University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

A Replication Mechanism for Open Distributed Processing

M.J. Schreiner

begeleiders: Prof.dr.ir. L.J.M. Nieuwenhuis
Prof.dr.ir. L. Spaanenburg
ir. A.T. van Halteren

september 1995

RuG

Abstract

Computing systems are more and more intensively used in our society. Hence, more and more real-life situations depend on these computing systems. Therefore, the dependability, (i.e., reliability and availability) of computing systems becomes more and more important.

This thesis defines a replication mechanism to achieve fault-tolerant computing. The Reference Model for Open Distributed Processing (RM-ODP) defines concepts and rules for the description of open distributed systems. The replication mechanism that we propose is RM-ODP conform. The replication mechanism is described from the computational, engineering and technology viewpoint of RM-ODP. One conclusion of this is that the replication mechanism is implemented in the protocol object. This thesis can be a basis to realise the replication mechanism on any platform. In this thesis the mechanism has been realised in ANSAware.

Preface

This master thesis is the result of my graduation assignment for Computer Science at the University of Groningen.

The graduation assigmnent was conducted at KPN Research Groningen and lasted from March 1995 till December 1995. During these months, I worked for the department CAS (Communication Architectures and Open Systems).

I had a very pleasant time here at KPN Research in Groningen. I therefore want to thank several people who enabled this stay and guided my during this stay in Groningen.

First of all, I want to thank the members of the graduation committee: Bart Nieuwenhuis (KPN Research), Ben Spaanenburg (University of Groningen) and especially Aart van Halteren (KPN Research). Their comments and reviewings of my thesis made a big difference and certainly had a great impact on the final quality of this thesis.

Thanks to my fellow graduates, for they have improved the pleasure of working at KPN Research greatly.

And last but not least I want to thanks all my friends and family, for their support, encouragement and positive thinking, during my studies.

Maurice Schreiner,

Groningen,

August 1995.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dependability is an important requirement for modern computer systems. The dependability of a system is a reliance that can be justifiably placed on the service it delivers. Four means to achieve dependability are distinguisted[1]:

- fault avoidance
- fault removal
- fault tolerance
- fault forecasting

Generally, a distinction is made between *human-made faults* and *physical faults*. The former kind of faults include design faults and interaction faults, i.e., faults made by human beings during the design, maintenance and operation phase of the system. The latter type of faults is induced by physical phenomena. Software faults are considered as human-made design faults. Obviously, during the development of a system, a number of activities can be carried out to avoid the introduction of faults. Appropriate design and test procedures can be applied to avoid and remove as many faults as possible. However, human beings are imperfect and probably are not able to govern all physical and logical aspects of computing design, verification and operation and some only come into existence during the life-cycle of the product. Therefore, to improve the reliability of computing systems, fault tolerance will be needed to achieve sufficient quality of service for a large class of applications. However, systems may still fail in spite of fault avoidance, fault removal and fault tolerance activities. Therefore, fault forecasting is needed to determine the probability that the computing systems will operate without failures.

Current developments in computing technology can be characterised by an enormous increase of processing power for decreasing costs. At the same time, high capacity data-communication networks can be supplied for decreasing costs. Obviously, these trends contribute to an increase of the application domain of distributed computing.

Originally the telecommunication infrastructure mainly consisted of cross connects and switching systems. Additional computing systems have been used to support operation and provisioning of new telecommunication services. During the last decade, the "intelligence" has been shifted from switching systems towards these supporting computing systems. These telecommunication infrastructure is therefore evoluating towards a large distributed computing system. An example is the Intelligent Networks architecture[2].

This integration will involve heterogeneous computers and communication systems. Efforts are made to support this integration. Computers from multiple vendors with different operating systems and different networks should be able to cooperate. Structuring computers and communication systems in a standardised way can simplify the integration.

Together, integrated computer and communication systems must provide a so-called *platform* for the execution of distributed applications. Applications, which can be coded for example in a variety of programming languages, must be able to run on any computer in the platform. Different components of an application can be distributed on this platform. This can be achieved when applications running on this platform are also structured in a uniform way and use the standardised structure of computers and communication systems in the platform.
A standardised structuring of computer systems, communication systems and applications can be achieved by providing an achitectural framework. Systems and applications structured according to this framework will then be able to interact. The Reference Model for Open Distributed Processing (RM-ODP) defines such a framework [3] [4] [5]. It defines concepts and rules for the description of open distributed systems.

In February 1995, the Reference Model for Open Distributed Processing has been accepted by ISO as an International Standard. The objective is to provide the community with a common framework for the development and operation of distributed applications. Much more experience with RM-ODP is needed before this framework can be considered a paradigm for distributed computing. Therefore the development of applications and mechanisms conformity to RM-ODP is an important research issue. This thesis is a contribution to these research activities.

In this thesis we focus on fault tolerance, i.e., we propose a mechanism to achieve fault tolerance. The mechanism is based on replication of processes.

The three main objectives of this thesis are:

- How to obtain reliability through redundancy in a distributed environment.
- To design a mechanism that enables the construction of reliable applications.
- Specify the solution of the above objective according to concepts of the Reference Model of Open Distributed Processing (RM-ODP).

Approach and Structure

The developers of ANSAware (a platform) have started to implement a replication mechanism. As they have never finished the implementation, there is no documentation available. There exist only plain C source code. The replication mechanism proposed in this thesis is filtered from the C code.
The goal of this thesis is to make a clean ODP specification of this replication mechanism and realise this mechanism completely in ANSAware. To achieve this goal a full understanding of RM-ODP is necessary.
The RM-ODP is not a design methodology. In order to design the replication mechanism that will be proposed in this thesis, it is necessary to define a design methodology. The replication mechanism is designed according to this design methodology. First the requirements of the replication mechanism, which are necessary in our opinion, are defined. With these requirements the replication mechanism has been designed. The decisions which are made will be described in the particular chapters.
Conform this approach the thesis is organized as follows:

Chapter 2 describes the concepts of the Reference Model for Open Distributed Processing. This chapter constitutes the basis for the following chapters.

Chapter 3 describes the design methodology that is used in this thesis. ODP viewpoints are related to distinct design phases that are identified by this methodology.

Chapter 4 formulates the requirements of the mechanism that will be presented in this thesis. Also it present the architectural phase according to the design methodology stated in chapter 3.

Chapter 5 presents the engineering specification of the model. This chapter defines the mechanism that enables the construction of reliable applications.

Chapter 6 gives some background of the distributed computing platform ANSAware. The computational and engineering mechanism that ANSAware provides, are compared with the computational and engineering concepts of RM-ODP. Further it is explained how the engineering mechanisms of chapter 6 are realised by means of library functions.

Chapter 7 draws some conclusion and also contains recommendations for future research.

Chapter 2

Reference Model for Open Distributed Processing

The rapid growth of distributed processing has led to a need for a co-ordinating framework for the standardisation of Open Distributed Processing (ODP). The Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It creates an architecture within which support of distribution, interworking and portability can be integrated. The aim of this chapter is to describe to concepts the RM-ODP provides.
Section 2.1 describes the general aspects of RM-ODP. Section 2.2 presents the Computational model and Section 2.3 presents the Engineering model. The computational model and engineering model are only described because these are the two viewpoint which are necessary to specify the mechanism.

2.1 Framework

The RM-ODP defines a framework comprising [5]

- five *viewpoints*, called enterprise, information, computational, engineering, and technology which provide a basis for the specification of ODP systems;
- a *viewpoint language* for each viewpoint, defining concepts and rules for specifying ODP systems from the corresponding viewpoint;
- specification of the *functions* required to support ODP systems;
- *transparency prescriptions* showing how to use the ODP functions to achieve distribution transparency.

2.1.1 Viewpoints

The complete specification of any non-trivial distributed system involves a very large amount of information. Attempting to capture all aspects of the design in a single description is generally unworkable. Most design methodologies aim at establishing a coordinated, interlocking set of models, each to capture one facet of the design, satisfying the requirements that concern some particular group involved in the design process. In ODP, this separation of concern is established by identification of five viewpoints, each expresses a particular area of concern.

Enterprise viewpoint

The purpose of the enterprise viewpoint is to explain and justify the role of an ODP system as used by one or more organisations. It is concerned with the business activities of the specified system. It covers business policies, human user roles with respect to the system and the environment with which the specified system interacts.
An enterprise specification describes the overall objectives of a system in terms of roles, actors, goals and policies. Such a specification dictates the requirements on an ODP system.

Information viewpoint

The purpose of the information viewpoint is to identify and locate information within the ODP system, and to describe the flow of information within the system. The syntax and semantics of the information within the system are the main concern.

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. A computational object provides a set of capabilities that can be used by other computational objects. So computational objects interact with each other. A computational specification of a distributed application specifies the structure by which these interactions occur and specifies the semantics of these interactions.

In this viewpoint, aspects of distribution are abstracted from. These abstractions are called distributed transparencies and presume that an underlying mechanism is present, that provides these transparencies. In section 2.1.4 this will be explained in more detail.

Engineering viewpoint

The purpose of the engineering viewpoint is to provide mechanisms and functions to support distributed interaction between objects in the system. The engineering viewpoint is a refinement of the computational viewpoint; here the distribution transparencies must be solved.

Technology viewpoint

The purpose of the technology viewpoint is to describe the physical components, both hardware and software, of a distributed system. The components are e.g. operating system, peripheral devices, communication hardware etc.

The technology viewpoint can be used as a further refinement of the engineering viewpoint, where engineering objects are realised in hardware and software components.

2.1.2 ODP viewpoint languages

The RM-ODP defines five languages, each corresponding to one of the viewpoints defined above. Each language is used for the specification of an ODP system from the corresponding viewpoint and consists of definitions of concepts and rules for the specification of an ODP system. These languages are not formal specification languages.

2.1.3 ODP functions

The ODP function defined in RM-ODP [5] are those that are either fundamental or widely applicable to the construction of ODP systems. The specification for individual ODP functions can be combined to form specifications for components of ODP systems.
Some function descriptions in RM-ODP introduce objects as a simplifying modelling construct. Except where explicit constraints are given on the distribution of these objects, they do not define necessary structure in an implementation.
Each ODP function description contains

- an explanation of the use of the function for open distributed processing
- prescriptive statements, about the structure and behaviour of the function
- a statement of other ODP functions upon which it depends

2.1.4 ODP distribution transparencies

ODP standards aim to enable the implementation of ODP systems that operate consistently and reliably while allowing the distribution of resources and activities. Therefore, an ODP system infrastructure needs to transparently support access to services available in the network.

The ODP infrastructure supports a set of distribution transparencies. Distribution transparencies are used to hide aspects of ODP systems that arise through their distribution. Using the infrastructure the application systems may select the needed transparencies and handle other aspects of distribution characteristics as wanted.
Application designers may not need to be aware of those aspects of distribution and can therefore focus on their application development. When addressing the distribution of their applications, they only have to express their requirements for transparencies.

These are the ODP distribution transparencies [5]:

- **Access transparency**: a distribution transparency which masks differencies in data representation and invocation mechanisms to enable interworking between objects.
- **Failure transparency**: a distribution transparency which masks, from an object, the failure and possible recovery of other objects (or itself), to enable fault tolerance.
- **Location transparency**: a distribution transparency which masks the use of information about location in space when identifying and binding to interfaces.
- **Migration transparency**: a distribution transparency which masks, from an object, the ability of a system to change the location of that object. Migration is often used to achieve balancing and reduce latency.
- **Relocation transparency**: a distributed transparency which masks relocation of an interface from other interfaces bound to it.
- **Replication transparency**: a distributed transparency which masks the use of a group of mutually behaviourally compatible objects to support an interface. Replication is often used to enhance performance and availability.
- **Persistence transparency**: a distribution transparency which masks, from an object, the deactivation and reactivation of other objects (or itself). Deactivation and reactivation are often used to maintain the persistence of an object when a system is unable to provide it with processing, storage and communication functions continuously.
- **Transaction transparency**: a distribution transparency which masks coordination of activities amongst a configuration of objects to achieve consistency.

2.1.5 Consistency rules

A set of specifications of an ODP system written in different viewpoint language should not make mutually contradictory statements, i.e. they should be mutually consistent.

In this thesis we must be aware that there is a correct mapping between the computational specification and engineering specification. The RM-ODP defines some rules which can be applied.

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

Except for transparencies that require the replication of objects, each computational interface corresponds exactly to one engineering interface, and that engineering interface corresponds only to that computational interface.

Where transparencies which replicate objects 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 orginal computational interface.

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.

2.2 Computational model

A computational model defines the functional decomposition of an ODP system into objects that interact at interfaces. The computational model hides the actual degree of distribution of an application from the specifier, thereby ensuring that applications contain no deep-seated assumptions about which of their components are co-located and which are separated. Because of this, the configuration and degree of distribution of the hardware on which ODP applications are executed can easily be altered without having a major impact on application software.

2.2.1 General modelling concepts

The main object modelling concept is an object. Objects within ODP are instantiated from a template and belong to a class. The following definitions are used [5] [6]

Object A model of some real world entity. The state of an object is encapsulated. It can only change as the result of an interaction with its environment or an internal action.

Type A predicate characterizing an object. An object is of a certain type, if the predicate holds for that object.

Template A template is a specification of the common features of a collection of objects, in such a way that an object can be instantiated from it.

Class A class is a set of objects satisfying a type. An object can be member of that class.

Signal An atomic shared action resulting in one-way communication from an initiating object to a responding object.

Operation An interaction between a client object and a server object which is either an interrogation or an announcement.

Announcement An interaction -**invocation**- initiated by a client resulting in the transfer of information from the client object to the server object, requesting a function performed by that server object.

Interrogation An interaction consisting of

- one interaction -the **invocation**- initiated by a client, resulting in the transfer of information from the client object to a server object, requesting a function by the server object, following by
- a second interaction - the **termination** initiated by the server object, resulting in the transfer of information from the server object to the client object in response to the invocation.

2.2.2 Structure of a computational specification

A computational specification describes the functional decomposition of an ODP system, in distribution transparent terms, as

- a configuration of computational objects (including binding objects)
- the internal actions of those objects
- the interactions that occur among those objects
- environment contracts for those objects and their interfaces.

