manager",lHostsThatSurvived);
```

When the suspicious hosts have been marked, the Manager should restore the groupsize.

```
  aCore->restoreGroupSize(_myRank(IT_env),
```

```
                                        _getGroupview(IT_env)->groupSize(IT_env)
                                    );
};
```

We conclude with the Launch method of the interface. This is not the same as the Launch method of the core. In our implementation the interfaces invoke the core (this is because of other implementation details which are not relevant to the topics in this thesis).

If every core of a replicated Manager would launch an ApplicationComponent, then the ApplicationComponent would be launched too often. Only one Manager should perform the actual launch, but all replica Manager objects should have the same state. Orbix+ISIS provides support through different communication styles. We use the coordinator-cohort style to let only one replica Manager object launch an ApplicationComponent.

```
void ConfigurationManagerImpl::Launch (const ApplicationComponent
&pApplicationComponent,

CORBA_Environment &IT_env) {

  switch (coordCohort(defaultCoordCohortChooser,NULL, IT_env)) {
      case takeover:
        DB("Taking over failed coordinator.");
      case coordinator:
            {
                  DB("Invoking Launch on Core object");
                  aCore->Launch(pApplicationComponent);
                  // No need for state transfer after a launch
                  break;
            }
     case gotReply:
            {
                  DB("received but no coordinator");
                  break;
            }
      case notCoordCohort:
            {
                  DB("ISR file for Launch not right! Use CoordCohort
for Launch operation");
                  exit(GERROR);
                  break;
            }
      default:
            {
                  DB("bad value from coordCohort()");
                  exit(GERROR);
                  break;
            }
  };
};
```

Based on the rank of an object (see section 5.2.1) the coordinator is chosen (rank 0 means an object is the coordinator). In the `switch` statement, the application asks Orbix+ISIS using `coordCohort` if it is a coordinator or not. If it is, it invokes the launch method on the core of the Manager. If not, nothing happens. Although it cannot be seen in this code, there is also a possibility for a state-transfer afterwards (see also the receive and send state functions).

7.2.3 Functions

We now focus on the support of Orbix+ISIS and the prototype QoS Manager for the ODP-RM functions we discussed.

Management Functions

The *node management* functions is in fact implemented by the standard C++ libraries and the Orbix+ISIS libraries that provide the processing, storage and communication functions within a node.

The *object management* function is partially done by the protos daemon, which determines whether objects are alive or not by pinging them and controlling the mechanisms for state-transfer between objects (also see the checkpoint and recovery function).

As said in the introduction to ODP-RM, we did not find direct support *for cluster-management*. The same goes for the *capsule management*, although some elements like the constructor en destructor functions of C++ can be related to the management of objects in processes. But there is no clear and distinct mapping.

Coordination Functions
We found a direct support for an event notification function in Orbix+ISIS. The communication primitives which are the basic building blocks of the services offered by the Orbix+ISIS environment, can and are used for a reliable Event Stream service. Events can be sent to a repository from which (replicated) servers can read, all done with reliable communication links.

The *checkpoint* and *recovery* function can be partially recognized in the so called `_sendstate` and `_receivestate` (C++) functions. These are offered by the Orbix+ISIS environment to the application programmers: in these functions the variables that make up the state can be sent and received through reliable communication primitives. These functions are in fact highly related to checkpointing an object, i.e. freezing the state for state-transfer or retrieval.

Because we did not find direct support for cluster management functions we also did not find direct support for the deactivation and reactivation of clusters. However, the QoS manager is able to activate and deactivate capsules containing clusters/basic engineering objects on hosts through the use of agents.

Orbix+ISIS provides very good support for the group function. The *membership*, i.e. deciding which members of a group (of objects) participates in which interactions, according to an interaction policy is handled by the GMS (protos). The same goes for the *ordering* of events and messages, for which there implementations of multicast communication primitives. *Collation* is provided for in the shape of programmer definable smart proxy objects that can collect replies from replica-objects and perform operations on them. Support for *interaction* is indirectly provided by a rank variable that every member of a group has. This variable can be accessed by the implementation code of every object separately, which enables the members to decide by *themselves* whether or not they will interact.

