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

Department of Computing Science

A Reference Model for Open Distributed Storage Architectures

M.A. Nankman

begeleiders: Prof.dr.ir. L.J.M. Nieuwenhuis
Prof.dr.ir. L. Spaanenburg

July 18, 1995

RuG

Abstract

In many distributed telecommunications applications, the Quality of Service is largely determined by the performance and reliability of the distributed storage system. In this thesis, a reference model for distributed storage architectures is presented. This reference model is specified in conformity to the Reference Model of Open Distributed Processing (RM-ODP), the ISO/ITU-T standard. The reference model for distributed storage architectures is based on the basic architectural alternatives: fragmentation and employment of redundancy. These architectural alternatives basically control the performance and reliability, i.e., performability, of a distributed storage architecture. The reference model for distributed storage architectures can be used for an integrated analysis of performability, and validation of performability models through implementations in an open distributed environment like TINA-DPE or ANSAware.

Summary

In this thesis OSI's Reference Model of Open Distributed Processing is used to present a specification of a *Distributed Storage System* in an open distributed environment. The result is a *reference model* for different implementations of distributed storage architectures using the *basic architectural alternatives*: *fragmentation* and *employment of redundancy*. This model can be used for an integrated analysis of performance and reliability, i.e., *performability*, using performability models. These performability models can be validated through implementations in an open distributed environment, for instance TINA-DPE or ANSAware.

Preface

This master's thesis is the result of my graduation assignment for the Computer Science department of the faculty of Mathematics and Informatics of the university of Groningen (RuG). The assignment was performed from November 1 1994 through May 31 1995 at the Communication Architectures and Open Systems (CAS) department of the research laboratory of the Royal Dutch PTT (KPN) in Groningen.

Acknowledgements

I have encountered several problems during my assignment. I could never have solved these problems without the help and support of a number of people I closely co-operated with.

First of all, I would like to thank Bart Nieuwenhuis (KPN Research and RuG), Leonard Franken (KPN Research), and Ben Spaanenburg (RuG), the members of the graduation committee, for their support and help. Their comments and reviewings had a positive impact on the final quality of this thesis.

Next, special thanks to all people who have showed special interest in my work, and also reviewed and commented upon my thesis. Their names are Aart van Halteren, Iko Keesmaat, and Irene Kwaaitaal (all working at KPN Research).

Working at KPN's research laboratory was a great experience. The CAS department is an interesting and stimulating environment to work at. The people working at this department were very interested in what I was doing and were always willing to answer all of my questions about their experiences on subjects that related to my assignment. Therefore, I would like to thank all people working at the CAS department.

Also, many thanks to my family and friends for supporting me and showing interest. Last but certainly not least, I would like to thank my girlfriend who also was a great support during my graduation assignment. Thanks Ilse!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Broadband communications networks and switching systems as well as low-cost, high performance personal computers and workstations enable the growth of distributed applications in a wide range of areas. These developments are of great interests for telecommunications service providers, public network operators, and users of the telecommunications services, because the Quality of Service (QoS) of infrastructures for sharing and distributing information is improved while the costs of communication hardware go down.

Almost all distributed applications include in one way or the other a distributed database or storage system. In many of these applications, the performance and reliability of this storage system determines to a large extend the QoS experienced by the service end-users.

The QoS of such applications consists of requirements for the performance, the reliability and the consistency of an application. For each application, different QoS requirements can be made. Thus, the demands made upon the performance and reliability of the distributed storage system used by these applications can be very diverse.

An example of an application using a distributed storage system is a Video On Demand (VOD) Service. This application makes very high demands upon the performance of the underlying storage system. Many streams of video information are simultaneously transmitted at a constant, high speed. The storage system should be able to deliver large amounts of data at a high speed. Less high demands are made upon the reliability of VOD server, because occasional bit errors causing little noise or flicker are acceptable. The transactions requested from a VOD service mostly consist of read operations. Read operations executed in parallel cannot violate the consistency of stored data. Therefor, a VOD service only requires weak consistency, i.e. locking of data items is not required and the results of a write operation are not required to be measurable immediately after this operation has completed.

Another example of an application using a distributed storage system is a Banking Service. The users of this application can make orders to transfer money from their accounts to other accounts or request their account status. In this case, the storage system should be very reliable. Errors in stored data are inacceptable. However, the performance require-

ments are less strict compared to the previous example. Information does not have to be transmitted at a constant speed and small delays while executing orders are acceptable. The transactions requested from a banking service consist of a mixture of read and write operations. Transactions with write operations can violate the consistency of data items when executed in parallel with other transactions.

The distributed storage system for a banking service should satisfy strong consistency requirements. Hence, certain invariants on multiple data items need to be maintained at all points of time and data stored during a write operation must be accessable through read operations executed immediately after this write operation.

These examples will be discussed with more detail in Chapter 2 and demonstrate that the requirements for the distributed storage system in distributed applications differ with respect to performance, reliability and consistency.

1.1 Objectives

The main objective of this thesis is to develop a reference model for distributed storage architectures. This reference model should provide the basis for an integrated analysis of performance and reliability, i.e., the performability, of distributed storage architectures in conformity to the reference model.

Many research activities focus on the design and specification of distributed systems. These research activities resulted in the definition of an International Standard for a Reference Model of Open Distributed Processing (RM-ODP) [9, 10, 11, 12]. The RM-ODP is a standardised framework, providing rules and concepts for the specification of distributed systems at different levels of abstraction. The reference model for distributed storage architectures must be specified in conformity to the RM-ODP.

1.2 Problem definition

The above mentioned Video On Demand and Banking Service examples show that distributed storage architectures differ with respect to performance and reliability, i.e., performability. Generally, a distributed database architecture will be based on data fragmentation and data replication to achieve various levels of end-user Quality of Service (QoS), e.g., performance and availability. The reference model for distributed storage architectures should be sufficiently generic to support architectures based on various degrees of data fragmentation and data replication, that supports various distributed storage architectures for a range of application areas. The objective is to derive a parameterised performability model from the reference model, where the parameters relate to characteristics of the architecture and the implementation technologies. The architecture parameters are based on the degree of fragmentation or replication. The technology parameters relate to, for example, the speed of processors or disk access times. The performability

model must be capable of predicting the end-user QoS for a variety of architectures and implementations.

In order to achieve the above mentioned goals the following questions need to be answered:

- What are the basic architectures and implementations for distributed storage architectures?
- How can these architectures and implementations be modelled using the RM-ODP?
- How can a performability model be derived?

1.3 Scope & Approach

The scope of this thesis is a delimited area within the united problem spaces of Open Distributed Processing (ODP), Performability Modelling (PM) and Distributed Storage Architectures (DSA) (see Figure 1.1).

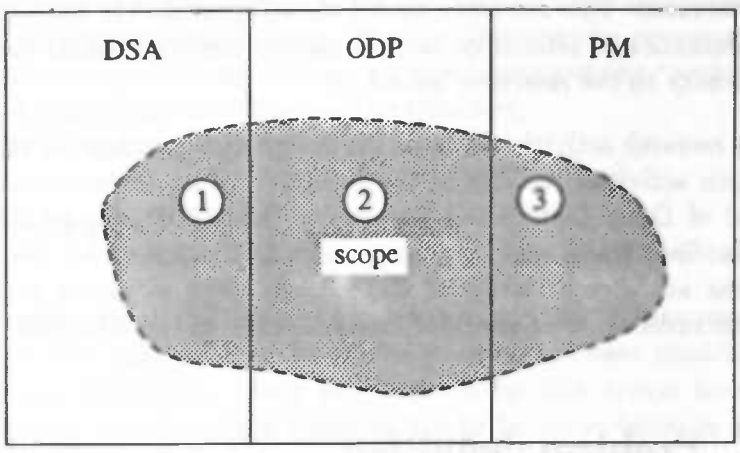


Figure 1.1: Scope of this thesis

The following approach is used to answer the three questions of this thesis's problem definition. To answer the first question, we need to explore different distributed storage architectures in order to find the common (i.e. basic) architectural alternatives for these architectures.

To answer the second question, a profound study of the RM-ODP is required. We need answers to the secondary questions "how are systems modelled in the RM-ODP?", and "what modelling concepts are available in the RM-ODP?".

To answer the last question of the problem definition, a profound study of Performability Modelling is required. The secondary questions that need to be answered here are "how

can performability be modelled?", and "how can performability models be obtained from existing implementations of systems?".

1.4 Thesis structure

This thesis can be roughly divided into three parts: a general part, a specification part, and a final part.

The general part begins in Chapter 2. In this chapter, different distributed storage architectures are explored in order to find the basic architectural alternatives that have to be modelled in the reference model for distributed storage architectures. Next, Chapter 3 introduces OSI's Reference Model of Open Distributed Processing (RM-ODP), and finaly, Chapter 4 introduces a suitable design methodology, derived from the RM-ODP, for the design of Open Distributed Systems.

In the specification part, the basic architectural alternatives found in Chapter 2, are specified using the modelling concepts available in the RM-ODP. The result of this is a reference model for distributed storage architectures specified in conformity to the RM-ODP. First, an introduction to the specification languages used in this part is given in Chapter 5. In the next three chapters, i.e., Chapter 6, 7, and 8, the requirements, the computational model, and the engineering model of the reference model for distributed storage architectures are specified. Finaly, in Chapter 9, the engineering model, as specified in Chapter 8, is specified in more detail using ITU-T's Specification and Description Language (SDL).