For a configuration of computational objects the RM-ODP prescribes structuring rules (figure 2.1). The rules include binding, interaction, type, template, portability and failure rules.

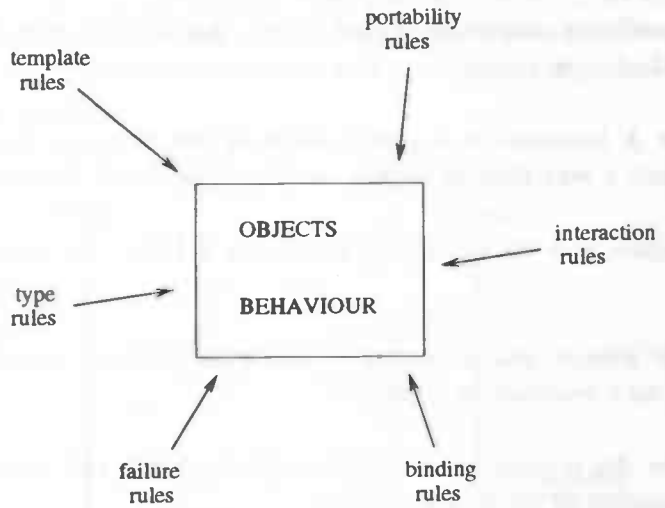

Figure 2.1: A computational specification and its rules

2.2.3 Computational object

A computational object is instantiated from an object template. This template is structured as depicted in figure 2.2. This figure shows that an object consists of a set of interfaces, a behaviour specification and an environment contract.

2.2.4 Computational interface

The computational interface template consists of three components i.e. the signature, behaviour and the environment contract.
The **signature** depends on the interface type which can be either operational, stream or signal;

- The **operational interface** describes a series of interrogations and announcements.
- The **stream** interfaces describes behaviour which is a single non-atomic action that continues throughout the lifetime of the interface. It can be characterised by isochronous information such as audio and video. Its signature contains the type of flow and an indication of causality (e.g. direction of flow).
- The **signal** interface initiates signals and expects responding signals. Its signature describes the characteristics of each signal.

The **behaviour** is described by the set of actions sequences in which it can take part. The action sequence may include internal actions. The behaviour that actually takes place is

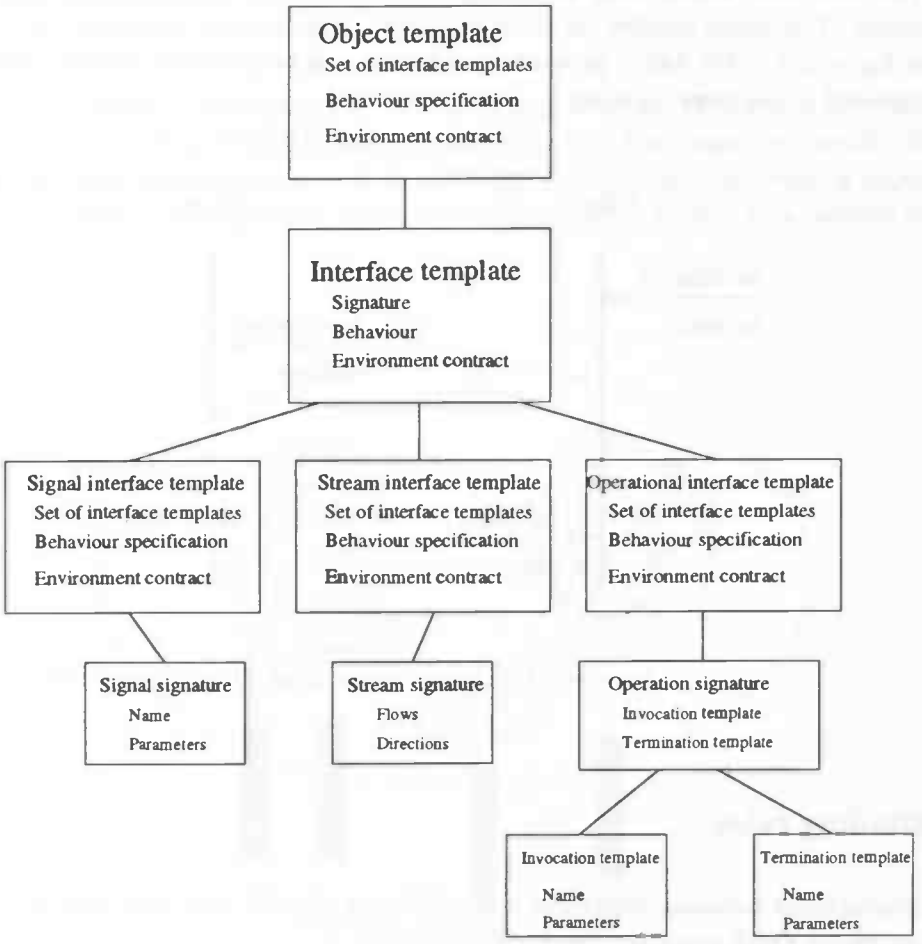

Figure 2.2: Structure of an object template

restricted by the environment in which the object is placed.

The **environment contract** corresponds to a contract between an object and its environment, including quality of service constraints, usage and management constraints. Examples of items in a contract are:

- The object can be located in a certain domain (security constraint, location constraint);
- The object has a maximum probability of failure, i.e. ,it has e certain reliability (failure constraint);
- Interactions at interface X of the object should be performed within a maximum time. (temperal constraint).

The RM-ODP states that an invocation comprises a submit signal followed by a deliver signal. The same applies for a termination. The relation between these signals, is shown in figure 2.3. This figure presents invocation and termination signals, which are all signals involved in an interrogation.

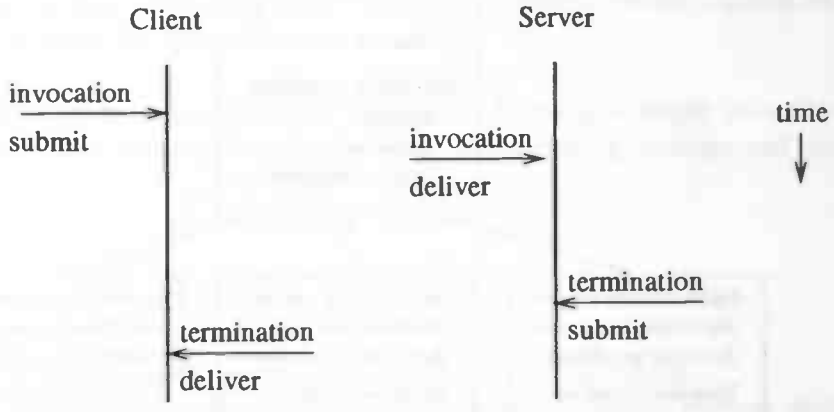

Figure 2.3: Signals comprising an interrogation

Binding rules

Interactions between interfaces can only take place if they are bound. For the binding of interfaces ODP provides the binding object.

Binding objects model the communication between interfaces.

2.3 Engineering model

An engineering specification describes how objects from the computational viewpoint can be distributed geographically.

Objects derived from the computational specification will require several distribution transparencies and the use of resources from the underlying level such as storage, processing and communication facilities. These resources must be provided in a uniform way to enable distribution of objects among heterogeneous architectures.

To hide these effects of distribution, ODP describes an infrastructure of engineering objects. The prescribed functionality of these engineering objects and interactions between engineering objects will then provide the mentioned transparencies and availability of resources.

2.3.1 Structuring of engineering objects

Figure 2.4 represents a configuration of engineering objects within a single node. On a node there is one nucleus bound to one or more capsules. Each capsule consists of clusters, a manager for each cluster and a capsule manager. Every basic engineering object (BEO) is bound to a cluster manager and every cluster manager is bound to a capsule manager.

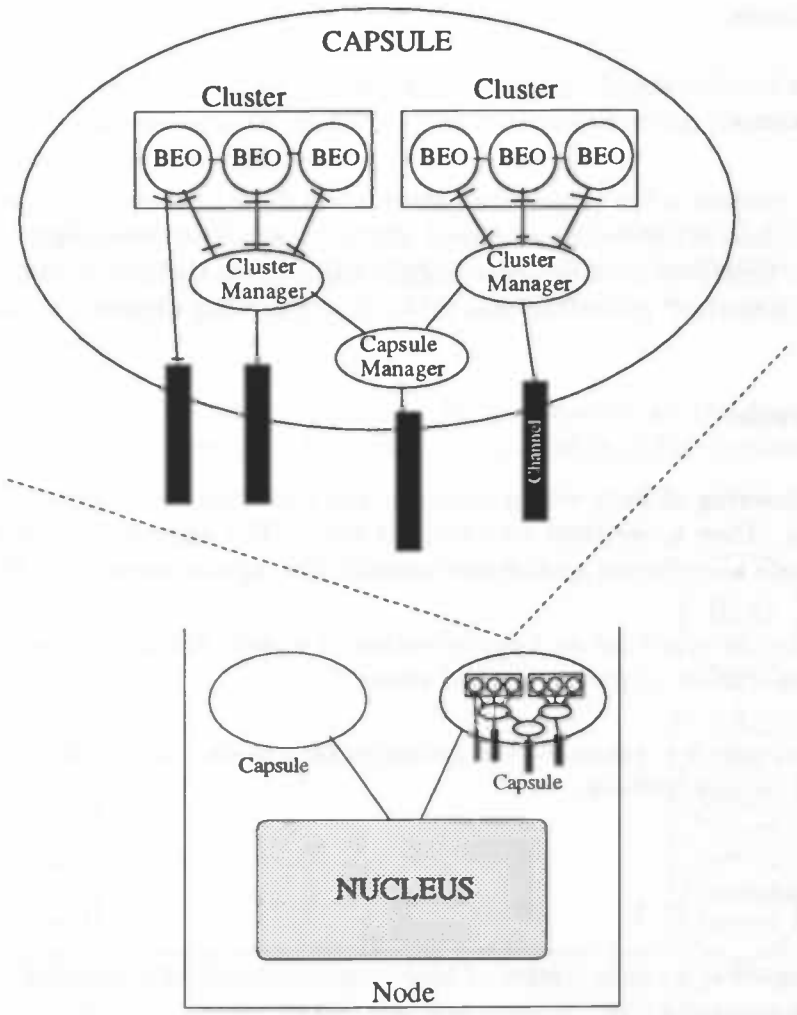

Figure 2.4: An engineering node

The capsule manager is concerned with the instantiation and deletion of clusters and cluster managers.

2.3.2 Description of engineering concepts

Node

A node is an engineering abstraction of a (physical) computing system. In this context a node is meant to be a computing node. A single processor system capable of independent operation is a node, but a terminal is not a node.

A node is defined as a configuration of objects forming a single unit for the purpose of location in space, and which embodies a set of processing, storage and communication functions.

Nucleus

The nucleus is the engineering abstraction of an operating system.
A nucleus is defined as an object which co-ordinates processing, storage and communications functions used by other engineering objects within the same node.
The RM-ODP prescribes that all basic engineering objects are bound to a nucleus.

Capsule

Engineering objects within a capsule are protected from engineering objects in other capsules. They hsave their own address space. If a capsule fails only the objects inside the capsule are affected and objects outside the capsule remain unaffected.

A capsule is defined as a configuration of objects forming a single unit for the purpose of encapsulation of processing and storage.

A capsule is a subset of the resources of a node. An example of a capsule is a UNIX heavyweight process.

Cluster

A cluster is a configuration of basic engineering objects forming a single unit of deactivation, checkpointing, recovery and migration.

Basic engineering object

A basic engineering object requires the support of a distributed infrastructure. Basic engineering objects are grouped together in a cluster. Basic engineering objects have an

engineering interface which is bound to an engineering interface of another basic engineering object in the same cluster or to a channel. The basic engineering objects are always bound to the nucleus.

The mapping of a computational object on a set of engineering objects show how a computational object can be distributed geographically.

2.3.3 Channel

The only concept not explained from figure 2.4 is a channel. Apparently, channels can be bound to cluster managers, capsule managers and basic engineering objects. A channel can cross the boundary of a cluster, capsule and even a node.

A channel is a configuration of stub, binder, protocol and interceptor objects providing a binding between a set of engineering objects, through which interaction can occur (figure 2.5). The purpose of a channel is to support distribution transparent interaction among basic engineering objects.

In this thesis the channel is most important. In the channel all the mechanisms can be found to get reliable applications. This will be explained in detail in chapter 5.

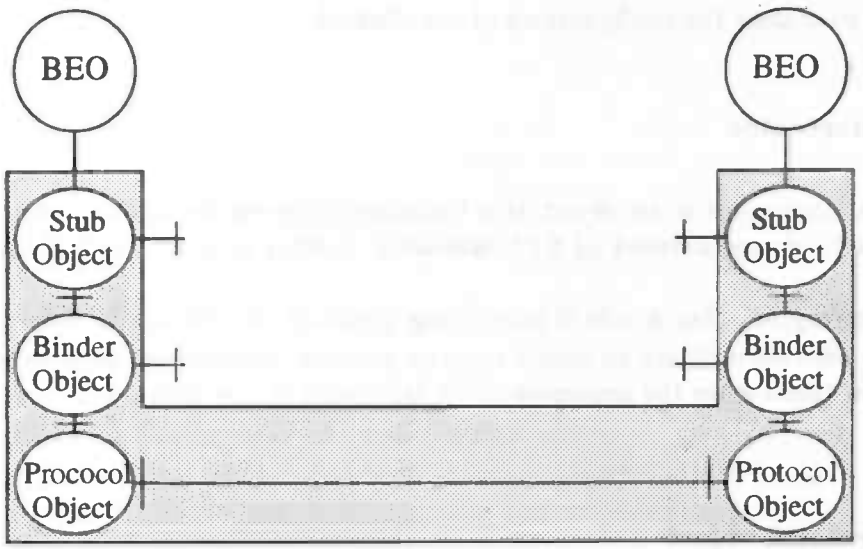

Figure 2.5: An engineering channel

Stub

A stub is an object which provides conversion functions for data exchanged between two or more BEO's.
For an operational channel stub objects provide wrapping and coding functions for the parameters of an operation. This means that the parameters of an operation invocation are presented to the binder object as a sequence of bytes. With an operation termination, the binder presents this sequence of bytes to the stub, that will unwrap en decode the result. Wrapping and coding is also referred to as marshalling.