The good support for the group function, enabled us to identify the replication function. In fact Orbix+ISIS has named the grouping of objects with a synchronous state "Active Replication".

Because of the offered state-transfer and other replication facilities migrating can be done by activating a replica at another host and deactivating a replica at a different host. Coordinating the migration is done by the QoS manager. For example: when the agents notify the manager that the workload on specific hosts becomes unbalanced, objects are migrated (on-line) to other hosts.

7.3 Experiments

We have performed small experiments which were focused on the recovery of the QoS controlling application in the presence of failures. Both hard-stop and soft-stop scenarios were used to test the application. In the soft-stop case, a replicated Manager component was killed by using system-calls from the operating system (e.g. the KILL signal from

UNIX). When a replicated component was killed (softly), the system responded within milliseconds: the moment the soft kill was visible on the host were the component was originally located, a new replicated component was activated at a different host. The host which had performed the softkill was marked by the application as suspicious and not used until the remaining Manager replicas had established that the host could be used again. During a hard-stop scenario we disconnected a host (with a Manager component) from the network. In this case it took considerably longer (20 seconds) for the QoS controlling application to regroup. This is because the underlying Orbix+ISIS layer needs time to find out whether there is a simple delay of messages, or that the host is unreachable. Too much disconnects or too much packet loss (1%) will degrade the performance considerably.

8 Conclusions & Recommendations

Conclusions: in this thesis we have provided a model of Quality of Service in an Open Distributed Environment. This model enabled us to develop a mechanism to control QoS in an ODE. The mechanism is based on exploiting the intrinsically available redundancy in an ODE, by replicating software components like service providing objects. A major feature of the model and the mechanism is the use of a feedback loop in which the ODE is continuously adjusted on a local scale by adding or migrating replica objects. This in contradiction to many other models in the literature which try to control QoS by constructing probability models and by computing entire ODE configurations based on the QoS demands. Problems concerning communication between clients and replicated service providing object-groups have been solved (under the assumption of a crash-failure model) by using a communication protocol based on virtual synchrony.

The mechanism has been used in implementing a QoS controlling application in an ODE. The application showed replication is a viable means to assure QoS in ODE in the presence of non-deterministic factors like the amount of (End-)Users, the allocation of resources and external faults. The application also showed that the Orbix+ISIS environment (CORBA-compliant client/server environment) provides good support for the Event Notification, Group, Replication and Migration function from the Coordination functions in ODP-RM.

Recommendations: Using replication based on virtual synchrony in an ODE is a good means for controlling QoS, however it should not be used without care. Applying this kind of technology to control QoS in an ODE, requires a good understanding of the underlying network. Much latency and packet-loss in the network can render virtual synchrony useless.

We do recommend that this kind of technology should be incorporated into any modern day ODE. One of the major advantages of and motivations for using open distributed environments is available redundancy. It would be a shame not to make use of this (in the case of QoS control). We also recommend that replication should be supported by the DPE, the communication primitives based on virtual synchrony should be implemented at the DPE level also.

We did not discuss the financial aspects of QoS control based on replication, although these aspects can be a motivation for (not) applying replication. We looked at these aspects shortly and found out that an IT-economical overview of service down-time and bad QoS is useful for motivating whether or (not) replication based on virtual synchrony should be applied in QoS control. With respect to other further research we suggest that the research on QoS control should continue by transforming the prototype QoS application into a platform or an extension of the middleware layer instead of an application running on the middleware layer. This could be done by offering QoS control as a service of the DPE. We also suggest researching applying fuzzy logic in the QoS controller.

9 References