The final part of this thesis includes directions for future research. In Chapter 10, a brief introduction to Performability Modelling is given in order to give some directions for an answer on the third question of the problem definition of this thesis (see Section 1.2). Finaly, in Chapter 11, the conclusions of this thesis and indications for future research are presented.

Chapter 2

Distributed Storage Architectures

In this chapter, different distributed storage architectures, and applications of distributed storage architectures are explored in order to find the basic architectural alternatives for distributed storage architectures.

The first two sections of this chapter discuss the architectural techniques of Disk Array architectures and Distributed Databases. Finally, this chapter presents the basic architectural alternatives for distributed storage architectures, and the "architecture space" in which distributed storage architectures can be classified.

2.1 Disk Arrays

Disk arrays were proposed in the 1980s as a way to use parallelism between multiple disks to improve aggregate I/O performance. The driving forces that have popularised disk arrays are performance and reliability. Many architectures for disk arrays have been proposed (e.g., RAID). In each architecture a trade-off has to be made between performance and reliability.

Disk arrays organise multiple, independent disks, which usually reside within the same case and are connected by some I/O bus, into a large, high-performance logical disk. Disk arrays distribute data fragments across multiple disks and access them in parallel to achieve both higher data transfer rates (throughput) on large data accesses and higher I/O rates (efficient low level disk access) on small data accesses. Fragmentation results in uniform load balancing across all of the disks, eliminating hot spots that otherwise saturate a small number of disks while the majority of the disks sits idle.

However, large disk arrays, i.e., disk arrays with many disks, are highly vulnerable to disk failures. A disk array with 100 disks is 100 times more likely to fail than a single-disk array. An MTTF (Mean Time To Failure) of 200,000 hours, or approximately 23 years, for a single disk implies an MTTF of 2000 hours, or approximately three months, for a disk array with 100 disks. The obvious solution is to employ redundancy in the form of

error-correcting codes to tolerate disk failures. This allows a redundant disk array to avoid losing data for much longer than an unprotected single disk. However, redundancy has negative consequences. Since all write operations must update the redundant information, the performance of write operations in redundant disk arrays can be significantly worse than the performance of write operations in non-redundant disk arrays. Also, keeping the redundant information consistent in the face of concurrent I/O operations and system crashes can be difficult.

For disk arrays we assume that all of its disks are capable of indicating their own failures. This assumption implies that a disk either gives correct results or no result at all. According to [15] this assumption is based on the *omission failure model*. Within this failure model it is assumed that faulty components omit results. So, results from components are always correct.

The majority of redundant disk array architectures can be distinguished based on two features:

- the granularity of data fragmentation, and
- the method and pattern in which the redundant data is computed and distributed across the disk array.

Data fragmentation can be characterised as either fine grained or coarse grained. Fine grained disk arrays conceptually fragment data in relatively small units so that all I/O requests, regardless of their size, access all of the disks in the disk array. This results in very high data transfer rates for all I/O requests but has the disadvantages that only one logical I/O request can be in service at any given time and all disks must waste time positioning for every request.

Coarse grained disk arrays fragment data in relatively large units so that small I/O requests need access only a small number of disks while larger requests can access all of the disks in the disk array. This allows multiple small requests to be serviced simultaneously while still allowing large requests to see the higher transfer rates afforded by using multiple disks.

The incorporation of redundancy in disk arrays brings up two somewhat orthogonal problems. The first problem is to select the method for computing the redundant information. The trade-off we have to make here is between minimal update times for redundant information on the one hand, and a minimal data loss probability on the other hand. Most redundant disk arrays use a simple (low cost) parity code, though some use the more expensive Hamming or Reed-Solomon codes.

The second problem is the selection of a method for distributing the redundant information across the disk array. These methods can be classified into two different distribution schemes: those that concentrate redundant information on a small number of disks and those that distribute redundant information uniformly across all of the disks. Schemes that uniformly distribute redundant information are generally more desirable because they avoid hot spots and other load-balancing problems suffered by schemes that do not

distribute redundant information uniformly. Although the basic concepts of fragmentation and redundancy are conceptually simple, selecting between the many possible fragmentation and redundancy schemes involves complex trade-offs between reliability, performance and cost.

2.1.1 RAID architectures

The RAID (Redundant Arrays of Inexpensive Disks) organisations classify disk arrays into five levels where each subsequent level defines a finer fragmentation granularity, a less costly redundancy scheme or a more uniform distribution of redundant data ([17]). We have assumed that disks are capable of indicating their own failures. The RAID architectures benefit from this property when failures occur. In the case of a disk failure the disk array is able to reconstruct the missing data from the redundant data only if it knows exactly which disk has failed. Figure 2.1 illustrates the five different RAID levels. The levels are numbered from 1 to 5 and are described below.

In RAID level 1 disk arrays, the traditional solution, called *mirroring*, is employed. A RAID 1 disk array uses twice as many disks as a non-redundant disk array. If a disk fails, the other copy is used to service requests.

RAID level 2 disk arrays employ Hamming codes which contain parity for distinct overlapping sets of data. Data is stored in $m+n$ partitions, m data blocks and n parity blocks, and is distributed over $m+n$ disks, m data disks and n parity disks. If one of the disks fails, the original data can be computed using the remaining disks and the parity disks. The number of redundant disks is proportional to the log of the total number of disks in the system, storage efficiency increases as the number of data disks increases.

In RAID level 3 disk arrays, data is distributed over m data disks and 1 parity disk in stripes of a single bit or byte. The parity drive contains the *Exclusive OR* (XOR) of the data disks. If one of the data disks fails, the original data can be reconstructed from the remaining disks by taking the XOR of the data on the remaining data disks and the parity disk.

RAID level 4 disk arrays attempt to enhance on a RAID 3 organisation by striping the data in stripes of large blocks. This arrangement allows simultaneous multiple access to the same data volume. Each data disk contains a stripe of large data blocks, e.g., 4KB blocks. However, it also introduces a significant *write penalty*. Since a *write request* needs to rewrite parity information, it requires to read the old data, old parity and then write the new data and the new parity.. Thus slowing down the write requests considerably. Additionally, since there is only one parity drive, only one write request can be active at any time. Therefore, the parity disk becomes a bottleneck to the subsystems performance. The reliability of RAID 4 is equal to that of RAID 3 systems.

To eliminate the parity bottleneck of the RAID 4 configuration, the parity information is rotated across the disks in RAID 5 systems. This solves the parity bottleneck but the write penalty still remains.

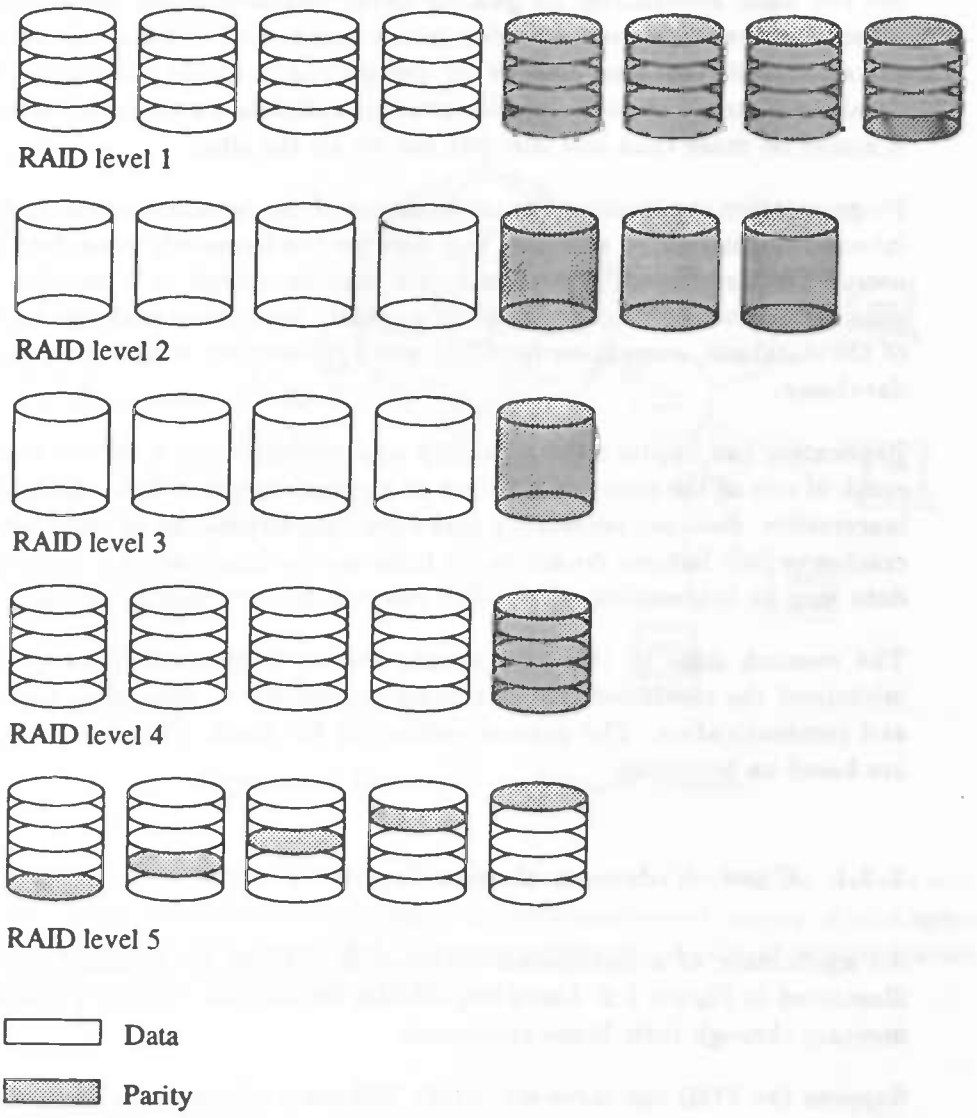


Figure 2.1: RAID levels 1 through 5. All RAID levels are illustrated at a user capacity of four disks. Disks with multiple platters indicate block-level striping while disks without platters indicate bit-level striping.