A stub can interact with other objects outside the channel through its control interface. This interface can be used to negotiate about the data presentation the stub object has to provide.

Binder

A binder is an object which maintains a binding among interacting engineering objects.

A binder object manages the end-to-end integrity of a channel. It ensures that data presented by a stub object is transported to a correct stub object. A binder object also manages the quality of service of the channel.

By means of its control interface a binder can interact with objects outside the channel. This control interface can be used to obtain the location of other engineering objects or to change the configuration of the channel.

Interceptor

An interceptor is an object at a boundary between domains. A domain is, for instance, the local area network of KPN Research. A relay is an example of an interceptor.

Interceptors play a role if interacting protocol objects are in different domains. It can be used for instance to ensure security policies. Interceptors objects are not considered in this thesis since the implementation is located in one domain.

Protocol object

A protocol object communicates with other protocol objects to achieve interaction between engineering objects.

RM-ODP identifies protocol objects capable of interworking to be in the same commu-

nication domain. Protocol objects based on TCP/IP, for example, belong to the same communication domain, but do not belong to the communication domain of the protocol based on ATM. The communication between protocol objects takes place at their communication interface.

Chapter 3

Designing Open Distributed Systems

The RM-ODP is explained in the previous chapter. This chapter describes how the RM-ODP matches with a design methodology. This design methodology is used later in this thesis.
Section 3.1 presents an introduction to the design methodology. Section 3.2 describes the design phases in the design process and Section 3.3 formulates the design trajectory which is used in this thesis.

3.1 Design Methodology

The RM-ODP does not prescribe a design methodology. The following design methodology is presented in [7]. The definitions used in this section also come from [7].
A system is supposed to fulfill the needs of a set of users. The needs of a user are expressed in a set of user requirements on the properties that the real system must possess.
A *design process* can be defined as the activity in which the user needs are formulated and transformed into a real system. The process is shown in figure 3.1.

A *design methodology* can be defined as a set of methods to formulate the user requirements and transform them into a real system

The design methodology presented in this thesis is used for designing a mechanism to build reliable applications in a complex distributed system. To design such a system in one step is almost unworkable and certainly undesirable, because it is easy to overlook some design issues.

That's why the design process is split up into a sequence of design steps, where in each step only a limited set of design decisions is considered and evaluated against design criteria and design constraints, and some of them are incorporated in the design. This is called

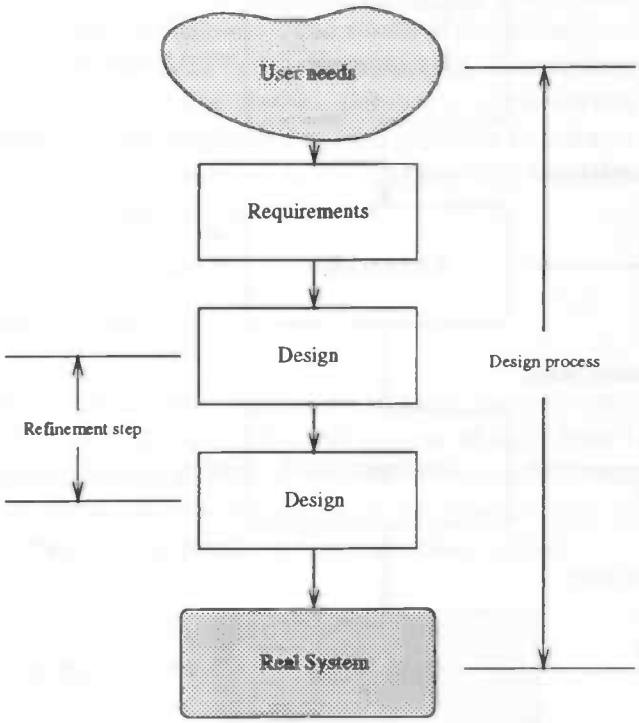

Figure 3.1: The design process

stepwise refinement or a *top-down* design method. As a consequence, each next design step in the sequence produces a more refined version of the design which is closer to the real system.

The resulting sequence of design steps, produced by stepwise refinement, that starts with the formulation of the user requirement and terminates with a design that will mapped into a real system is called a *design trajectory.* This is shown in figure 3.2.

3.2 Design Phases

Figure 3.2 also shows that a design trajectory can be divided into design phases, where each phase is characterized by different design objectives. In every phase, stepwise refinement is used. Each design step results in a design at a lower level of abstraction. This means that more characteristics are taken into account. The different phases are outlined in the next part.

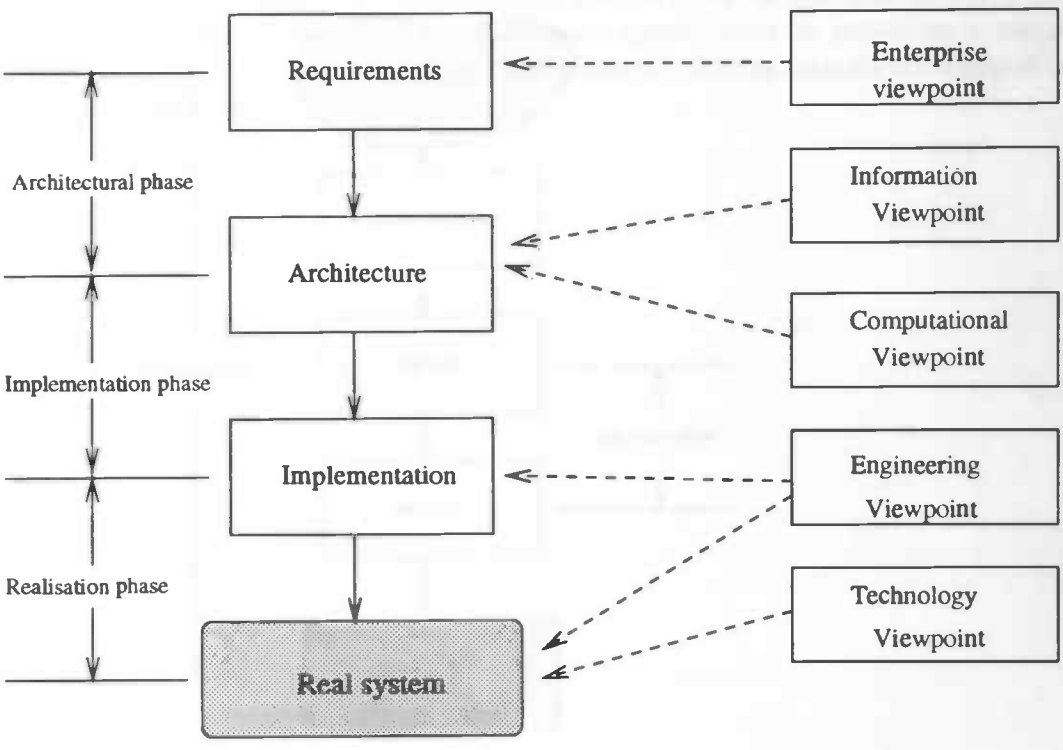

Figure 3.2: A design trajectory

3.2.1 Requirements capturing phase

In the requirement capturing phase, requirements are documented. These requirements are extracted from user needs. User needs are directed by the objectives and activities of an enterprise. The requirements on the distributed system can be derived from an enterprise viewpoint.

3.2.2 Architectural phase

The objective of this phase is to produce a design of the system that expresses the relevant properties of the real system. The result of the second phase is a service description or architecture, describing *what* properties a system should posses, not *how* these properties are achieved.

3.2.3 Implementation phase

In the implementation phase, detail is added such that mapping of the architecture onto physical and logical elements becomes possible. This phase is necessary since the mapping

is not straightforward. Usually the elements of the service description cannot be efficiently mapped onto physical or logical elements. The result of an implementation phase allows for an efficient and direct mapping onto system components. An engineering specification can be used as an implementation of the system, because it is concerned with distribution of the objects of the computational specification and provides an infrastructure that supports this distribution. The objects identified in the engineering specification can be mapped directly onto hardware and software components.

3.2.4 Realisation Phase

In the realisation phase the implementation is actually mapped onto physical and logical elements that form the real system. The real system should have the properties of the architecture and implementation phase and consist of real world components. A technology specification describes the hardware and software components that make up a real system and can therefore be used as the result of the realisation phase.

3.3 Cyclic design

The design process, as depicted in Figure 3.2, is a simplified view of reality. It suggests that a real system is designed in a strict top-down manner. In reality a system is not designed in such a way. Often a subset of the requirements is transformed into an architecture and eventually into a partial realisation system. Then the design process is entered again, but now a larger subset of the requirements is taken together with bottom up knowledge of the previous design cycle. This result in another partial realisation. This repetition of the design process is called cyclic design. In the final cycle, all requirements are taken into account and the result is called the final realisation.

3.4 Design trajectory for reliable applications

The starting point has been to build reliable applications through redundancy. This means that we have a server object that exists of multiple members (further called a replicated server object). The aim has been to define a mechanism, that enables a replicated client object to communicate with a replicated server object, in concepts defined by the RM-ODP, and to realise this mechanism with ANSAware.
In an initial attempt to realise this mechanism, a system with one client and a replicated server object has been defined and realised in ANSAware.
The next step includes the definition of a mechanism that enables a replicated client object to communicate with a replicated server object.
The user requirements will be defined in chapter 4. These user requirements were derived from myself and people who work at KPN Research.

Based on these requirements a computational specification of the mechanism is given in chapter 5. Chapter 6 refines the computational specification by providing a mapping onto engineering objects. Chapter 7 shows how these engineering objects are mapped onto hardware and software.

Chapter 4

Requirements and Architecture

The first and second phase of the design process in Chapter 3 are the requirements capturing phase and architectural phase. In this chapter both phases of the model are presented and the necessary choices are explained.

4.1 Requirement phase

The requirements of the replication mechanism are presented in this section. In Section 4.1.1 the problom domain is described. The design requirements of the replication mechanism are described in Section 4.1.2 and the implementation requirements are described in Section 4.1.2.

4.1.1 Problem domain

Computing systems are more and more intensively used in our society. Hence, more and more real-life situations depend on these computing systems. Therefore, the dependability, i.e., reliability and availability, of computing systems becomes more and more important. The dependability of the applications depends to a large extend on the dependability of computing systems, executing these applications. In many cases, reliability and availability is achieved using dedicated hardware in a main frame computing system. Current developments in processing and communication technologies enable the use of computing networks rather than stand-alone solutions. The inherent redundancy of processors and communication links of such networks provide the means to build low-cost fault- tolerant systems. Obviously, fault-tolerance mechanisms are needed to benefit from the processor and communication redundancy of the computing network.
In this chapter, we propose a generic mechanism to support the developments of reliable distributed applications based on fault-tolerance through replication.

4.1.2 Design requirements

We assume that the user requirements include strong reliability and availability requirements. Our objective is to develop a flexible mechanism that can be used to provide various levels of availability and reliability. This concept is based on replication of the engineering objects, i.e., a result is based on voting (majority) on the results of multiple objects. The number of replicas is derived from the reliability and availability user requirements and the technologies used to implement the system. In order to achieve this result, we propose the following requirements:

- 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;
- the voting mechanism should be externally adjustable, i.e., it is able to have multiple voiting mechanisms;
- dynamic creation of group members must be possible in order to have the number of group mebers of a group constant over a reasonable period of time.

In literature appear implementations of a replicated object that use a queue between the client and the server. The reason why we will not use this is that a queue is a single point of failure. Another reason is that ODP has a group function we want to use and the group function doesn't match with the queue. The reason we use the group function is that one of the constraints of the model is that it must be ODP conform.

The model must be application independent which means that existing applications can also work with the mechanism. This is the reason why the designed mechanism is put at the engineering viewpoint.

Dynamic creation of group members must be possible meaning that if we decide to remove a group member this will automatically result in the appearance of an new member within a certain amount of time.

4.1.3 Implementation requirements

The implementation is restricted by the available hardware and software at KPN Research. For the actual realisation of the model, the following implementation requirements apply:

- the model has to be realised with the use of ANSAware, version 4.1.1;
- the hardware installed are Sun Sparc workstation with SunOs 4.1.3;

- communication between the workstation is based on ethernet;
- the available protocols are based on the Internet Protocol, there is a reliable connection oriented (TCP) and an unreliable connectionless (UDP) transport protocol. IPC is used for communication between processes on the same node.

4.1.4 Fault model

A fault tolerant computing system is provided with the ability to deliver its service in spite of faults. Fault tolerant systems can be classified by the kind of fault that are tolerated. As stated in the introduction there are four means to achieve fault tolerant computing. In this section we only described the classification of fault for fault tolerance.
The classification of fault tolerance is a set of assumptions about the behaviour of fault components of the system and often called the fault model or the failure model of the fault tolerant computing system. Only the external communication behaviour of the components is considered.
There are three kinds of failures [8]:

- *crash* failures
  Within the crash failure model it is assumed that the faulty component remains silence forever. So, message sent by components are always correct. If the components fails by not sending a message, it will never send any message again. Within the crash failure model, the behaviour of faulty components is restricted compared to the other failure models.
- *omission* failure
  Within the omission failure model is is assumed that faulty components omit messages. So, message sent by components are always correct. Within the omission failure model, the behaviour of faulty components is restricted less than within the crash failure model. The behaviour is restricted compared to the malicious model.
- *malicious* (or *'Byzantine'*) failures
  Within the malicious failure model (almost) nothing is assumed about the faulty components. Faulty components may send (almost) any message. Faulty components may be 'malicious', i.e., faulty components may conspire in order to achieve overall system failure.

The replication mechanism, which we propose, is based on an omission fault model.

4.2 Architectural phase

The modeling concepts defined in chapter 3 show that the requirements (Enterprise viewpoint) is followed by the architecture (Computational Specification). A computational

specification of the model is given in this section. The specification will be realisation independent. According to the structuring rules a computational specification must describe:

- a configuration of computational objects.
- the interactions that occur among these objects.
- the internal actions of these objects.

The configuration of computational objects is described in section 4.2.1.