[1] J.I.Ansell and M.J.Phillips, "Practical Methods for Reliability Data Analysis", Clarendon Press, Oxford, 1994. ISBN 0-19-853664-X.

[2] Birman, K.P., "Building Secure and Reliable Network Applications", Manning, Greenwich, 1996. ISBN 1-884777-29-5.

[3] Uyless, D. Black, Network Management Standars, the OSI, SNMP and CMOL protocols, ISBN 0-07-005554-8, McGraw-Hill Series on Computer Communications, 1992.

[4] Ricky W. Butler and George B. Finelli. (from NASA Langley Research Centre) The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering, 19(1):3-12, January 1993.

[5] J.Dobson, J.Laprie, B.Randell, "Predictably Dependable Computing Systems: An ESPRIT Basic Research Action", Proceedings of the Centre for Software Reliability Conference entitled "Software Reliability and Metrics" 12-14 September 1990

[6] EURESCOM P517, "Distributed Object Technology, a Booklet for Executives".

[7] L. Franken, Quality of Service Management: a Model-Based Approach, Ph.D-thesis 1996, Enschede, The Netherlands

[8] Michael A. Friedman, Jeffrey M.Voas, "Software assesment : reliability, safety, testability", 1995, John Wiley & Sons, Inc, New York, USA.

[9] R.J.Friedrich and J.A.Rolia, "Performance evaluation of a distributed application performance monitor", proceedings of the IFIP/IEEE International conference on Distributed System Platforms, 1996. Chapman and Hall.

[10] D. Hutchison et. al., 1994, Quality of Service Management in Distributed Systems, Computing Department, Lancaster University, book: Network and Distributed Systems Management, edited by Morris Sloman, 1994, University of London. Addison-Wesley Publishing Company.

[11] ISO/IEC DIS 10746-2, "Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model"

[12] K.Kant, "Introduction to computer system performance evaluation", 1992, McGraw-Hill Inc., USA.

[13] S.Kätker, "A Modeling Framwork for Integerated Distributed Systems Fault Management", IBM European Networking Center, proceedings of the IFIP/IEEE International conference on Distributed System Platforms, 1996. Chapman and Hall.

[14] Q.Kong and G.Chen, "Integrating CORBA and TMN Environments", IEEE/IFIP Network Operations and Management Symposium 1996.

[15] A.Kumar, S.Rai, D.P.Agrawal, "On Computer Communication Network Reliability Under Program Execution Constraints", IEEE Transactions on Selected Areas in Communications, Volume 6, Number 8, Octorber 1988, pages 1393-1400.

[16] P.A.Lee, T.Anderson, "Fault Tolerance, "Principles and Practice", 2nd revised edition, Springer-Verlag, New York, 1990.

[17] P.Leydekkers, "Multimedia Services in Open Distributed Telecommunications Environments", KPN Research, May 1996.

[18] P.Leydekkers et Valerie Gay, "ODP View on Quality of Service for Open Distributed Multimedia Environments", Internatinal Workshop on QoS (IWQoS), Paris, March 95.

[19] Min-Sheng Lin and Deng-Jyi Chen, "Distributed Reliability Analysis", proceedings of the Third Workshop on Future Trends of Distributed Computing Systems, 1992, Taiwan.

[20] B.Littlewood, "Limits to evaluation of software dependability", Proceedings of the Centre for Software Reliability Conference entitled "Software Reliability and

Metrics" 12-14 September 1990, edited by N.Fenton et al. Cnetre for Software Reliability, London, UK. Elsevier Applied Science.
[21] S.Maffeis and D.C.Schmidt, "Constructing Reliable Distributed Commuication Systems with CORBA", IEEE Communications Magazine, vol.14 No.2, February 1997.
[22] L. Mejlbro. QOSMIC-Deliverable D1.3C: QoS and Peformance Relationships. Deliverable QOSMIC R1082, RACE, 1992.
[23] V.T. van der Meulen et A.T. van Halteren, "Using Orbix as a TINA-DPE", Mapping Computation objects to CORBA objects, 1995, KPN Research.
[24] M.A. Nankman, "A Reference Model for Open Distribtued Storage Architectores QoS and Open Distributed Processing" .,18 juli 1995, R&D-RA-95-699-FINAL, KPN Research.
[25] L.J.M. Nieuwenhuis, "Fault-Tolerance through Program Transformation", Phd. Thesis, 1991, Enschede, The Netherlands
[26] M. van Opstal, "Software-ontwikkeling in de telecom industrie", januari 1997, jaargang 39, "Informatie", Kluwer Bedrijfsinformatie.
[27] A.Schade, P.Trommler, M.Kaiserswerth, "Object Instrumentation for Distributed Applications Management", IBM Research Division, Zuricht Reserarch Laboratory, proceedings of the IFIP/IEEE International conference on Distributed System Platforms, 1996. Chapman and Hall.
[28] M.Schreider, "A replication mechanism for Open Distributed Processing", september 1995, R&D-SV-95-817, KPN Research.
[29] B.A.Schroeder, "On-Line Monitoring: a Tutorial", IEEE Computer, june 1995.
[30] R. Keith Scott, James W. Gault, David F. McAllister, "Fault Tolerant Software Reliability Modeling", IEEE Transactions on Software Engineering, Volume SE-13, Number 5, May 1987, pages 582-592
[31] I.Stenmark, 26 April 1995, "Availability Management 101", KA-DSM-1614, Research Note Gartner Group, Software Management Strategies.
[32] I.Stenmark, November 1995, "Objects In Production: Systems Management Issues", Research Note Gartner Group, Networked Systems Management (NSM)T-DSM-22429.
[33] J.Warmer, A.Kleppe, "Praktisch OMT", Addison-Wesly, 1996. ISBN 90-6789-776-0.