2.2 Distributed Database Systems

According to [14], a distributed database system (DDBS) can be defined as a collection of multiple, logically interrelated databases distributed over a computer network. There are two basic alternatives for placing data: *fragmented* and *replicated.* In the fragmented scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* where the entire database is stored at each site, or *partially replicated* where each partition of the database is stored on more than one site, but not on all the sites.

Fragmentation can improve the performance of the database accesses, given the parallelism inherent in distributed systems, and because the frequently used data is proximate to the users. Data retrieved by a transaction may be stored at a number of sites, making it possible to execute the transaction in parallel. Also, since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralised databases.

Replication can improve the reliability and availability of a DDBS. If data is replicated, a crash of one of the sites, or a failure of a communication link making some of these sites inaccessible, does not necessarily make the data impossible to reach. Furthermore, system crashes or link failures do not cause total system inoperability. Even though some of the data may be inaccessible, the DDBS can still provide limited service.

The research done in this area mostly involve mathematical programming in order to minimised the combined cost of storing the database, processing transactions against it, and communication. The general problem is NP-hard. Therefore, the proposed solutions are based on heuristics.

2.2.1 Case: Video on Demand

An application of a distributed database is a Video on Demand Service (VOD) and is illustrated in Figure 2.2. Users request this application to play a selected movie or documentary through their home equipment.

Suppose the VOD can serve maximally 350 users at the same time. Assume that a single video requires a transmission speed of 300 kilobytes per second. A video server would then require a performance of approximately 100 megabytes per second.

A video consists of frames that have to be displayed with a constant speed, high enough to experience smooth motion. So, VOD users make very high demands upon the performance and the availability of the system but less high demands upon the reliability of the system (occasional bit errors causing little noise are acceptable).

As illustrated by Figure 2.2, the video database is distributed over multiple sites and each site contains a complete version of the database. Each location serves a local group of users and consists of a server that is powerful enough for real-time VOD. These servers

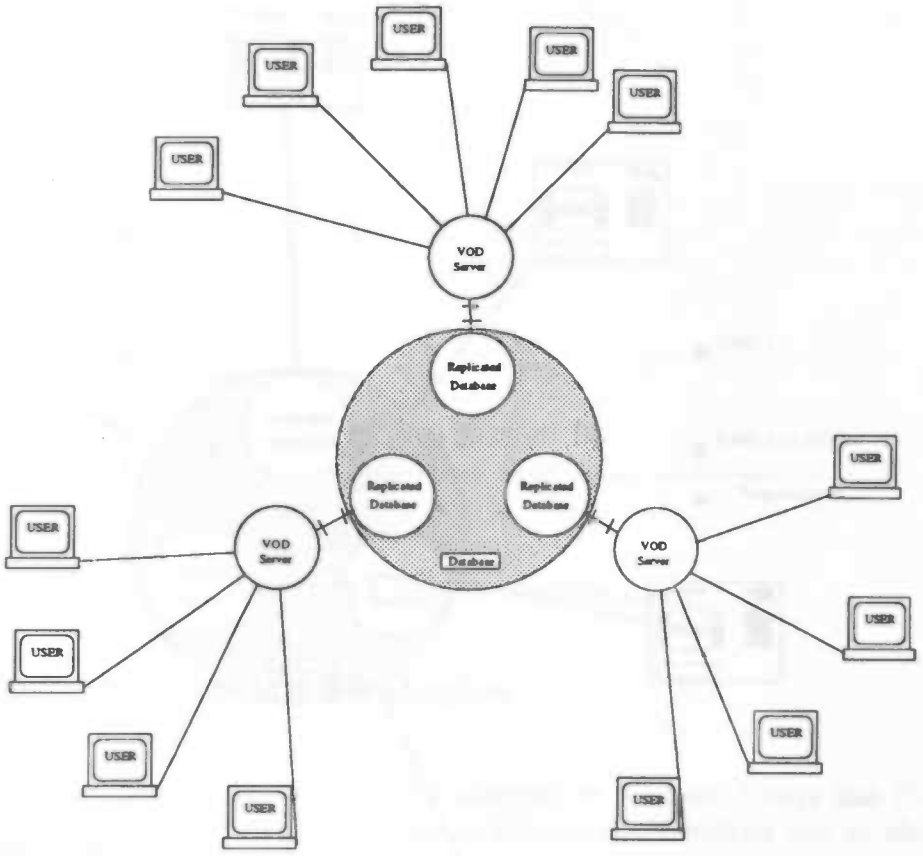


Figure 2.2: The Video on Demand System.

have direct (read-only) access to the replicated database situated at the server's site. The transactions of the VOD service executed at its database consist merely of read operations which are all executed at a single site. Write operations occur only at administrators level and do not necessarily require high performance.

2.2.2 Case: Banking

Another application of a distributed database is a Banking Service (BS). Users of this application can get their account status or draw money from their account. The BS is illustrated in Figure 2.3

Suppose there are 5,000,000 users of this application over an entire country and the average user makes five transactions per day with an average transaction size of 64 bytes. Then the average total size of the data flow through the system is approximately 1220 megabytes per day and the required system performance would then be approximately 15 kilobytes per second. Also knowing that BS users are generally very patient, you can conclude

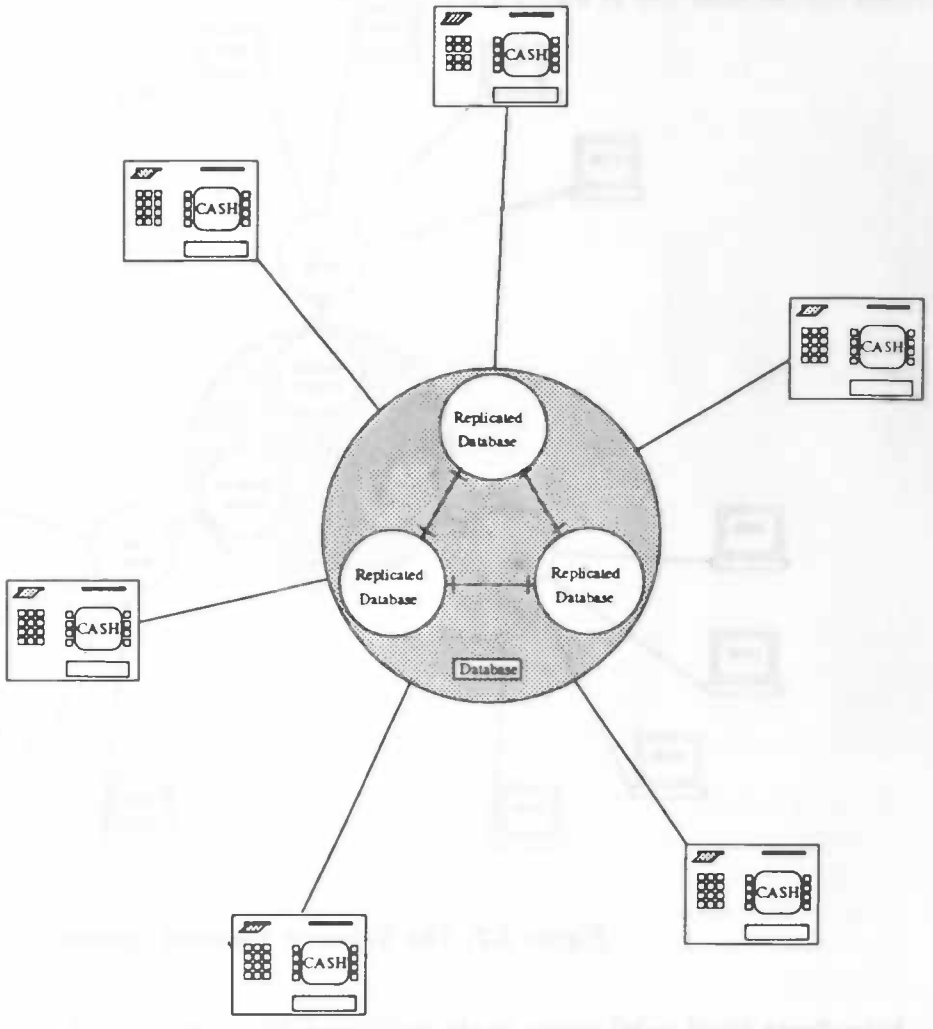


Figure 2.3: The Banking Service.

that the application's demands made upon the system's performance are relatively low. The demands upon the system's reliability, however, are very high. Commonly, BS users don't appreciate money loss caused by system failures. Therefore, the probability of data corruption or data loss should be nil.