4.2.1 Computational objects

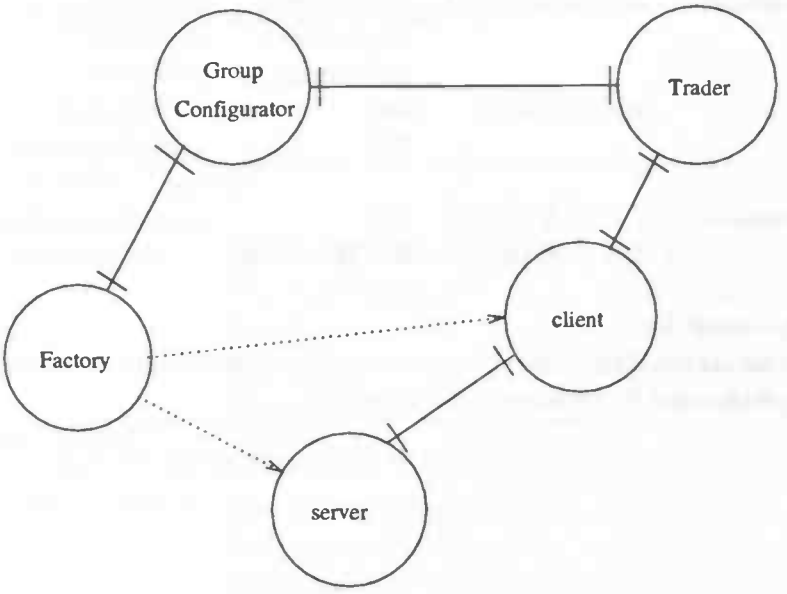

Figure 4.1: Computational configuration

In a computational configuration all the involved objects are described. These objects are shown in figure 4.1. The assignment is to specify a reliable system through redundancy, based on replication similar as the approach described in [8]. Replication is one of ODP distribution transparencies, so replication is not visible in a computational specification. The functionality of the involved objects is outlined below.

Group Configurator

The group configurator is a computational object that builds the group. If the end-user wants a server to be started and the server has to be a group server (a server that's

actually a group of replicated servers) then the user must use the Group Configurator. So the Group Configurator has a user interface. The group configurator is only used when a replicated server or replicated client is started.

Factory

The Factory is a basic computational object that provides dynamic object instantiation. The Factory is used by the group Factory to start servers and clients. Also this basic computational object is provided by the distributed computing platform. The OMG-IDL of this object is described in tabel 4.1.

Interface template *FactoryIF*
operations

Instantiate :	OPERATION [	Template:	STRING;
		Arguments:	STRING;
		Environment:	STRING;
	RETURNS [	Ref:	CapsuleRef;
		Cid:	ansa_CapsuleId];
Terminate:	OPERATION [	Cid:	ansa_CapsuleId]
	RETURNS [	BOOLEAN];	

behaviour
"An instance of this template enables objects to instantiate a new object with certain properties and to terminate the object."

Table 4.1: OMG-IDL of the Factory

Server

A Server is a basic computational object that provides a service. Replication is a distribution transparency so in the computational specification the server is one object. In the engineering specification it becomes clear that the server in Figure 4.1 is actually a group of replicated servers.

Client

A Client is a basic computational object that uses a service. Replication is a distribution transparency so in the computational specification the client is one object. Further in the engineering specification it will be shown that the client in Figure 4.1 is actually a group of replicated clients.

Trader

The Trader is a basic computational object that facilitates dynamic binding, i.e. binding at runtime, between client and server. If a server can provide a certain service at a specific operational interface, it can export its interface reference to the Trader.
In a distribution transparent environment, where a client doesn't know the location of a certain service, the Trader (which is always located at the same place) will supply the client with a reference to that service, enabling the client to access the service.

The Trader also acts as a broker between clients and services. The servers register the properties of their services to the Trader. A request from a client for a specific service is matched with the properties of the offered services and upon agreement a reference of the appropriate service is sent to the client. This basic computational object will be provided by the distributed computing platform.
The OMG-IDL of this object is decribed in table 4.2.

Interface template *TraderIF*
operations

Export :	OPERATION [	InterfaceSpecificationName :	STRING;
		NamingContext :	STRING;
		PropList :	STRING;
		Ref :	InterfaceRef;
		Capsule :	CapsuleRef]
	RETURNS [	Result];	
Import :	OPERATION [	InterfaceSpecificationName :	STRING;
		NamingContext :	STRING;
		MatchingConstraints :	STRING;
	RETURNS [	InterfaceRef];	

behaviour
"An instance of this template enables servers to Export an interface reference with certain properties in the Trader. Clients can Import an interface reference from the Trader."

Table 4.2: OMG-IDL of the Trader

New functions are also added in the Trader to handle groups. The operations *addNewGroup* and *readServiceParms* have the same function as the operations *Export* and *Import*. The OMG-IDL of these operations are described in table 4.3.

Interface template *TraderIF*
operations

addNewGroup :	OPERATION [	Resilience:	CARDINAL
		Sleep:	CARDINAL;
		groupName:	STRING;
		context:	STRING;
		propertyList:	STRING]
	RETURNS [];		
readServiceParms :	OPERATION [	gpRef:	InterfaceRef]
	RETURNS [	Resilience:	CARDINAL;
		Sleep:	CARDINAL;
		replType:	ReplType;
		svcNonce:	Nonce;
		seqNonce:	Nonce];

behaviour
"An instance of this template enables the group configurator to add an group interface reference with certain properties to the Trader (addNewGroup). Clients can read the parameters from the group out of the Trader."

Table 4.3: OMG-IDL of the extra operations added to the Trader

4.2.2 Interfaces

The supported interfaces of the previously described objects are now outlined. As described in Section 2.1.4, there are distribution transparencies involved in a computational specification . In this case the important one is the replication transparency. In Figure 4.2, the interfaces are shown between client and server in a computational viewpoint. In an interaction, objects take roles. Each object may have several roles during its life time with respect to different interactions. The role reveals something about the reason why the object takes part in the interaction. Thus we have the client role and the server role. If a object has a client role it means that this object sends a message to an object to perform an action. If an object has a server role it receives a message from an object to execute an action.

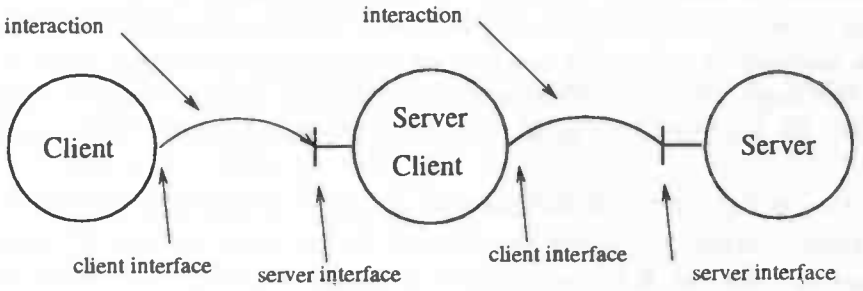

Figure 4.2: Computational interfaces

Chapter 5

Engineering Specification

The computational specification results in a realisation-independent specification. In this chapter, realisation aspects such as the distribution of components among heterogeneous systems are taken into account. A transformation from the computational specification into an engineering specification is made. The transformation from the computational to the engineering viewpoint can be seen as the implementation phase in the design process. In the engineering specification the distribution transparencies must be specified. In this thesis the mechanism to replicate server and client becomes visible at the engineering level.

Section 6.1 gives the definition and concepts of the ODP functions introduced in this chapter. Section 6.2 shows the channel that is used. Section 6.3 describes how to create a group. Section 6.4 presents how to invoke a group. In Section 6.5 we describe how to synchronize the group members. Section 6.6 defines the actions that are necessary if something is going wrong. Section 6.7 specifies a group that has a client and a server role.

5.1 ODP Concepts

Our objective is develop RM-ODP conformant systems. This means that system engineering includes the generic functionalities as defined in the RM-ODP [5]. The objective of this thesis is to investigate the suitability of RM-ODP's *replication* function for the design of fault tolerant, distributed systems. The *replication* function as defined in the RM-ODP is a special case of the *group* function. Therefore, ODP's *group* function is explained first. The group function provides the necessary mechanisms to provide the interactions of objects in multi-party binding. An interaction group is a subset of the objects participating in a binding managed by the group function. The following rules are defined by RM-ODP. 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 replication function is a special case of the group function in which the members of a group are behaviourally 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 archieves the same effect as if a member of the group had been cloned and then added to the group in a single atomic action.

The constraint that all members participate in all actions is achieved by retransmission of the operation to other members by members. This will be explained in section 6.9.2
The constraint that all members participate in all actions in the same order is achieved by a synchronisation mechanism. This will be explained in section 6.8

Replication transparency

In Chapter 2 it is stated that an ODP-system has distribution transparencies to mask distinct aspects of the system. In this chapter, one of the distribution transparencies, i.e., replication transparency, is unmasked.

Replication transparency masks the use of a group of mutually behaviourally compatible objects to support an interface.

The replication schema of replication transparency is defined as follow: A specification of constraints on the replication of an object including both constraints on the availability of the objects and constraints on the performance of the object.

Replication transparency is provided by the use of the replication function to coordinate the replication of an object to satisfy a replication schema. The replication transparency transformation includes the following steps:

- defining an object management interface supporting object checkpointing and deletion for the computational object;
- introducing a replication function;

- setting a replication policy for the cluster containing the object;
- associating a relocator with each of the object's interfaces.

5.2 Engineering configuration

The ODP engineering language describes how to structure the computational objects in order to execute them on the infrastructure. Normally a straightforward mapping of the computational objects and interfaces onto engineering objects can be made. In this case we are dealing with replicated servers and replicated clients which become visible in the engineering configuration. Other interfaces become visible and there are more engineering objects. The engineering specification describes the functionality of basic engineering objects (BEO) and additional engineering objects needed to realise distribution transparencies.

5.2.1 Replication of the server object

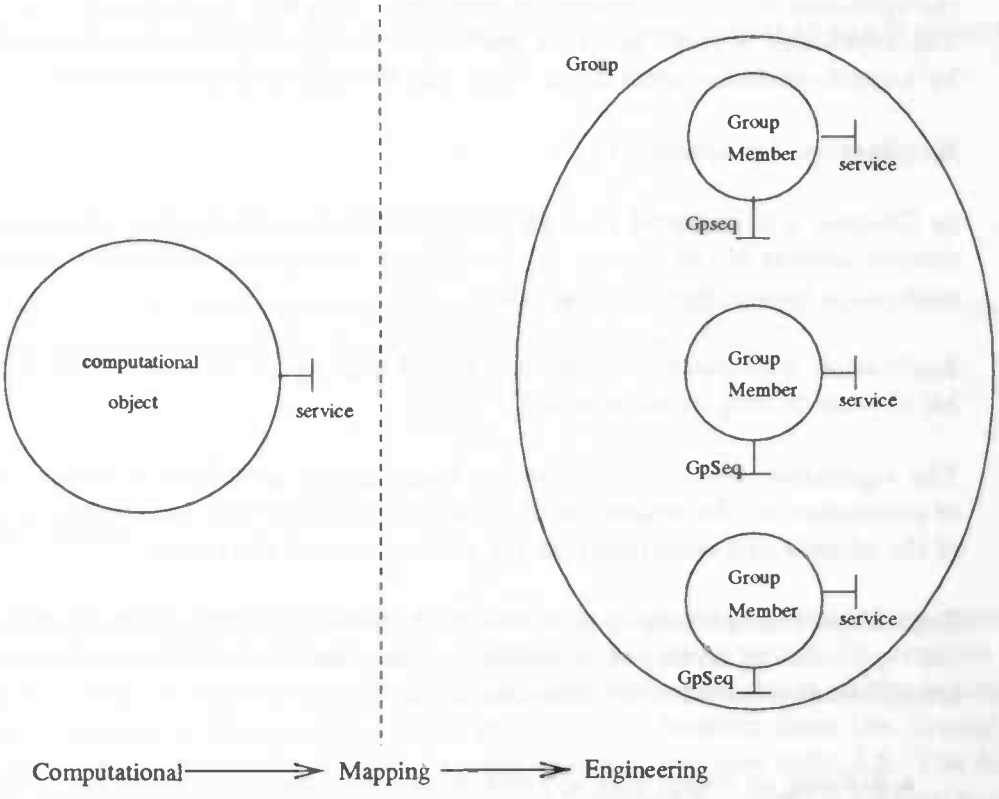

Figure 5.1: Mapping of a computational object

Figure 5.1 displays how a fault-tolerance computational object is mapped onto an set of engineering objects. A new interface is introduced. This new interface is for communication within the group. This interface is necessary to synchronise the group members. Every member in the group is exactly the same.

5.3 Group Channel

Figure 5.2 displays the mapping of computational model on an engineering model.