A BS consists of a large number of "automatic teller machines" which are connected to a central database. As illustrated in Figure 2.3, this database is distributed over three sites. Comparable to the VOD service, each site contains a replicate of the database. However, the transactions to the database consist of read operations as well as write operations. Each read or write operation should be executed at all sites and the results should be compared by means of a voting mechanism in order to maintain the consistency and integrity of the data.

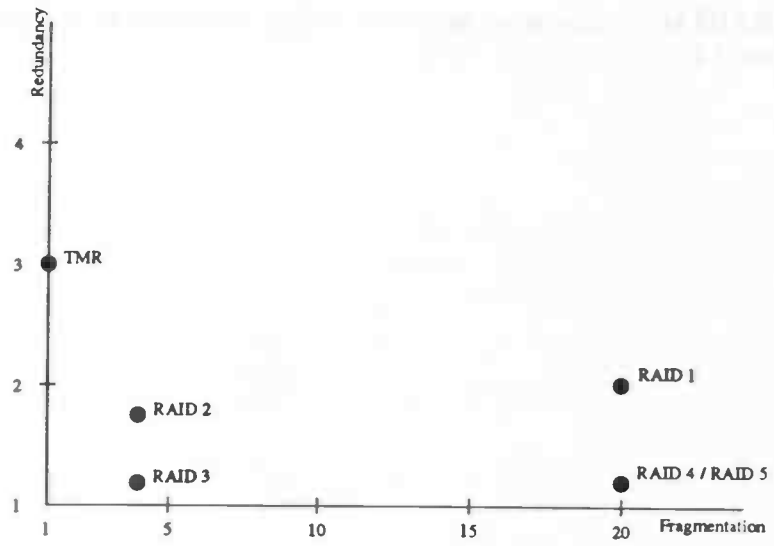


Figure 2.4: The architecture space

2.3 Implementation Strategies

In the previous sections, the architectural alternatives for Disk Arrays and Distributed Databases were explored. Apparently, two architectural alternatives can be identified to satisfy the requirements for distributed storage architecutres used by a distributed application. The first architectural alternative is *fragmentation*, i.e., the database is subdivided into a number of parts that are located at distinct physical locations. The QoS experienced by the user improves if the application can benefit from the parallel execution of multiple transactions at different locations.

The second architectural alternative is employment of *redundancy*, i.e., additional data is stored to obtain fault tolerance. An example is replication, i.e., copies of the original database are stored at distinct physical locations. Another example is to add error correcting codes to reconstruct the original data if parts of the data are lost. The QoS experienced by the user improves if the service is properly provided when the database without redundancy would have failed.

In practice, combinations of both architectural alternatives can be used, e.g. an increase of performance is achieved if replicas are accessed in parallel for simultaneous read-only transactions.

Basically, fragmentation and employment of redundancy result in different distributed database architectures. Figure 2.4 shows a two-dimensional *architecture space*, with various levels of fragmentation and redundancy. The level of fragmentation is defined as the number of logical database partitions stored at distinct physical locations. Redundancy is based on adding information r to the original information d. The level of redundancy is defined as $\frac{r+d}{d}$. An architecture based on Triple Modular Redundancy (TMR) and

the RAID architectures as described in [17] are drawn in the architecture space shown in Figure 2.4.

Chapter 3

The Reference Model of Open Distributed Processing

This chapter gives an introduction to the Reference Model of Open Distributed Processing (RM-ODP) and an overview of its structure, its modelling concepts and functions. The main objective of this chapter is providing answers to the questions "how are systems modelled in the RM-ODP?", and "what modelling concepts are available in the RM-ODP?".

3.1 Introduction

The OSI Reference Model provides a standard for the interconnection of systems. Until 1987 it was restricted to communication standards and did not go into the matter of distribution problems. This is why in 1987 the workitem Open Distributed Processing (ODP) is approved on. The Reference Model ODP (RM-ODP) provides a standardisation framework for distributed systems.

The RM-ODP provides general definitions of concepts and terms for distributed processing and a generalised model of distributed processing using these concepts and terms. The main objective of ODP is to enable the interworking between heterogeneous distributed systems and applications. The RM-ODP defines the basis for ODP standards. At the most general level, it defines a framework for distributed processing independent of the area of application.

The reference model consists of four parts:

- Part 1 [9]: "*Overview*"

  This part contains a motivational overview of ODP giving scope, justification and explanation of key concepts, and an outline of the ODP architecture. It contains

explanatory material on how RM-ODP is intended to be understood and applied by its users, who can include standards writers and architects of open distributed systems.

- Part 2 [10]: "*Foundations*"

  This part contains the definition of the concepts and analytical framework and notation for normalised description of (arbitrary) distributed processing systems. This is only to a level of detail to support the prescriptive model (part 3) and to establish requirements for new specification techniques.

- Part 3 [11]: "*Architecture*"

  This part contains the specification of the required characteristics that qualify distributed processing as open. These are constraints to which ODP standards must conform. It uses the descriptive techniques from part 2.

- Part 4 [12]: "*Architectural semantics*"

  This part contains a formalisation of the ODP modelling concepts defined in the descriptive model (part 2). The formalisation is achieved by interpreting each concept in terms of the construct of the different standardised formal description techniques (such as SDL, LOTOS, Z).

3.2 Structure of the RM-ODP

In ODP, a system is viewed from different *viewpoints*, each highlighting different aspects of the system. For each viewpoint, rules for specifying a system from this viewpoint are defined in *viewpoint languages*.

3.2.1 Viewpoints

ODP aims, among other things, at modelling open distributed systems. As illustrated by figure 3.1, distributed systems are viewed from five different viewpoints, each representing a different view on the distributed system. The viewpoints are:

- The Enterprise viewpoint
- The Information viewpoint
- The Computational viewpoint
- The Engineering viewpoint
- The Technology viewpoint

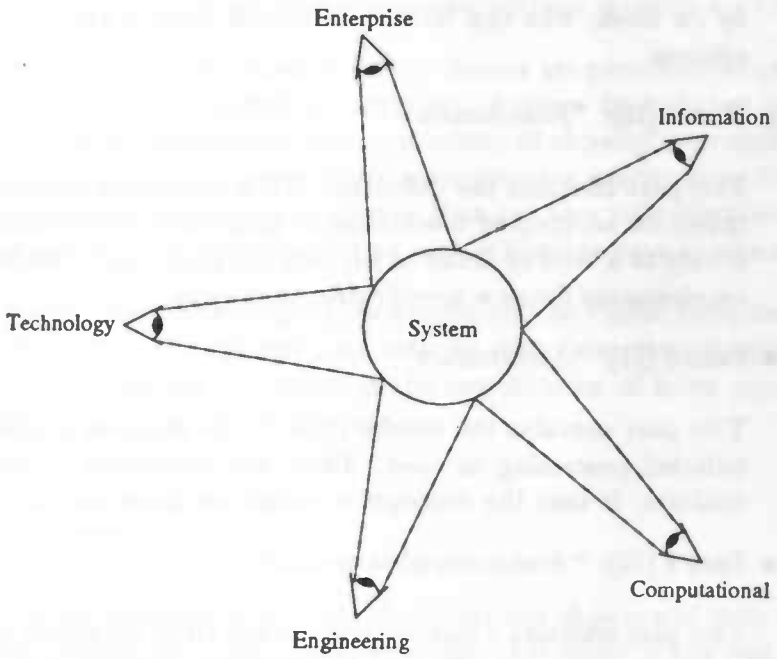


Figure 3.1: The ODP viewpoints

The purpose of the *enterprise viewpoint* is to explain and justify the objectives of an ODP system used by one or more organisations. Such a enterprise specification describes the overall objectives of a system in terms of roles, actors, goals and policies. The system is regarded as one object in a community. With each object in this community, role(s) and policies are associated. An enterprise specification dictates the requirements of the ODP system.

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

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 structures by which these interactions occur and specifies the semantics of these interactions.

By providing a functional decomposition of the ODP system, a distribution of application components becomes possible. However, the details of mechanisms required for interaction between application components are invisible in the computational specification of a distributed application. This process of hiding the effects of a geographical distribution is known as *distribution transparency.*

ODP distinguishes a number of distribution transparencies, such as location, access, concurrency, replication, failure and migration transparency. A computational specification must indicate which of these transparencies are assumed to be present.

The *engineering viewpoint* is concerned with the provision of mechanisms to enable distribution of computational objects in the computational specification of a system. An engineering specification must describe the infrastructure required to support distribution of an ODP system. It shows how objects from the computational viewpoint can be distributed geographically. Mechanisms for the distribution of computational objects and for the provision of the selected transparencies in the computational specification are described in the engineering specification. An engineering specification consist of a description of functionality and interaction between engineering objects.

The purpose of the *technology viewpoint* is to describe the physical components, both hardware and software, required for realising an ODP system. A technology specification is given in terms of technology objects. These technology objects must be names of implementable standards. Technology objects can be such components as operating systems, peripheral devices, or communication hardware.

3.2.2 Viewpoint languages

In order to specify an ODP system from a particular viewpoint it is necessary to define a structured set of concepts in terms of which that representation (or specification) can be expressed. This set of concepts provides a language for writing specifications of systems from that viewpoint, and such a specification constitutes a model of a system in terms of concepts.