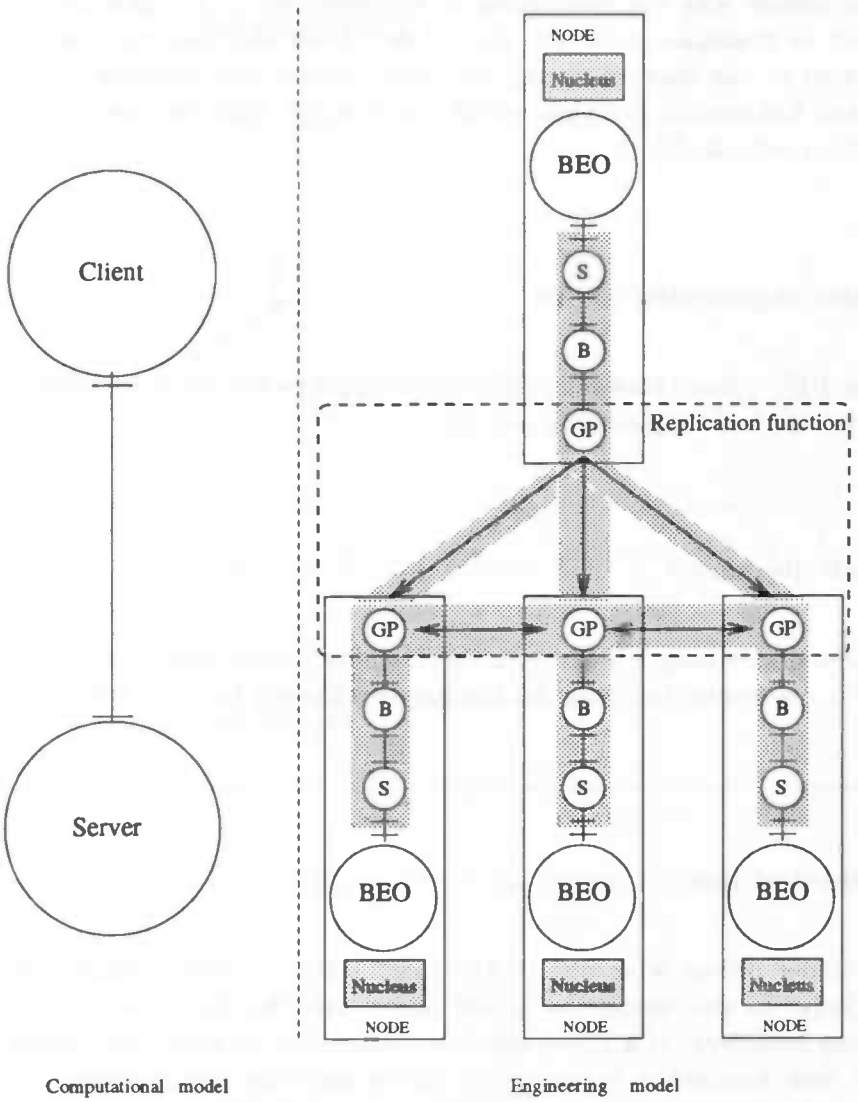

Figure 5.2: Mapping of the server group

5.3.1 Objects in the group channel

In the computational viewpoint, the server has an interface, the service interface. In the engineering viewpoint one interface is added. This interface, group sequencer (Gpseq), is a management interface used by the group sequencer to communicate with other members of the group. An engineering interface is mapped onto the creation of a stub, binder and protocol object, comprising a channel (Figure 5.2).
Sequencing and collation will take place in the protocol object. This thesis does not define a new application, but it specifies and defines a mechanisms to use replicated groups. This mechanism is mainly implemented in the protocol object.

The reason why the mechanism is implemented in the channel is that the mechanism must be transparent for the client (one of the end-user requirements). There are three objects in the channel namely the stub, binder and protocol. The stub and the binder object has specific functions which don't match with the mechanism. The only place left is the protocol object.

Basic engineering object

The BEO offers the service. The functionality of a computational object is preserved by a (set of) basic engineering objects.

Stub object

There hasn't been added extra functionality to the stub object. So the stub object provides conversion functions for the data exchanged between BEOs.

Binder object

A binder object is an object which maintains a binding among interacting engineering objects. In this model the binder object provides the service to get the location of the group members. If a group member changes its location, the binder object ensures that the new location is found. The binder gets the new location from the Trader. The extra functions added to the Trader are used in this object (see table 4.3. The function *readServiceParms* is used to support transparent rebinding if a client is holding an out-of-date group reference. If the clients copy of the interface reference is up-to-date, then the Trader is not consulted.

Group Protocol object

The group protocol object is the most important object of our proposed mechanism. In this object the mechanism is implemented to communicate with group objects. Figure 5.3 displays the configuration of the group protocol object. In the group protocol object the following functions are distinguished.

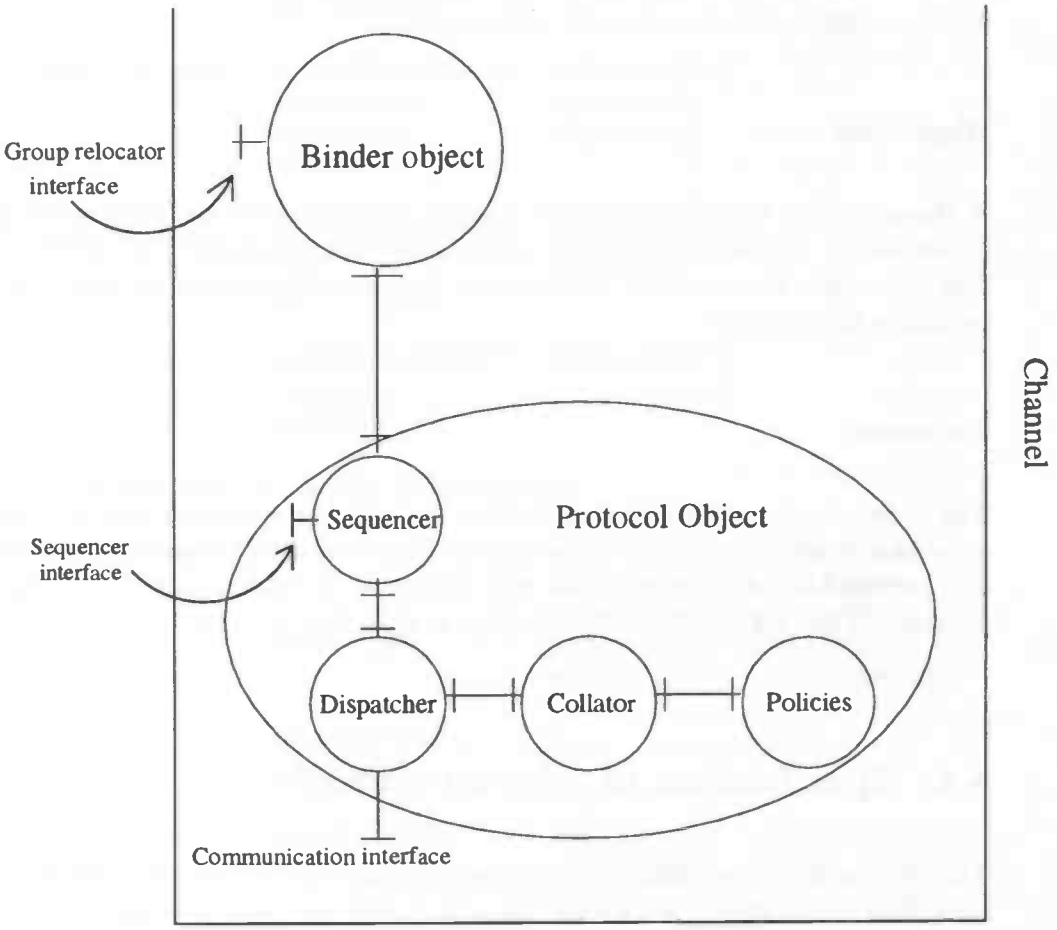

Figure 5.3: Configuration of the protocol object.

Collator

A Collator is a module which takes multiple messages from a group and produces a single message(the process of collating). This means that an application will only see one message from a group, regardless of how many members of that group send a message. There are a number of policies which could be used to collate messages (e.g. majority or

unanimous). The Collator can collate requests from many client groups concurrently, but it can only collate replies from one server group at a time.

Policies

The Policy module defines the properties that a set of messages must have to be collated e.g. that a majority all be alike. This conforms to the end-user requirements that the voting model should be externally adjustable.

Dispatcher

A Dispatcher at the client side is a module which receives incoming messages from the binder object and sends invocation submits to other members of the group.
The dispatcher at the server side receives the invocation deliveries and calls the collator to collate the messages.

Sequencer

The Sequencer is responsible for selecting the next invocation to evaluate, and communicates this to other members of the group. This function is responsible for synchronising all the members of a group. This will explained in section 6.8. The Sequencer has an interface. The OMG-IDL of this interface is described in table 5.1.

5.4 Specification of a group creation

This section describes which actions are necessary to start a replicated server object. A replicated server object is a group whose members are replicated servers. Client usage of a group is transparent.

5.4.1 Creation of a group member

A group member is created by a Factory, which is invoked by the configuration program. One of the start-up arguments to the newly created member is the target group's interface reference to enable the new member to add itself to the group. The member creates then two engineering interfaces, namely the service and the management interface, this latter is the Sequencer interface. It is possible to initialise the environment and the service when a member is created. The local Sequencer is then started and will attempt to join the group.

Interface template *SequencerIF*
operations

```
Initialise :            OPERATION [
                        Resilience :        CARDINAL;
                        Sleep :             CARDINAL;
                        GpRef :             InterfaceRef
                        mySequencerRef :    InterfaceRef;
                        myServiceRef:       InterfaceRef ]
                        RETURNS [];

RetransmitRequest :     ANNOUNCEMENT        OPERATION [
                        InvocationId :      CARDINAL;
                        requestorId :       CARDINAL]
                        RETURNS [ ];

RetransmitReply :       ANNOUNCEMENT        OPERATION [
                        msg:                reTXtype]
                        RETURNS [ ];

NextInvocation :        ANNOUNCEMENT        OPERATION [
                        Tokenlist           : GexTokenList]
                        RETURNS [ ];
```

The next operations are used for group-reforming :

```
GroupReforming :        OPERATION [         senderId:           CARDINAL;
                                            Incarnation:        CARDINAL;
                                            probeOffset:        CARDINAL]
                        RETURNS [           probeStatus:        CARDINAL;
                                            senderHWM:          CARDINAL];

Probe :                 OPERATION           [ ]
                        RETURNS             [ ];

Restart :               OPERATION [         oldIncarnation:     CARDINAL;
                                            newServiceRef       : InterfaceRef;
                                            newSequencerRef     : InterfaceRef;
                                            nextTokenSite       : CARDINAL;
                                            nextInvId           : CARDINAL]
                        RETURNS [ ];
```

behaviour
"An instance of this template enables objects to *Initialise* themself with the right properties. Every action on the service interface and Sequencer interface are executed by means of a *NextInvocation*. Whenever a group member misses an invocation the group member performs a *RetransmitRequest*. The answer is sent to him by a *RetransmitReply*. If a group has to be reformed, this action is instantiate by the operation *GroupReforming*. The operation *Probe* is used to ping other group members, i.e. check their existance. The group is restated by the operation *Restart*".

Table 5.1: OMG-IDL of the Sequencer

5.4.2 Joining the group

When the Sequencer is started it attempts to invoke the *AddMember* operation on the group service interface. If successful, the new member receives its initial Sequencer state, e.g., invocation ids and any application state. The new member installs the application state and awaits invocation.
If unsuccessful, this may be because the reference is out of date in which case another join is attempted after interaction with the relocator.
If it's the first member of the group, then it attempts to register with the relocator as a one member group and if successful carries on, otherwise it relocates and re-attempts the join.
When a new member joins the group, the relocator is also notified by the currently token holder[1] implementing the change. One of the results returned by *AddMember* is the application state. When a new member joins the group it needs the right application state. To get the right application state each member of the group takes a snapshot of its application state. These snapshots are returned to the new member for collation. If collation is successful the new member installs the snapshot and the Sequencer is initialised. At this point it has successfully joined the group.

5.5 Specification of a group invocation

At this point, the group members are created as described in the previous section. Server membership of a group is not completely transparent, but no extra actions are necessary on the application level. In this section we describe the necessary steps to invoke a group. The replicated server exist of three members. A client needs a service of a server, so the client does an invocation on that particular server that offers the service. This means that the client invokes the service interface. The server is another capsule so the client has to use to channel defined in figure 2.5. As previously discussed the mechanism to communicate with other objects is implemented in the channel. The protocol object is part of the channel.

The flow of a buffer from client to server

When a client invokes a operation on an group, first the client does an invocation submit (see for terminology Figure 2.3). The request argument is passed to the stub object where the arguments are marshalled into a buffer. The Stub passes the buffer to the binder object. The Binder object refers to the Trader to get the location of the server object. The Binder gets an answer from the Trader. At this moment the client knows that it is dealing with a replicated server. The Binder knows the location of the group members of the replicated server. The Binder passes the buffer and the location to the group protocol object. The Dispatcher in the protocol object dispatches to buffer to the server group

[1] This will be explained in Section 6.8

members.

All the group members receive an invocation deliver. Invocations arriving at the group member, pass through the Dispatcher who hands the request to the Sequencer. When the Sequencers have synchronised the invocation deliver, the Sequencer gives the request through to the Binder and Stub for unmarshalling and execution. When the request is executed all the group members send the buffer back to the client by means of a termination submit.

At the client, replies come back and get collated. When this is completed the reply is handed to the Binder and the Stub for unmarshalling and then to the client.
The interactions between engineering objects are also displayed in figure 5.4. The reason that the two arrows from Dispatcher to Dispatcher in Figure 5.4 do not hit the line is that these lines come from Dispatchers of other group members. In this figure we see the difference between a regular communication between client and server and a communication between a group member client and a group member server. The extra modules are the Sequencer, the Collator and the Policies.

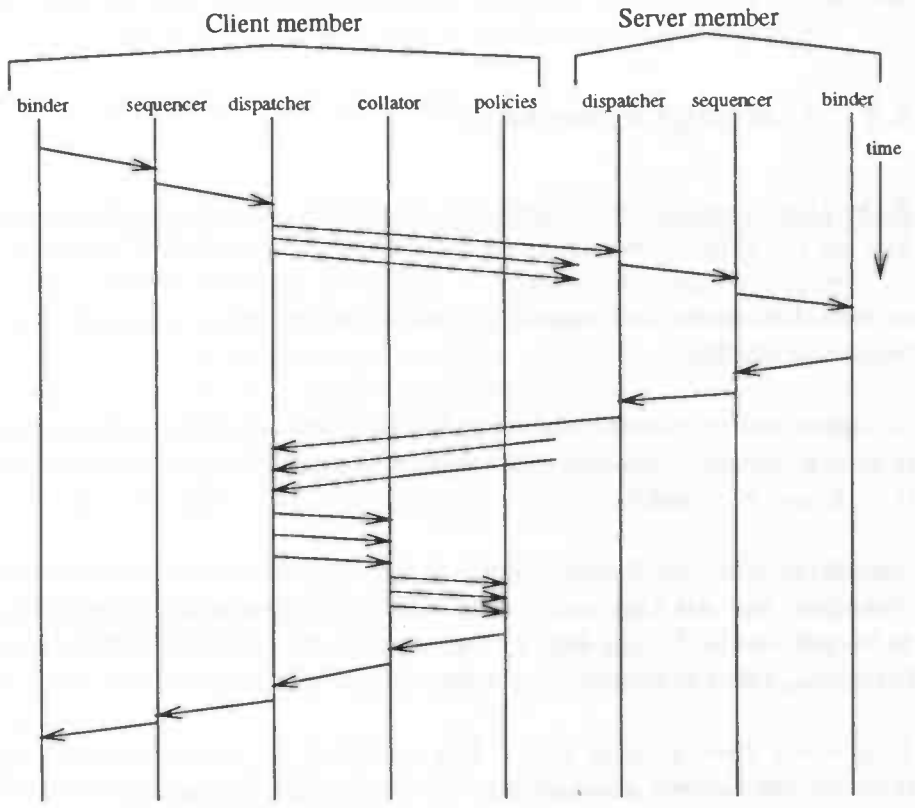

Figure 5.4: Interaction between engineering objects

5.6 The life cycle of a request buffer at the server

In order to synchronise the invocations. the invocations pass through two message lists at each server member. These are for requests awaiting evaluation (the waiting list or Wlist), and request which have been processed but which are being retained in case retransmission is required (the history list or Hlist). Retransmission is explained in Section 6.7.2.

The format of items in the entry list is:

Client id	client group id
invocation id	the invocation id assigned to this request
request buffer	the request data
reply buffer	the reply data
event counter	suspend threads until resumed by Sequencer
various flags	miscellaneous control information

After collation, a request is placed in the Wlist and in due course is assigned an invocation id by the token holder[2]. Immediately prior to evaluation, real (that is not retransmitted) invocations are moved to the Hlist where there remain until garbage collected.

5.7 Garbage collection

To be able to respond to delayed retransmission requests, previously received message are kept on the Hlist. However, to avoid consuming unnecessary resources, the Hlist needs to be regularly purged. The resources concerned are those associated with evaluated invocations: list entries and request buffers. It is convenient to synchronise this process with being token holder.

A request buffer from the client can be discarded when it has been evaluated and when it is known that no Sequencer will ask for a copy. A list entry can be discarded if its buffer is no longer required.

Associated with the history list are three invocation ids known as LWM, PWM, HWM (low, previous and high water mark). HWM indicates the most recent (or top) message to be put on the history list, PWM indicates the value of HWM last time this site held the token, LWM indicates the previous value of PWM.

Thus every time a token holder has evaluated its invocation and replied to the client it moves the current message onto the history list (updating HWM). It then purges the history list by erasing messages between LWM and PWM and updates the values of LWM and PWM. This is safe because the fact the token has circulated back to this site must mean that all messages up to PWM must have been received by all other members.

[2]This will be explained in section 5.8

Garbage collection is performed by a sweep through the Hlist every time a site acquires the token. The algorithme is described is table 5.2. This removes request buffers as soon

```
Every time this sequencer is the active sequencer and has passed on the token
it does the following:

LWM = PWM
PWM = HWM
for (each entry in Hlist) {
   if the entry's invocation id < LWM {
      if we have replied to the client {
         delete this entry it is no longer required
      } else erase the entry's request buffer if it has one
   }
}
HWM = id of last invocation evaluated
```

Table 5.2: Algorithme that is used for garbage collecting

as possible and reply buffers when reasonable.

5.8 Specification of synchronisation of group members

There are two aspects of group communication: the external communication by which a client group may invoke a server group to receive and replies, and the internal communication by which group members achieve synchronisation and sequencing. Synchronisation is necessary in order to have all group members evaluate the same expression at the same time. In this section I specify the mechanism to synchronise the group members. The Sequencer in figure 5.3 puts up the synchronisation mechanism.

Intra-group communication is based on the Chang and Maxemchuk protocol [9], with some simplifications made possible because clients communicate with a group using Remote Procedure Calls (RPC) rather than message passing. The reason is that Remote Procedure Calls establishes a reliable connection by means of retransmissions. RPC uses message passing.

The Chang and Maxemchuk protocol avoids the necessity for distributed agreement by having a distinguished member of the group, called the token holder, choose the next invocation [3] to be evaluated and assign it an invocation id. To choose the next invocation is necessary, because if there are two clients A en B (Figure 5.5) which both do an invocation

[3]This is done fairly, e.g. in fifo order

submit, it is possible that the invocation deliver do not arrive in the correct sequence at the group members. The Sequencer is meant to sequence the message so that the message are evaluated in the correct sequence. It then broadcasts a scheduling message notifying

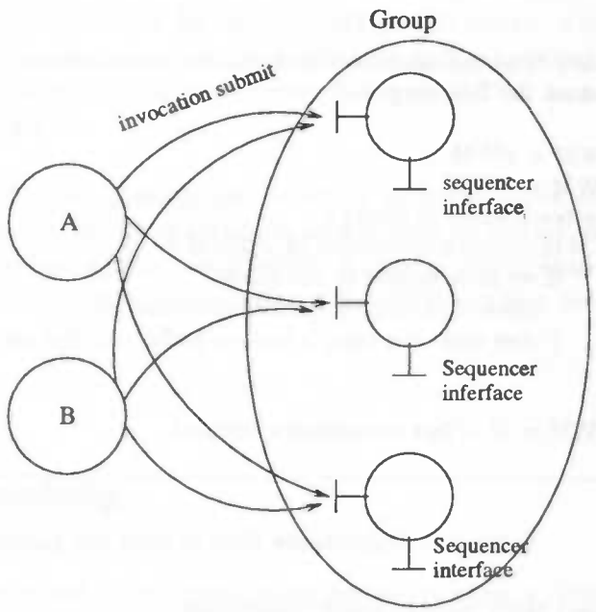

Figure 5.5: Function of the Sequencer

the rest of the group of its choice, finally the invocation is actually performed. The invocation is moved to the history list.

However, this is vulnerable to the failure of the distinguished member, and means that any message not received by it can never be selected. To avoid this, the responsibility for chosing the invocation is shared among all members. The process of notifying the next token site is called passing the token in Chang and Maxemchuk [9].

The members of a group form a logical ring. Because each member has the same view of the membership (contained within the group interface reference) and uses the same algorithm for selecting the next token holder, all members will eventually receive the token, and for any incarnation of a group each member can be assigned a unique position in the logical ring.

Intra-group messages are not sent reliably because the token site will repeat its scheduling message if it thinks that it has not been received (e.g. because it does not see the token circulating).

The token passing is done by a function called the Sequencer contained within each member. The joint aim of all the Sequencers in a group is to maintain a common evaluation schedule so that client invocations are evaluated in the same order at all members.

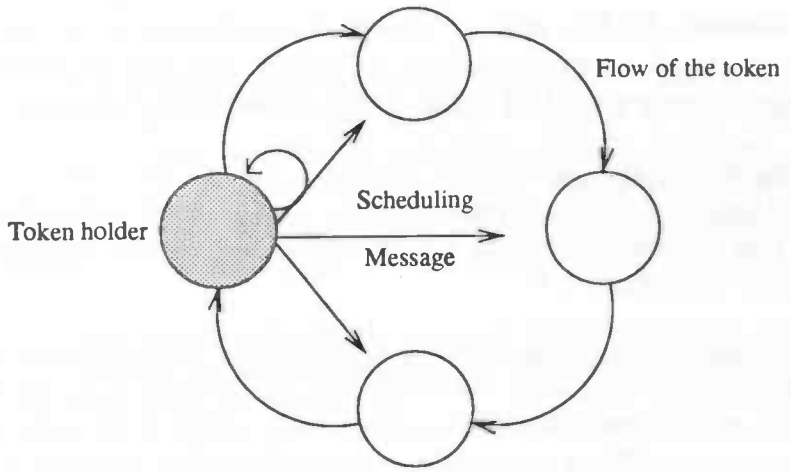

Figure 5.6: Message flow inside a group

The token holder is called the active Sequencer. Its responsibility is to choose a new invocation which is to be assigned an invocation id and added to the evaluation schedule. Also it notifies its choice to all other Sequencers and, in the process, it nominates another member to be next token site and hence the next active Sequencer. This is done using unreliable announcements, i.e. message passing.

The format of the token is:

Sender	– Senders rank in the group
Next Sequencer	– the nominated next token holder
Invocation Id	– highest invocation ID in this token
Invocation count	– number of members holding this invocation
Reply Id	– highest reply Id in this token
token List	– array of individual token entries

The format of the token entry is :

event code	– invocation or reply
Invocation Id	– it's invocation or reply id
count	– number of members holding this item
sender Id	– Id of the sender of this buffer

Missed scheduling messages can be detected by a gap in the sequence of the invocation ids (the value of HWM). However, when a member detects a gap, it does not know whether it has missed the invocation from a client, missed the scheduling message allocating it an invocation id or both. The recovery action is to attempt to maintain an identical execution history by asking the current token holder to send copies of missing messages (referencing them by the invocation ids).

It is necessary for this protocol that a site nominated to be token holder does not attempt to choose a new message to be invoked and pass the token until it possesses all preceding messages (so that it can service retransmission requests).

To cope with messages received by only a part of the group, any token holder which has nothing new to invoke passes on the token in a null scheduling message (after a delay). This ensures that so long as one member has received the message that all members will eventually evaluate it.

This combination of token passing and selective retransmission ensures that all members evaluate invocations in the same order, and so long as k (this value is defined when a group is created) members of the group receive a copy of the message, all others will eventually receive it. This is one of the constrains of the ODP function described in section 4.1.1

5.9 Error handling

In this section is described what action must be taken when a timer expires. In section 5.9.1 is described when a timer fires. If a member hasn't received the invocation deliver a retransmission has to be done. This is described in section 5.9.2. When a server does not respond on any action then this server has to be deleted of the group. This means that the group must be reformed. This is described in section 5.9.3

5.9.1 Alarms

Alarms are implemented by counters which are decremented whenever a times expires. If the decrements result in a zero value then an action procedure is invoked. The current schema uses a user-specified basic time interval and uses two counters with the following initial values:

reformGroup (50 seconds) represents an unreasonably long time to have not held the token. This counter is started when this site passes on the token. If it expires then it is assumed that some token holder has died and that the group must reform.

forceToken (2 seconds), represents the time by which some other token pass should have been observed. It is started by a token site when it passes the token and when it fires the token is retransmitted to the next token site and the timer rearmed. The timer is cancelled if another token pass is seen so that unless cancelled the timer runs contiously, eventually the **reformGroup** action will occur.

5.9.2 Retransmission

Retransmissions enable Sequencers to recover from lost or missing messages. If a Sequencer believes it has failed to pass on the token (usually because it cannot observe the token circulating) it will retransmit the token to next token site, this continues until the token circulates again or until the group is forced to reform. Immediately prior to passing the token the token holder sets an alarm; whenever a subsequent sequencing message is seen this alarm is cancelled. If the alarm fires then the token is retransmitted to the next site.

If the arrival of a token reveals a gap in the sequence of invocation ids, the site asks the token sender to retransmit the missing invocations. The site gets back a cast containing a copy of the invocation from the token site.

5.9.3 Reforming the group

The group is assumed to be functioning as long as scheduling messages arrive from token sides. The decision to reform the group therefore is made when a period has passed without a scheduling message arriving(Reform alarm has fired). This involves discovering who in the group is still functional (to recover from member failure), rebuilding the logical ring (to recover from communication failure) and establishing a consistent evaluation history (to resynchronise the group after reformation).

After reformation, members synchronise there pre-reformation execution state and all invocations sent to a previous incarnation of the group (whether assigned an invocation id or not) will be returned to the sender with an indication that the population has changed and that rebinding is necessary.

Re-establishing the current population

Reformation starts, when a site detects a failure, with a broadcast to all members of the group to perform the *GroupReforming* action. Members receiving this message are obliged to stop with evaluating any further client invocations or passing the token (so that a static picture of the group prior to reformation can be built) but must respond to any other *GroupReforming* requests. Before responding, they probe their neighbour in the logical ring that indicate whether it can communicate with its neighbour. After responding, they will also set a short timer, if this expires they will attempt to reform the group, this copes with the reforming member crashing.

To cope with a number of members simultaneously attempting to reform a member initiating reformation quotes its rank in the group and will defer if a member with lower rank sends it a *groupReforming* message. It is then the responsibility of the lower ranked member to reform the group members with which it can communicate. To ensure timeliness of the reformation attempt, the reforming member also includes the incarnation id of the

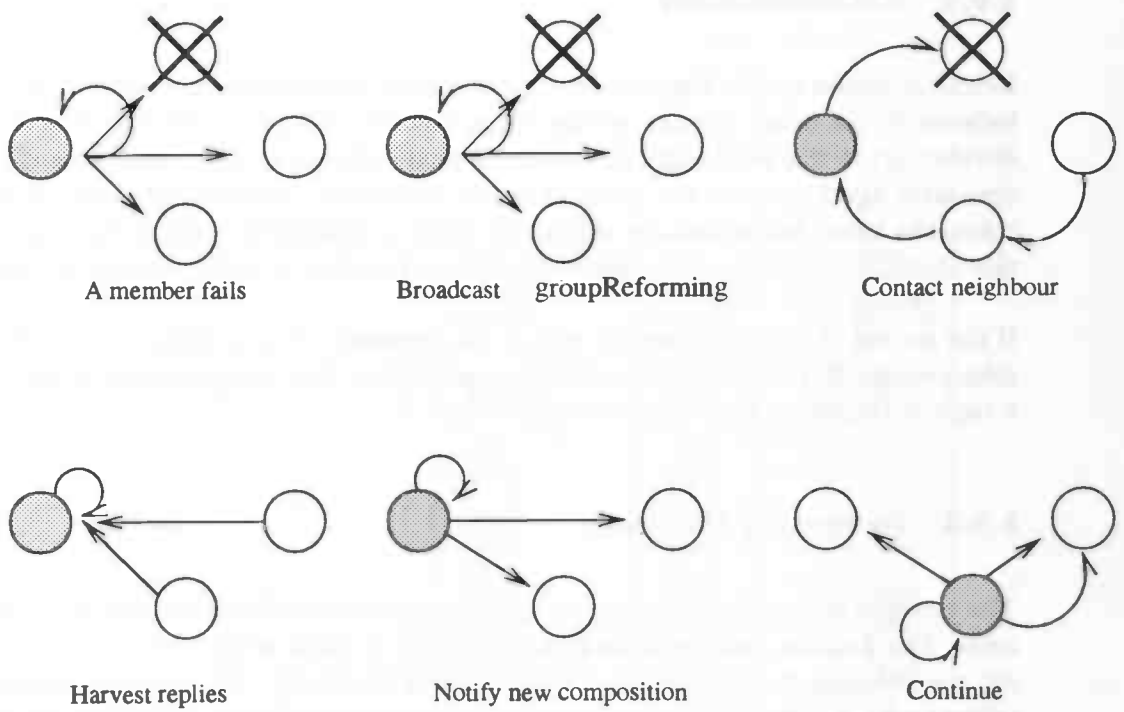

Figure 5.7: Reforming after failure of a group member

group it is attempting to reform, obsolete values are ignored.