Thus, for each viewpoint a language is defined for writing specifications of ODP systems. The terms of each viewpoint language, and the rules applying to the use of those terms, are defined using object modelling techniques and each language has sufficient expressive power to specify an ODP function, application or policy from the corresponding viewpoint. The purpose of a viewpoint language is to specify the set of concepts in terms of which specifications from that viewpoint must be structured in order to enable coordination and consistency with specifications from other viewpoints. Hence, any existing specification language can, in principle, be used for specifying a system from a particular viewpoint provided that those specifications can be interpreted in terms of relevant viewpoint concepts.

Enterprise language

The enterprise language contains concepts to represent an ODP system in terms of interacting *agents*, working with a set of resources to achieve business objectives subject to the policies of controlling objects.

Objects with a relation to a common controlling object can be grouped together in domains

which form federations with each other in order to accomplish shared objectives. Any such union mutually contracted to accomplish a common purpose is called a *community*.

Policies set down rules on which actions of which objects are permitted or prohibited, and also which actions objects are obliged to carry out. Actions that change policy (in that they alter the obligations, prohibitions and permissions of objects) are called *performative actions*. For example, giving a user system's administrator privileges or the creation of an object can be performative actions. Objects that are able to initiate actions have an *agent role*, whereas those that only respond to such initiatives have *artefact roles*.

Some elements visible from the enterprise viewpoint will be visible from the information viewpoint and vice-versa. For example, an activity seen from the enterprise viewpoint may appear in the information viewpoint as the specification of some processing which causes a state transition of an information entity.

Information language

An ODP system can be represented in terms of information objects and their relationships, where information objects are abstractions of entities that occur in the real world, in the ODP system, or in other viewpoints.

The information language contains concepts to enable the specification of the meaning of information manipulated by and stored within an ODP system. Basic information objects are represented by atomic information objects. More complex information is represented as composite information objects expressing relationships over a set of constituent information objects.

An information specification defines the classes of basic and composite information objects, and the activities that these objects can perform. Information objects are specified using three kinds of schema:

- static schema's,
- invariant schema's, and
- dynamic schema's.

A static schema describes the state and structure of an information object at some particular interesting situation. A static schema might be used to specify the initial state of an object, or for the state of an object at a certain moment in time. For instance, the initial state of a bank account consists of an account balance of $0 and the amount withdrawn on that day which is also $0.

An invariant schema describes some property which must always apply to the information object throughout its lifetime. For example, an invariant schema for a bank account

might specify that the balance must always be non-negative as the bank does not offer an overdraft facility.

A dynamic schema describes the way in which an information object can modify its state and structure. A bank account would require a dynamic schema for depositing money, withdrawing money, paying interest, and charging account fees. A dynamic schema might be applicable only in certain circumstances (which could be specified by the use of a static schema). For example, the dynamic schema for withdrawing $N might specify that the account balance is decremented by $N provided that the total amount withdrawn that day does not exceed $500. No dynamic schema can specify a resultant that violates the invariant constraint, i.e., only money in the account can be withdrawn.

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

Schema's for composite information objects can be composed from schema's of their component objects. RM-ODP does not require information objects to be encapsulated, i.e., schema's for composite information objects can reference the internals of their component objects. This permits the specification of such complex noun phrases as "the phone numbers of the customers with accounts that withdrew over $400 today".

Computational language

The computational language provides a small, complete set of concepts and rules that can be used to structure distributed applications. The computational language is designed to transparently cater for interactions between open distributed systems components that are remote for each other.

The computational language hides the actual degree of distribution of an application from the specifier, thereby ensuring that applications contain no assumptions about the location of their components. An application specified in the computational language is hardware independent. It might as well be implemented in a centralised environment, i.e., without distribution, as in a distributed environment. Also, the configuration and degree of distribution of the hardware on which ODP applications are run can easily be altered without having a major impact om application software.

A computational specification defines the functional decomposition of an ODP system into objects which interact at *interfaces*. These objects interact according to the client/server-model. A *client* object requests services from *server* objects. A server object provides services which are accessible through interfaces. A server object can *support* multiple interfaces which allows grouping of related services. If a client object wants certain services from a server it *requires* the interfaces at which these services are offered by the server object. In the client/server-model objects can be both client and server, allowing servers to request services from other services.

In the computational language, three types of interfaces are defined:

- the signal interface,
- the operational interface, and
- the stream interface.

All interactions at a *signal interface* are signals, i.e., one-way communication from client objects to server objects. Signals cause a server object to perform some internal action (which might cause a change of its state) without notification to the client who sent the signal.

All interactions at an *operational interface* are operations. An operation can either be an *announcement* or an *interrogation*. An announcement is an interaction between a client and a server, where the client requests a function to be performed by the server. An announcement can be compared to a procedure-call (without the use of output parameters) in, for example, the PASCAL programming language. The client initiates an *invocation* resulting in the conveyance of data (the procedure arguments) from the client to the server. The server performs some actions using the received data and does not return a result.

An interrogation consists of two interactions in different directions, one from client to server, the *invocation*, and one from server to client, the *termination*. An interrogation can be compared to a function-call in traditional imperative programming languages. The client initiates the invocation, resulting in the conveyance of data (the function arguments) from the client to the server. In response to the invocation, the server performs some actions using the received data and initiates a termination, resulting in the conveyance of data from the server to the client, returning the results of the actions.

All interactions at a *stream interface* are continuous flows of data from a *producer* object to a *consumer* object. Flows may be used for continuous sequences of data transmissions between clients and servers.

Engineering language

The engineering language contains concepts for describing the infrastructure required to support selective distribution transparent interactions between objects. The language contains rules for structuring communication channels between objects, using the concepts of stub, binder, protocol objects and interceptors, and rules for structuring systems for the purposes of resource management, using the concepts of node, nucleus, cluster and capsule.

These concepts are sufficient to enable specification of internal interfaces within the infrastructure, enabling the definition of distinct conformance points for different transparencies, and the possibility of standardisation of a generic infrastructure into which standardised transparency modules can be placed.

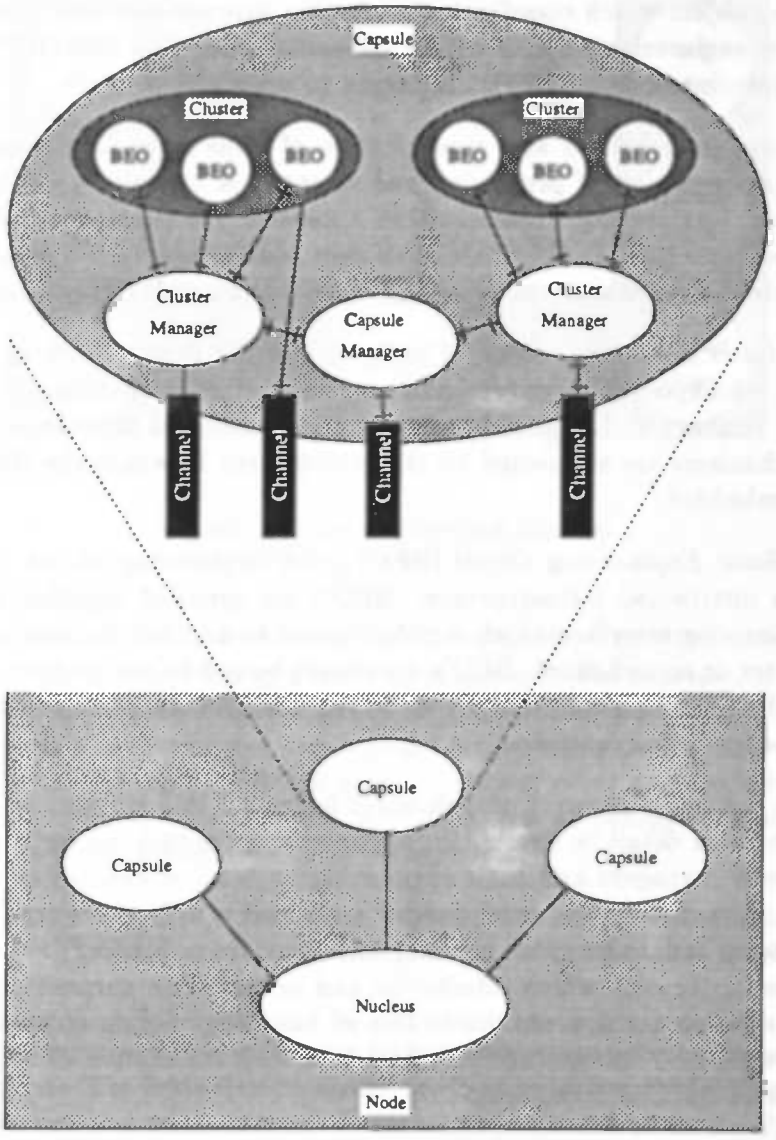


Figure 3.2: An engineering node

Figure 3.2 shows how engineering objects are structured. A *node* is an engineering abstraction of a (physical) computing system. 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.

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

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. Engineering objects within a capsule are protected from engineering objects in other capsules, i.e., they have their own address space. If a capsule fails, only the objects inside the capsule are affected and the objects outside the capsule remain unaffected.