The reforming member then builds up a new membership list, and sends a *register* message to the relocator and a *newGroup* message to all members of the newly reformed group and then restarts the token.

5.10 Group object that has a client and a server role

In the previous sections, there was one client and a replicated server. In this section will be explained what steps are necessary to deal with replicated clients. This is displayed in figure 5.8.

The extra step is: In every group server member the messages must be collated. This means that the dispatcher in the group server member gets n (replication factor) messages, gives these messages to the collate routine who comes back with one message. This message will be executed and the answer will be send to all client members.

There are two reasons to collate the invocation deliveries. The first one is to get a reliable request. If there are three invocation deliveries, then one can be wrong and still there will be a good invocation termination, because of the voting mechanism. The second one is that you want to minimize the number of messages in the network and the number of

processes in the group members. If there were not collation in the group server members the number of messages will increase rapidly.

5.11 Main difference between group object and a singleton object

This section summarizes the main differences between a group object and a singleton object.
The extra functions which are introduced to handle groups are the Sequencer, the Collator and the group relocator. The Sequencer contains the synchronisation mechanism for the group. The collator holds the mechanism to collate the messages and the group locator is introduced to contain the exact location of the members of the group. These functions are not used in singleton objects.

Group objects increase also the network traffic. For instance, assume that two group objects each exist of three members communicating with each other. If the client group does an invocation submit, nine message will be send. This is a large contrast to a singleton object which needs only one message. The same situation exists for termination submit.

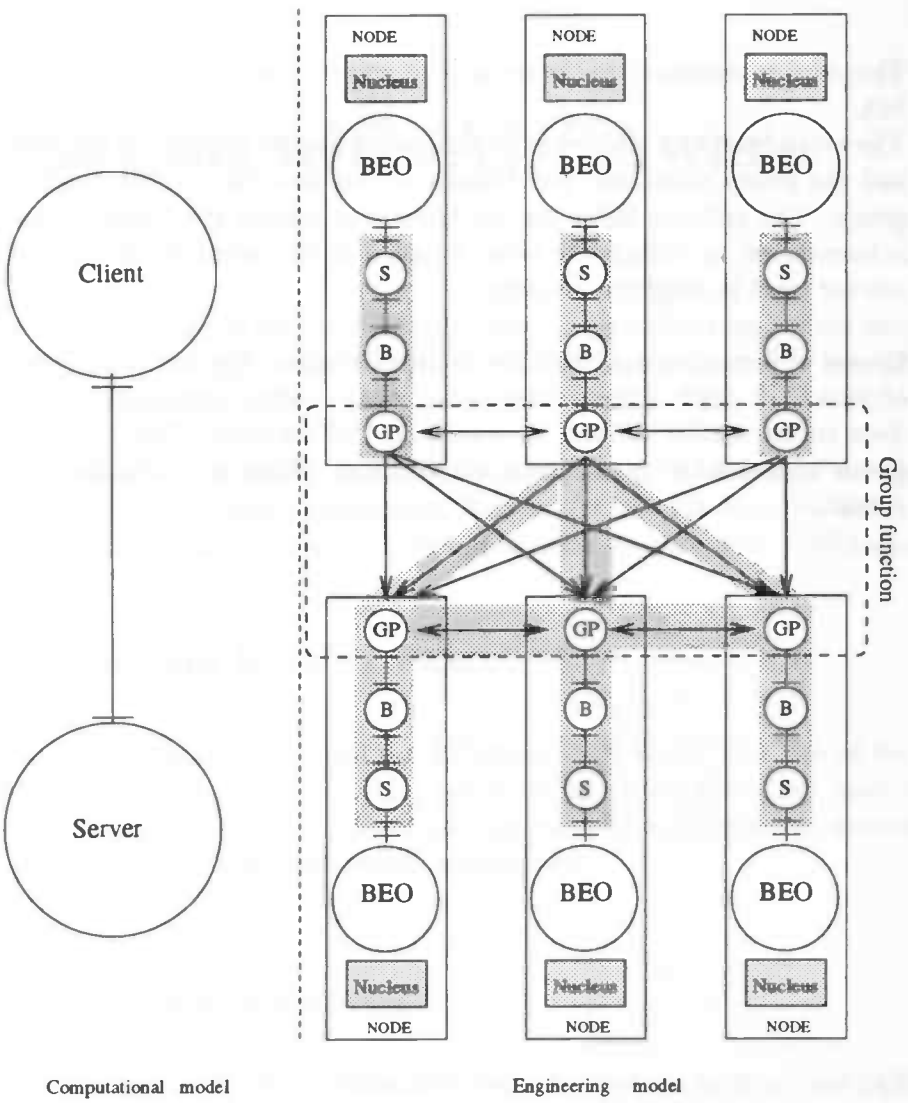

Figure 5.8: Mapping of client and server objects

Chapter 6

Realisation with ANSAware

In Chapter 5 and Chapter 6 we have modelled the group objects and other objects from the computational and engineering viewpoints. This chapter describes how ANSAware has been used to realise the model presented in these chapters.

Section 7.1 describes the background of ANSAware, the relation with ODP and how it can be used to develop distributed applications. Section 7.2 and 7.3 describe which concepts from the computational and engineering viewpoints are supported by ANSAware.

6.1 Introduction to ANSAware

The application programmers manual for ANSAware [10] states that one of the primary goals of ANSA is to "simplify the design, construction, deployment and maintenance of distributed applications". The ANSAware package provides application programmers with tools and mechanisms to build distributed applications.

6.1.1 Background of ANSAware

ANSAware is the testbench of the Advanced Network System Architecture (ANSA) [11]. This architecture aims to support "the design, implementation, operation and evolution of distributed information processing systems". The components that make up such a system (e.g. operating systems, computers and networks) may come from different vendors.

The ANSA team has developed five projections called enterprise, information, computational engineering and technology. Projections are intended to describe a system, each with the emphasis on a certain concern. These projections have been adopted by the ODP standardisation activities as ODP viewpoints.

The ANSAware testbench shows how (part of) the ANSA architecture can be implemented. It is a distributed platform, which means that ANSAware programs can run on top of different operating systems, interconnected by various networks and still interwork. Its main focus is on the computational and engineering projections.

6.1.2 Employment of ANSAware

Figure 6.1 reflects the way an application programmer realises a distributed applications. The computational specification is a starting point. From this specification, the programmer writes interface definition files and source code for the objects.

The interface definition files can be derived from the interface templates. The interface definitions are written in the Interface Definition Language (IDL).

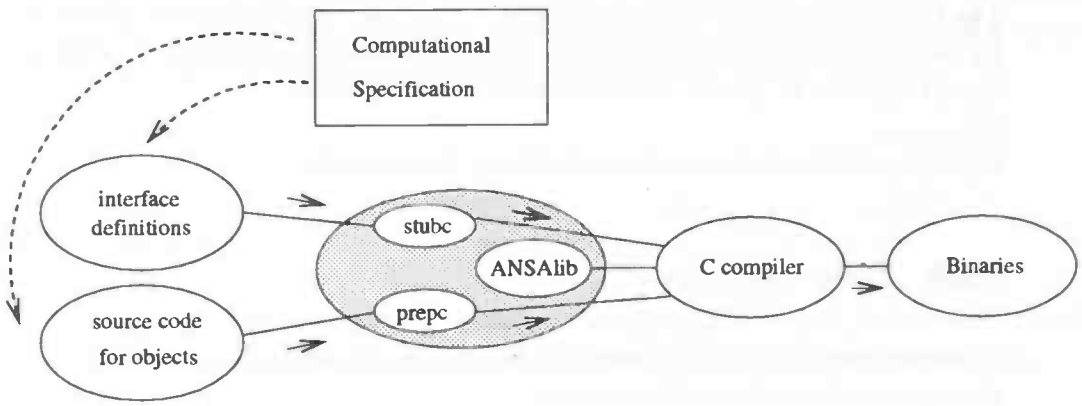

Figure 6.1: Building a application with ANSAware

The source code realises the behaviour of the objects and interfaces. ANSAware provides for a Distributed Processing Language (DPL). This DPL language is plain C-source embedded with ANSAware statements. These embedded statements are recognised by the PREPC compiler by an exclamation (!) mark in the first columm.
When the interface definition files and the source code have been written, they can be offered to the ANSAware tools. These tools transform their input to C- files, which will be offered together with a library to the C-compiler. This results in executable binaries.

6.2 Computational support of ANSAware

The computational model of ODP provides a set of concepts to describe a distributed application. Not all these concepts are supported by ANSAware. Some concepts, such as

object, operational interface, object template are supported. Other concepts such as an interface template, stream interface and subtyping are not (fully) supported.

6.2.1 Object

A computational object in ANSAware is a piece of C code with some embedded DPL statements. Careful usage of the C language enables the code to be ported to different machines, with different architecture and operating systems.

6.2.2 Operational interface

Operational interfaces are defined by means of an IDL file. This specifies the operations that the interface supports and for each operation the parameters and the result.

The C language is used to programm the behaviour. ANSAware can generate a C function for every operation in the interface. The application programmer can fill in the body of these functions to specify the behaviour of the interface.

Operational interfaces can be instantiated dynamically.

6.2.3 Object template

An object template in ANSAware is a binary file. An object is instantiated from a template through the execution of this binary.

6.2.4 Interface template

The signature of an interface is defined in a separate IDL file, but the behaviour of an interface is integrated with the behaviour of an object.

Therefore, interface templates are integrated into object templates. Is is not possible to instantiate an interface dynamically from an interface template.

6.3 Engineering support of ANSAware

ANSAware is based on a number of engineering objects. Differences and similarities with respect to the engineering model were found concerning the notions of an engineering object, cluster, capsule, node, nucleus and channel.

6.3.1 Object

Engineering objects are not easy to distinguish in ANSAware. Many engineering concepts are provided by means of a library, that is linked to the application code. This library contains a collections of functions that provide the engineering abstractions (distribution transparencies). The library is programmed in C which makes it difficult to identify engineering objects, because C is not an object oriented language.

The system programmer manual provides a good description of all the functions in the library and devides them into subsets, or modules. Such a subset of functions can be viewed as an interface to an engineering object.

6.3.2 Cluster

The notation of a cluster could not be identified in ANSAware. Activation, deactivation and migration of engineering objects is not supported by ANSAware, so there is no need to provide for clusters. This also eliminates the need for a cluster manager.

6.3.3 Capsule

An ANSAware capsule consists of a compiled computational object, linked with the ANSAware library. A capsule is mapped onto a process on the host operating system.

6.3.4 Node

A node in ANSAware is an operating system running on one machine, just as in ODP. Currently ANSAware runs on top of UNIX, VMS and MS-DOS.

6.3.5 Nucleus

ODP defines the nucleus as the operating system that runs on a node. ANSAware provides a system module, that is an interface to some of the functions of the underlying operating system. This system module can be compared to the ODP notion of a nucleus.

ANSAware also has a nucleus module, which should not be confused with the ODP nucleus. The ANSAware nucleus is not an abstraction of the host operating system, but is present in every capsule and provides multi threading facilities.

6.3.6 Channel

The channel and session modules in ANSAware are used to manage remote operation execution (REX). They keep track of the communication that takes place with other capsules. These modules can be seen as the ANSAware realisation of a channel.

All aspects of communication, such as binding, execution of threads, exchanging messages and buffering are implemented and biased for remote operating execution.

6.4 Groups in ANSAware

One of the objectives is to investigate and use the group mechanism in ANSAware. As described in Chapter 6, the group mechanism is visible from the engineering viewpoint. In ANSAware, this means that the mechanism is a library that is linked to the application.

6.4.1 Communication functionality in ANSAware

The communication facilities are provided at three levels. The highest level is at the interface of the interpreter module, followed by the interface of the execution protocol (REX) and at the lowest level there is a message passing service (MPS) (see Figure 6.2)

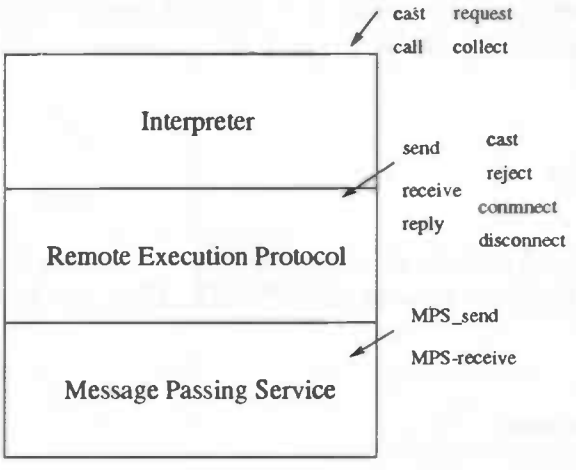

Figure 6.2: Communication stack on ANSAware

The MPS service just send messages contained in a buffer to an endpoint. Currently, message passing services are implemented on top of inter process communication (IPC), unreliable datagram protocol (UDP) and transport connection protocol (TCP). All three message passing services have the same interface. They only differ in service (e.g. UDP

provides an unreliable service and TCP a reliable service). The MPS based on IPC can only be used for communication between capsules residing on the same node.

The execution protocol adds extra functionality to the MPS. This includes fragmenting large buffers to fit the MPS and retransmission in case a message is lost.

The interpreter integrates the execution protocol with the scheduling of light-weight process or threads. The Call instruction transmits data to another capsule and waits until a reply is received. The Cast intruction just transmits data, without waiting for a reply. The Request instruction transmits data to another capsule and returns an identifier that can be used by the Collect instruction to receive the reply.

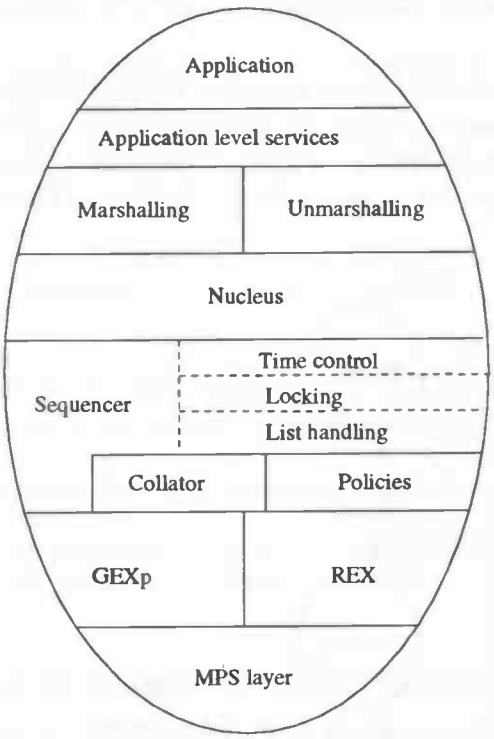

Figure 6.3: Infrastructure support for group communications

A new execution protocol is introduced, namely the group execution protocol (GEX), that is used for group communications [15]. The term GEX covers two aspects of group communications: the external communications by which a client group invoke a server group and receive and replies (GEXp) and the internal communications by which group members achieve synchronisation and sequencing. The term GEX used without qualification implies both of these aspects together.

Group facilities are provided by four infrastructure components whitin each capsule, these are concerned with both inter- and intra-group communications. These are shown in figure

6.3. The relationship between the components is not as clear cut as implied by the figure, for example, GEXp is used by applications and sequencer.

In chapter 5 the Collator, Policy and Sequencer have been defined. These object are reflected onto modules in ANSAware with the same functionality.

GEXp is the execution protocol which implements the inter-group communications. It uses REX for process-to-process buffer transport.

Application level services include the operation to join a group and to produce and install application state.

6.4.2 The group relocator

Chapter 6 describes that there are extra functions added to the Trader to handle groups. In ANSAware these functions are implemented in a new object called the gLocator. One of the reasons why ANSAware doesn't implement these functions in the Trader is that these functions are not atomic actions. In future, these functions could be added to the Trader.

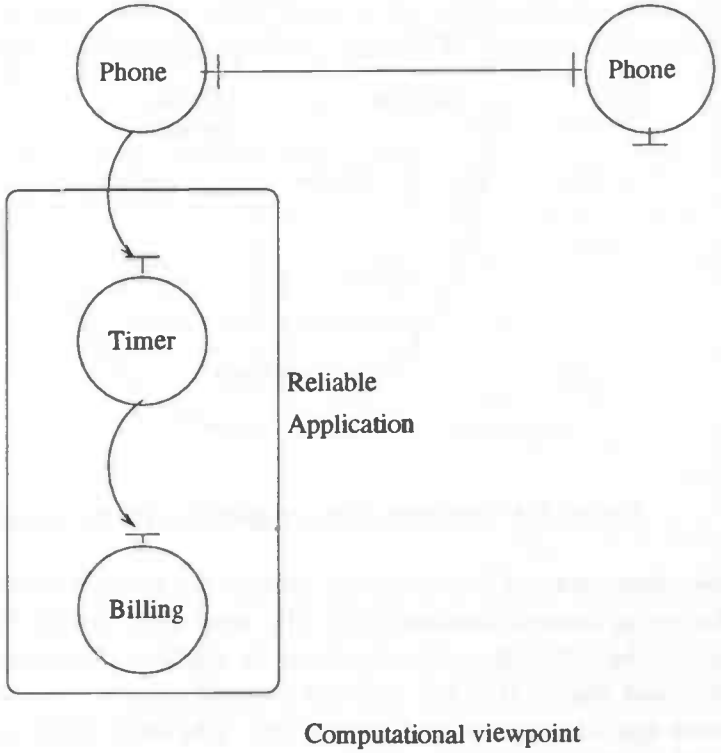

Figure 6.4: Example applications

6.5 Example Application

One of the requirements defined in Chapter 4, is that the model must be application independent. To improve our insight, we elaborate an example application. This however, can be generalised to any client/server configuration.
This example application is monitoring telephone connections. The service offered is a billing service. There are two telephones that can call each other. When they are connected a timer starts running. At the end of the phonecall it calculates how much money the telephone call costs. In Figure 6.4 the service is displayed schematically. The OMG-IDL of this interface is described in Table 6.1

Interface template *TimerIF*
operations

Start_timer:	OPERATION [	user_id :	STRING]
	RETURNS [ ]		
Stop_timer:	OPERATION [	user_id :	STRING]
	RETURNS [ ]		
Bill:	OPERATION [	user_id :	STRING]
	RETURNS [	amount :	STRING]

behaviour
"An instance of this template enables objects to start the timer.
The stop operation stops the timer. The Bill operation gives the bill."

Interface template *BillingIF*
operations

Calc_Bill :	OPERATION [	tics :	CARDINAL]
	RETURNS [	Cost :	REAL]

behaviour
"An instance of this template enables objects to calculate the bill."

Table 6.1: OMG-IDL of the Timer and Billing

6.6 Building a group in ANSAware

We will now give a step by step summary, how groups are built in ANSAware. This is displayed schematically in Figure 6.5.

1. First the gc (group configuration utility) are started.
2. The gc gives a signal to the factory with the message to create the gLocator.

3. The factory creates the gLocator and the gLocator has registered by the Trader.
4. The gc does an invocation to the gLocator to add a new group.
5. The gLocator creates a empty group.
6. The gLocator exports the group name to the Trader. The group has been registered by the Trader.
7. The gc extends the group. This means that members are placed in the group. The gc gives a signal to the Factory with the message to create the members.
8. The Factory creates the members.
9. The members perform the *Addmember* operation to add themself as a member of a group.

Now the group has been built and can be used by the client.

9. First thing the client must to do if he wants to use the server is to get the interface reference of the server of the Trader.
10. Now the client can do an invocation to the server. The group is completely transparent to clients. So the client believes he communicates with one server, but the client actually communicates with a group. In the group there can be anu number of servers.

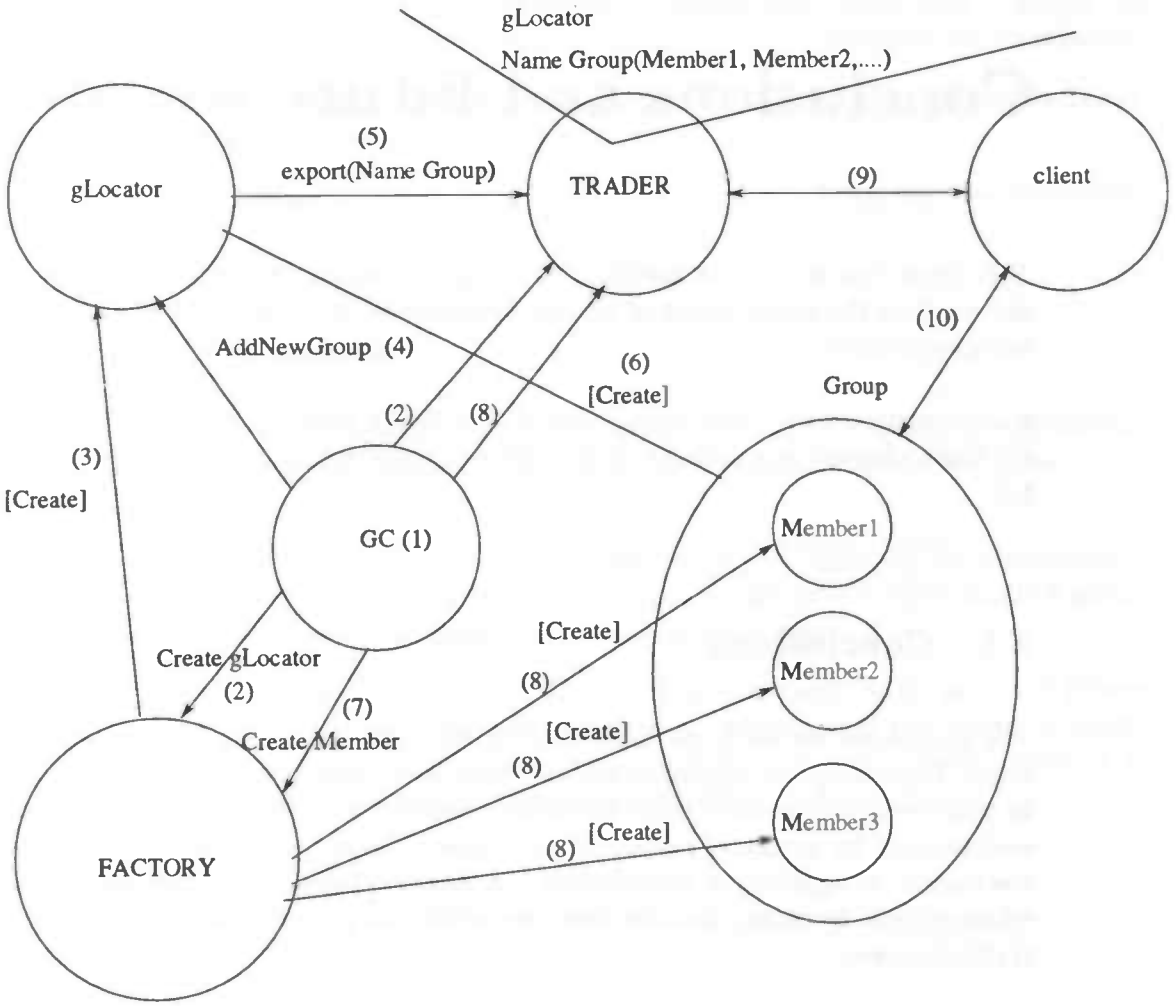

Figure 6.5: Building a group

Chapter 7

Conclusions and future research

This thesis has shown the specification, implementation and realisation of a mechanism that enables the construction of reliable applications in an open distributed environment, using replication.

In the introduction, three objectives of this thesis were outlined. Conclusions regarding these objectives are drawn in Section 8.1. Future research is summarized in Section 8.2

7.1 Conclusions

The first objective was to obtain reliability through redundancy in a distributed environment. Regarding this objective, the following conclusions can be drawn:
In this thesis the choice is made to obtain reliability through redundancy in a distributed environment by means of replicated processes. The advantage of this choice is that the mechanism is application independent. A disadvantage is that the performance of the whole system decreases, because there are additional processes in the system and network traffic increases.

The second objective was to design a mechanism that enables the construction of reliable applications. Regarding this objective, the following conclusions can be drawn:

- The intra-group communication increases the network traffic. One group object with n members produce every 2 seconds n messages. If all the objects in our system are group objects, this has a great impact on network traffic, and therefore performance of the system.
- The realisation of the mechanism in ANSAware was not a straightforward matter. Because ANSAware links all the objects to one library it is not easy to understand

the general structure of ANSAware. The lack of a good debugger has also been an inconvenience.

The last objective was to specify the solution of the previous objectives according to concepts of the Reference Model of Open Distributed Processing. Regarding this objective, the following conclusions can be drawn:

- The Reference Model of Open Distributed Processing was a great help to design the mechanism. It describes which functions are necessary to construct the mechanism.
- It is our decision to implement the replication mechanism onto the channel functions of ODP. It is an engineering specification.
- There was no need to define extra functions or concepts to specify the mechanism in ODP.

7.2 Future research

- The intra-group communications could be expanded in order to handle the Byzantine failure model. (e.g. in Chapter 7: how does the replicated timer-server agree on the start-time?)
- At this moment the replicated object must be started manually to cummunicate with a replicated object. The perfect situations would be if a client needs a group server, the members of the group are created at this time.
- The gLocator defined in Chapter 7 and the Trader are still single points of failure in the system. In order to solve the single point of failure of the Trader it might be possible to make a group trader, which is replicated itself. Future research is necessary on how this should be achieved, in an ODP conform manner.

Bibliography

[1] A. Avizienis and J.-C. Laprie. Dependable computing: from concepts to design diversity. Proceedings of the IEEE, 74(5):629-638, May 1986

[2] Separation of IN-functional architecture, IN-application programming interfaces and the impact on SIBs. Temp Doc no. 75 for ETSI STC NA6, Paris, 4-8 March 1991

[3] Project JTC1.21.43. Reference Model for Open Distributed Processing. Recommendation X.902; Part 2: Descriptive Model ITU/T X902-ISO 107046-2, ISO/IEC, February 1994. Draft International Standard

[4] Project JTC1.21.43. Reference Model for Open Distributed Processing. Part 1: Overview, ISO/IEC, July 1994

[5] Project JTC1.21.43. Reference Model for Open Distributed Processing. Part 3: Architecture, ISO/IEC, february 1995

[6] S. Rudkin. Templates, types and classes on open distributed processing. *BT Technol J* ,11(25), July 1993

[7] G. Scollo, C.A. Vissers, H. Kremer and L. Ferreira Pires. Protocol ontwerp. Technical Report 214007, University of Twente, January 1992

[8] L.J.M. Nieuwenhuis. Fault Tolerance through Program Transformation. PhD Thesis, Twente University, October 1991

[9] Chang, J. & Maxemchuk, N., F., Reliable Broadcast Protocols. ACM Transaction on Computer Systems, 2 (3), 251-273, August 1984

[10] Architecture Projects Management Ltd. ANSAware 4.1 Application Programming in Ansaware, February 1993. Document RM.102.02

[11] S. Mullender, editor. Distributed Systems. Frontier Series. ACM Press, New York, New York, 1st edition, 1989

[12] Architecture Projects Management Ltd. An overview of ANSAware 4.1, February 1993. Document RM.099.02

[13] Architecture Projects Management Ltd. ANSAware 4.1 System Manager's Guide, February 1993. Document 100.02

[14] Architecture Projects Management Ltd. ANSAware 4.1 System Programming in ANSAware, February 1993. Document 101.02

[15] E. Oskiewicz, N. Edwards. GEX Design notes. Architecture Projects Management Limited, Cambridge, United Kingdom, September 1992

[16] A.T. van Halteren. Realisation of a Multimedia Stream Object in ANSAware. Master's thesis, University of Twente, Enschede, The Netherlands, May 1994