A *cluster* is a configuration of basic engineering objects forming a single unit of deactivation, checkpointing, recovery and migration. The mechanisms of deactivation, checkpointing, recovery and migration are outside the scope of this thesis. It is assumed that these mechanisms are supported by the environment in which the Distributed Storage System is embedded.

A *Basic Engineering Object* (BEO) is an engineering object that requires the support of a distributed infrastructure. BEO's are grouped together in a cluster and have an engineering interface which is either bound to another engineering object within the same cluster or to a channel. BEO's are always bound to the nucleus. In this thesis, BEO's are used to model functionality that is not modelled by other engineering objects defined in the engineering language.

The concept of channel, also shown in figure 3.2, still remains unexplained. It is illustrated with more detail in figure 3.3. Apparently a channel can be bound to cluster managers, capsule managers and basic engineering objects. A channel can cross the boundary of a cluster, a capsule and even a node. A channel is defined as a configuration of *stub, binder, protocol* and *interceptor* objects, and provides a binding between a set of engineering objects, through which interaction can occur. The purpose of a channel is to support distribution transparent interaction of basic engineering objects. In this thesis, channels are used only for operational interaction, i.e., interaction of basic engineering objects at their operational interfaces.

A stub is an object which provides conversion functions for data, exchanged between two or more BEO's. A stub objects provides wrapping and coding functions for the parameters of an operation. This means that the parameters of an operation 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 the results. Wrapping and coding is also referred to as *marshalling* [11].

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

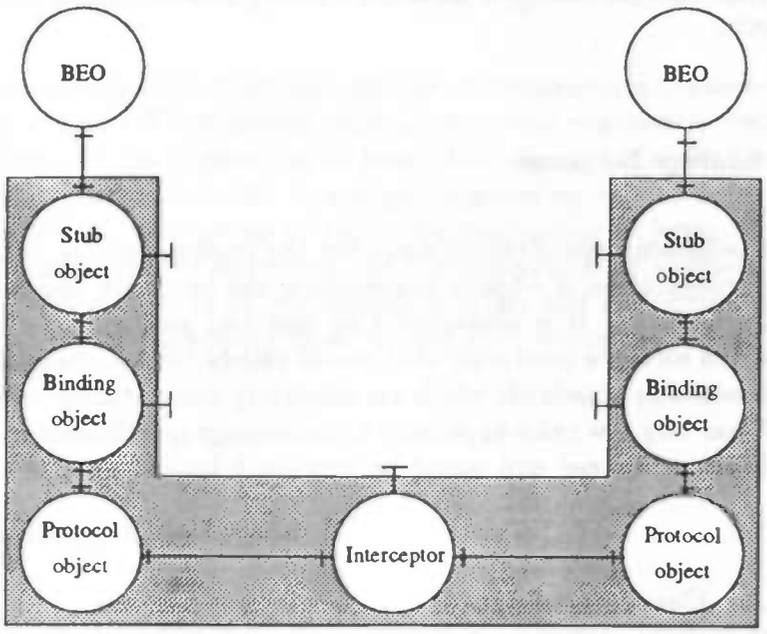


Figure 3.3: An engineering channel

presented by a stub object is transported to the correct target stub object. A binder object also manages the *Quality of Service* of a channel. For example, a binder object can control jitter in a continuous stream by setting a local buffer space. By means of a control interface, a binder object can interact with objects outside the channel. This control interface can be used to obtain location data of other engineering objects or to change the configuration of the channel. A channel can be changed if, for instance, the Quality of Service of the channel needs to be adjusted.

An interceptor is an object at a boundary between domains. Interceptors play a role if interacting protocol objects are in different domains. It can be used to enforce security policies.

A protocol object communicates with other protocol objects to achieve interaction between engineering objects. The RM-ODP identifies protocol objects capable of interworking to be in the same communication domain. Protocol object based on TCP/IP, for instance, belong to the same communication domain, but do not belong to the communication domain of protocol objects based on ATM. The communication between protocol objects takes place at their communication interface.

In this thesis, a derivate of the protocol object is also used: the *group protocol object.* This object is not defined in the RM-ODP, but is defined and implemented in ANSAware. ANSAware is an open distributed environment which is in accordance with the RM-ODP. Besides the normal functionality provided by a general protocol object, the group protocol object supports mechanisms for the coordination of the interaction of grouped engineering

objects. This includes, for instance, a voting mechanism for results from replicated server objects.

Technology language

The technology specification describes the implementation of the ODP system in terms of a configuration of objects representing the hardware and software components of the implementation. It is constrained by cost and availability of technology objects (hardware and software products) that would satisfy the specification. These may conform to implementable standards which are effectively templates for technology objects. The RM-ODP has very few rules applicable to technology specifications. Additional rules would be implementor defined and would be very much implementation dependent.

3.2.3 Consistency rules

A set of specifications of an ODP system written in different viewpoint languages should not make mutually contradictory statements, i.e., they should be mutually consistent. Thus, a complete specification of a system includes statements of correspondences between terms and language constructs relating one viewpoint specification to another viewpoint specification, showing that the consistency requirement is met. The RM-ODP does not declare generic correspondences between every pair of viewpoint languages, it is restricted to the specification of correspondences between a computational specification and the information specification, and between a computational specification and an engineering specification.

Consistency between the computational and information specification

The RM-ODP does not prescribe exact correspondences between information objects and computational objects. In particular, not all states of a (composite) computational object need to correspond to states of the corresponding information object. Multiple subsequent transitional states of a (composite) computational object may be abstracted as one atomic transactional state of the corresponding information object.

Where an information object corresponds to a set of computational objects, static and invariant schemas of the information object correspond to possible states of the computational objects. A change in the state of an information object corresponds either to interactions between computational objects or to an internal action of a computational object. The invariant and dynamic schemas respectively correspond to the contract of the computational objects with their environment and to the behaviour of the computational objects.

Consistency between the computational and engineering specification

The RM-ODP prescribes very strict rules for the correspondences between computational and engineering objects. There should exist a one-to-one relationship between the computational objects and the engineering objects. Each computational object should have an engineering image, whether this is a single engineering object, a group of interacting engineering objects, or a group of replicated engineering objects. The same rule is applied for the computational interfaces and the computational bindings. Each computational binding corresponds to an engineering interface and each computational binding either corresponds to an engineering local binding (i.e., within the same cluster) or to an engineering channel.

3.3 Modelling Concepts

In the RM-ODP, the primitive modelling concept is an object. Objects are entities containing information and offering services. Every ODP system specification should be based on the concept of objects. A system is composed of interacting objects. An object is characterised by its identity which makes it distinct from other objects and by encapsulation, abstraction and behaviour. From the point of view of any object, the ODP system consists of itself and its environment (i.e. all the other objects).

The object model is essential for describing, specifying and designing ODP systems. Abstraction is crucial to deal with heterogeneity, permitting different services to be implemented in different ways, using different mechanisms and technologies, enabling portability and interoperability. Object abstraction also builds a strong separation between objects, enabling them to be replaced or modified without changing their environment, provided they continue to support the services their environment expects. This approach to extendability is essential in large, heterogeneous, distributed environments, which by their nature are continuously evolving. The object model provides modularity and compositionality which are very useful for building flexible systems. The model is fairly general and makes a minimum number of assumptions. For instance:

- objects can be of any granularity, they can be as large as an entire telephone network and as small as an integer,
- objects can exhibit arbitrary behaviours and any arbitrary level of internal parallelism,
- interactions between objects are not constrained, interactions may as well be asynchronous as multi-way synchronous.

3.3.1 Encapsulation and abstraction

Encapsulation is the property that the information in an object can be accessed only through interactions at the interfaces supported by the object. Abstraction implies that the internal details of an object are hidden from other objects. Because objects are encapsulated there are no hidden effects of interactions. An interaction with one object cannot change the state of another object without some secondary interaction taking place. Thus, any change in the state of an object can only occur as a result of an internal action or as a result of an interaction with its environment. An object defines a set of services that can be offered to clients of the objects. The description of a service abstracts from the internal representation for that service. This supports system independency, the same object may be implemented in a number of ways on different systems, but each implementation supports the same service. A service may therefore be supported by many different technologies.

3.3.2 Behaviour versus state

The behaviour of an object is defined as the set of all potential actions an object may take part in. The object model does not constrain the form or nature of object behaviour. State and behaviour are interrelated concepts. State characterises the situation of an object at a given instant. The behaviour of an object describes all the object's potential state changes. The current state of an object is determined by its past behaviour. Conversely, potential actions an object may undertake in the future are determined by its present state. Of course, the actions the object will actually undertake are not entirely determined by its present state, they will also depend on which actions the environment is prepared to participate in.

For behavioural analysis, a system comprised by individual, interdependent and interacting objects can best be modelled as a finite state machine. Formal description languages, such as SDL or LOTOS, actually model system behaviour using interacting and interdependent processes which comprise (extended) finite state machines.

3.3.3 Interfaces

The only means to access an object are interfaces. An interface can be seen as a gate at which a particular subset of the object's behaviour can be observed. An ODP object can have many interfaces. This is a useful property since it can divide the interactions supported by the object into categories.

In order to use interfaces, it is necessary to have some means of uniquely identifying them within the context of the ODP system. Interface identifiers provide such means. Interface identifiers may be passed between objects. Once an interface is identified, particular interactions can be identified within the context of that interface.

3.4 ODP functions

The ODP functions are assumed to be supported, at the engineering level, by the environment in which the specified system is to be implemented. The ODP functions support the construction of ODP systems, and are assumed to be provided by standard engineering objects in the direct environment of basic engineering objects.

The ODP functions can be divided into four categories:

1. management functions,
2. coordination functions,
3. repository functions, and
4. security functions.

The management functions support the managing (i.e., creation, controlling, and termination) of objects, clusters and capsules. These functions can be used, for example, for instantiating a new capsule or a new cluster within a capsule.

The coordination functions support the distribution transparent coordination of the interaction between engineering objects. For example, objects participating in a multi-party binding are controlled by an object providing the *group function*, and replicated objects are controlled by an object supporting the *replication function*.

The repository functions support the storage of various types of data. These include functions for the storage of application specific data, but also for the storage of object related data, such as location and supported functions. The latter can be used to obtain the location of a server object by a client object in order to achieve an (implicit) binding between these objects. The location and the functions supported by server objects are usually recorded by a *Trader*. Client objects consult the Trader if they want some function to be performed by some server object.

The storage of application specific data is usually supported by *data repositories*. For these objects you could think of database managers, supporting the creation, access, and deletion of data records.

The security functions support the setting of constraints on activities of objects. For example, the *access control function* prevents unauthorised interactions with an object.

Chapter 4

Design of Open Distributed Systems

In this chapter, a design methodology for designing systems in conformity to the RM-ODP is introduced. The design methodology is derived from the five ODP viewpoints and is also used and presented in [8, 19].

4.1 Definitions

In Webster's dictionary, a system is a regularly interacting or interdependent group of items forming a unified whole. According to this definition, a system consist of parts. A system is functionally distributed over these parts.

A system can be viewed from two perspectives: the integrated perspective and the distributed perspective. From the integrated perspective the system is viewed as a whole. The system is represented as a black box providing a function $\mathcal{F}$ (see figure 4.1).

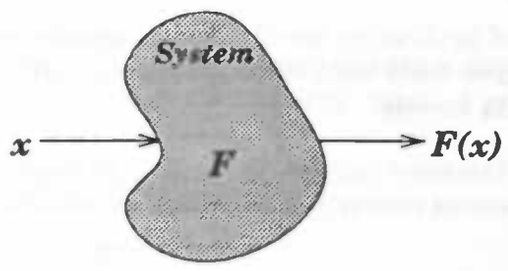


Figure 4.1: A system viewed from the integrated perspective

From the distributed view separate parts of the system are identified. The system is represented as an interacting or interdependent group of items. Each item provides a

function $\mathcal{F}_i$. Interconnected, the items provide the function $\mathcal{F}$, which is the composite function of the separate $\mathcal{F}_i$'s (see figure 4.2).

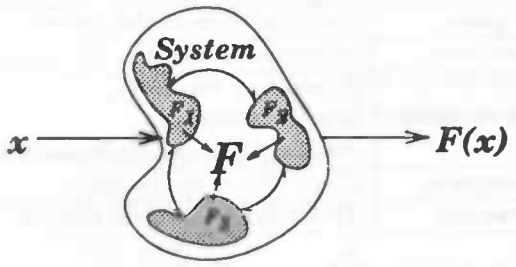


Figure 4.2: A system viewed from the distributed perspective

Usually, a system consists of a communication infrastructure. This implies a geographical distribution of the system parts. In this thesis, system parts are assumed to be geographically distributed. Therefore, the following definition of a distributed system is applied:

A distributed system is a system consisting of at least two system parts connected by some kind of transport network.

4.2 Design Methodology

Distributed systems are complex, they consist of parts that might be further substructured. The design process used in this thesis is also used in [8] and is presented and defined in [19].

4.2.1 Design phases

The ODP viewpoints can be used in the development of a distributed system. In this approach the viewpoints are layered and constitute a top down design process of stepwise refinement. The intermediate steps in this process are called design steps. The result of a design step is a symbolic representation of (parts of) the system. In the top down approach each subsequent design step is a refinement of the previous one. The design process can be structured into the following design phases:

1. The requirements capturing phase.
2. The architectural phase.
3. The implementation phase.
4. The realisation phase.

Table 4.1 summarises the ODP viewpoints and shows to which viewpoints the design phases can be assigned. Figure 4.3 shows how these phases are related.

Viewpoint	Visible	Design phase
Enterprise viewpoint	Requirements of the system	Requirements capturing phase
Information viewpoint	Information flows in the system	Architectural phase
Computational viewpoint	Processing functions, synchronisation, communication and data types	
Engineering viewpoint	Distribution mechanisms	Implementation phase
Technology viewpoint	Technical realisation aspects	Realisation phase

Table 4.1: Assignment of the ODP Viewpoints to subsequent design phases

In the requirements capturing phase, the requirements of the system are determined. The objectives and activities of the enterprise with interest in the system direct these requirements. The requirements are visible from the enterprise view.

In the architectural phase, the system is viewed from the computational viewpoint. From this viewpoint you can only see *what* the system does. The result is a *computational specification* describing the relevant properties that the system should possess. The computational specification is realisation-independent, i.e. it doesn't describe *how* the required properties can be achieved.

In the implementation phase the computational specification is further refined. This results in an *engineering specification*. The engineering specification describes how the required properties of the system can be achieved using physical or logical components. The engineering specification is no longer realisation-independent, using physical or logical components implies taking the constraints imposed by these components into account.

In the last phase, the realisation phase, the implementation is actually mapped onto physical and logical elements that form the real system. The hardware and software components that make up the real system are described in a *technology specification*.

4.2.2 design of a general model for distributed storage systems

In this thesis a general model for distributed storage systems is designed. In this design the following design phases from the design methodology discussed previously are followed:

- the requirements capturing phase (Chapter 6),
- the architectural phase (Chapter 7), and
- the implementation phase (Chapter 8).

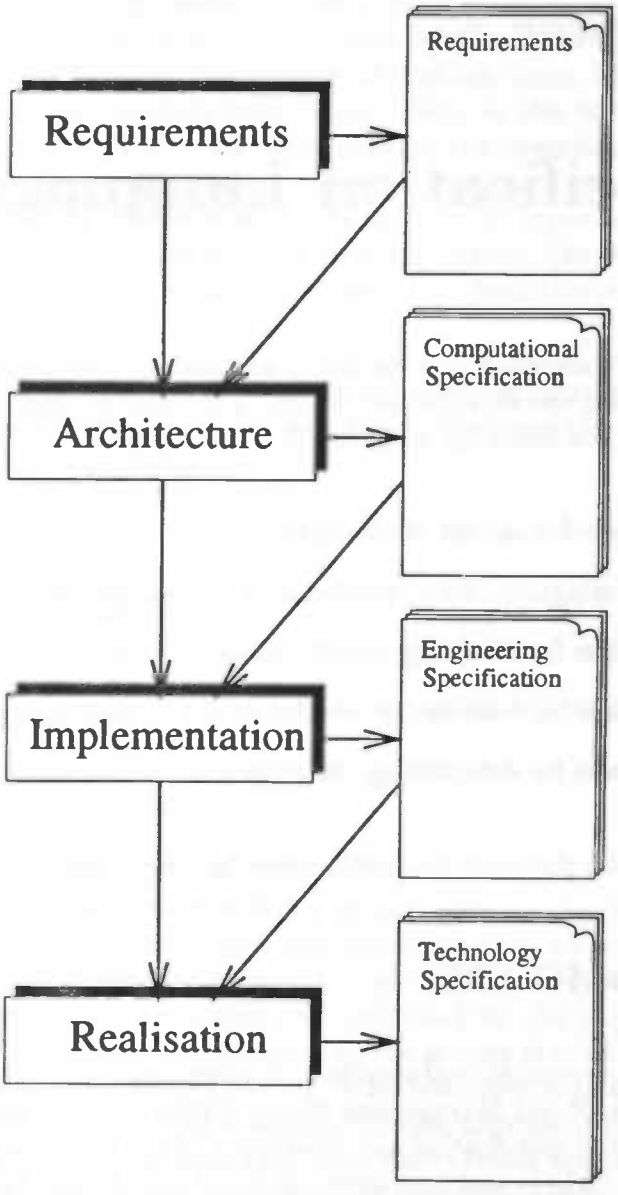


Figure 4.3: The top down design process constituted by the ODP viewpoints

Chapter 5

Specification Languages

It is widely accepted that the key to successfully developing a system is to produce a thorough system specification and design. This task requires a suitable specification language, satisfying the following needs [3, 5]:

- A well-defined set of concepts.
- Unambiguous, clear, precise, and concise specifications.
- A basis for analysing specifications for completeness and correctness.
- A basis for determining whether or not an implementation conforms to specifications.
- A basis for determining the consistency of specifications relative to each other.

This chapter discusses the specification languages used in this thesis: OMG-IDL and SDL.

5.1 OMG-IDL

In ODP, a computational specification comprises a configuration of objects and their interfaces. The Object Management Group (OMG) defined the Interface Definition Language (IDL) for the specification and description of the interfaces at which these objects interact [16]. The objects and interfaces are instantiated from templates. OMG-IDL provides a syntax to describe object templates and interface templates.

Interface templates define the operations that an instance of its template supports. The general structure of an interface template is:

```
<interface_template_name>
[<inheritance_specification>]
```

```
<supporting_definitions>
<operation_signature_specification>
<informal_behaviour_specification>
```

The interface_template_name clause specifies a unique name for the interface template. The supporting_definitions clause contains type definitions that are used in the signatures for the operations. The inheritance_specification clause, which might be empty, specifies which existing interface templates are specialised by this interface template. The operation_signature_specification clause specifies the operations that an instance of the template supports. For every operation, its name, arguments and results are specified. Arguments are preceded by the keyword in to indicate an input value and arguments are preceded by the keyword out to indicate an output value. The informal_behaviour clause is a textual explanation of what an instance of the template is used for and how it behaves.

An object template defines how instances of the template are initialised, which interfaces are supported and which interfaces are required by an instance of the template. An object template generally has the following structure:

```
<object_template_name>
[<inheritance_specification>]
<supporting_definitions>
<initialisation_specification>
<supported_interfaces>
<required_interfaces>
<informal_behaviour_description>
```

The object_template_name clause specifies a unique name for the object template. The supporting_definitions clause contains type definitions that are used in the signatures for the operations. The inheritance_specification clause, which might be empty, specifies which existing object templates are specialised by this object template. The initialisation_specification clause specifies the actions that occur when the object is instantiated. These actions are specified as an operation that is implicitly invoked when the object is instantiated. This operation accepts arguments from and returns results to the creator object. The supported_interfaces clause specifies which interfaces an instance of the template supports. Supported interfaces are interfaces whose functionality is supported on request by the environment. An object can support stream interfaces and operational interfaces. The ODP signal interfaces can be modelled using operational interfaces where the operations have no arguments and return no results. The required_interfaces clause specifies which interfaces an object, instantiated from the template, uses. Required interfaces are interfaces that an object expects from its environment. The informal_behaviour_description clause specifies in an informal way, how an instance of the template behaves.

5.2 SDL

The Specification and Description Language (SDL) is a standard language, developed and standardised by ITU-T, for specifying and describing systems. Its purpose is to provide a language for unambiguous specification and description of the behaviour of telecommunications systems. SDL is descibed in in the CCITT recommendation Z.100 [3, 4]. Since 1988, ITU-T (the former CCITT) recommends the formal specification language SDL-88 as a standard language for specifying (mostly telecommunications) systems. Since 1992, ITU-T recommends SDL-92 (also known as OSDL or Object-Oriented SDL), an extension of SDL-88 with object oriented concepts. SDL-92 uses the same concepts as SDL-88, but with an object oriented approach, i.e., inheritance is applied.

In SDL the terms specification and description are used with the following meaning:

1. a specification of a system is the description of its *required* behaviour, and
2. a description of a system is the description of its *actual* behaviour.

But, since there is no distinction between use of SDL for specification and its use for description, the term specification is used for both required behaviour and actual behaviour.

In SDL, systems are viewed from overview to detail. At each level of abstraction, SDL provides structural and behavioural information about the described system. The type of system described by SDL can be real-time, interactive, distributed or any combination of these types. Applications of SDL include:

- call processing in switching systems (e.g. call handling, telephony signalling, metering)
- maintenance and fault treatements in general telecommunications systems (e.g. alarms, automatic fault clearance, routine tests),
- system control (e.g. overload control, modification and expansion procedures),
- operation and maintenance functions, network management,
- data communication protocols.

SDL can be used for both high level informal specifications, semi-formal specifications and detailed specifications. The user must choose the appropriate parts of SDL for the intended level of communications and the environment in which the language is being used. Depending on the environment in which a specification is used, many aspects may be left to the common understanding between the source and the destination of the specification. Thus SDL may be used for producing system specifications at any level of abstraction. This makes SDL a very suitable language for specifying ODP systems. In ODP, a system is viewed from different viewpoints (see Chapter 4). Each viewpoint abstracts from certain

aspects of the system. SDL may be used to describe the system aspects visible from any of these viewpoints.

In this thesis, SDL is used to describe the designed system from ODP's engineering viewpoint.

An SDL specification defines a systems behaviour in a stimulus/response fashion, assuming that both stimuli and responses are discrete and carry information. In particular, a system specification is seen as the sequence of responses to any given sequence of stimuli. The system specification model is based on the concept of communicating extended finite state machines. Our believe is that finite state machines can be converted to Markov models which can be used for performability evaluation.

SDL gives a choice of two different syntactic forms to use when representing a system:

1. a Graphical Representation (SDL/GR), and
2. a textual Phrase Representation (SDL/PR).

Both forms are semantically equivalent, as they are concrete representations of the same SDL semantics. In particular they are both equivalent to an abstract grammar for the corresponding concepts. Each of the concrete grammers has a definition of its own syntax and its relationship to the abstract grammar (i.e. how to transform into the abstract syntax). The semantics of SDL are defined in the abstract grammar. The concrete grammars inherit these semantics via their relations to the abstract grammar. This approach ensures that SDL/GR and SDL/PR are equivalent.

In this thesis, the Graphical Representation of SDL is used.

5.2.1 Modelling concepts

The basic concept in SDL is the *process type*. The process type defines the internal data, as well as the behaviour. Based on the process type, process instances can be created. Process instances communicate by means of *signals*, which are asynchronous, and *remote procedure calls* which are synchronous.

The behaviour of a process instance is described by means of a *process graph*, which consists of *states* and *transitions* between states. A transition is triggered by the reception of a signal or remote procedure call, and describes the actions which are carried out when the trigger is received. Parts of the process graph may be enclosed in *procedures*, and if such a procedure is marked as being *exported* it is a remote procedure and may be invoked from other process instances.

In SDL-92, process types can inherit from other process types while specialising their behaviour and communication interface, e.g. new states and transitions can be added, and new signals and exported procedures can be introduced.

Process instances can be created and stopped dynamically, and a specific process instance can be identified by its *process identifier*, i.e. signals and remote procedure calls can be addressed to a specific process instance by means of its process identifier. Process instances may be grouped into *blocks*; this advocates a well-structured specification.

5.2.2 Specifying ODP systems using SDL

Jensen, Jørgensen and Nørbæk [7] presented guidelines for the use of SDL for ODP-compliant services and a mapping from SDL specifications of a system to specifications of a system which can be interpreted in an ODP environment (ANSAware, TINA-DPE, etc.). This mapping preserves the SDL semantics while utilising the features for distributed computing offered by an ODP environment. In [2], SDL is used to specify a *Virtual Private Network* system in accordance with ODP. According to [2], SDL is useful for specifying the dynamic behaviour of ODP systems but a disadvantage of SDL is that it does not support the dynamic binding concept of ODP. SDL describes the behaviour of individual objects in the system. Hence, for complex systems, the global behaviour of the system is not always clear. Furthermore, graphical SDL specifications of such systems easily become large and unclear.

Chapter 6

Requirements

This section describes the first phase in the design process of the Distributed Storage System (DSS). Here, the requirements of the DSS are captured. The DSS is part of a distributed system. There are many applications for the DSS. The applications direct the requirements of the DSS. The enterprise viewpoint of ODP will be used for the specification of the user requirements of the DSS.

6.1 Problem Domain

The requirements for the Distributed Storage System (DSS) can be found in the following areas:

- Disk Arrays (RAID)
- Distributed Databases
- Telecommunications applications

In Chapter 2, these areas were explored in order to find the basic architectural alternatives of Distributed Storage Architectures. These basic architectural alternatives are *fragmentation* and employment of *redundancy*.

6.2 User Requirements

Commonly, users of a distributed application do not have direct access to the distributed storage system used by this application. Therefore, users cannot directly determine the requirements for the distributed storage system. They only determine the requirements for the application that uses the distributed storage system. The application's requirements

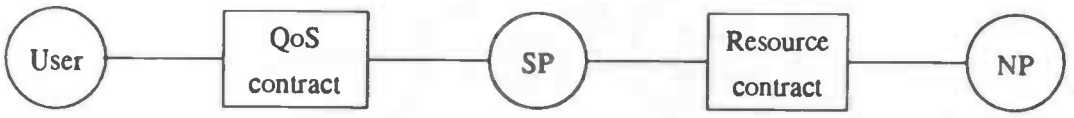


Figure 6.1: Actors and contracts

made upon the performance and reliability of the distributed storage system must be translated into values for the level of fragmentation and the employed redundancy factor, and into policies for the read and write operations on the data stored by the distributed storage system. These policies include locking mechanisms, and voting mechanisms.

As mentioned earlier, the general problem of finding the ideal values for the level of fragmentation and the level of redundancy, is NP-hard. One could, for example, make use of some predefined table which specifies commonly applied values for the levels of fragmentation and redundancy, or use heuristic search algorithms to find optimal values (the AI approach to NP-hard problems).

6.3 Enterprise Specification

The enterprise specification describes the objectives of the Distributed Storage System (DSS) in terms of roles, actors, goals and policies. In this section, some directions for an enterprise specification of a distributed application (for example, a Video On Demand or Banking service) are given.

Actors and roles:

- Service Provider (SP): an organisation providing services as the Video on Demand service or the Bank Transaction Service.
- User: a person or group of persons using the service offered by a service provider.
- Network Provider (NP): an organisation offering the required transport facilities.

For each role the requirements and policies must be stated. These rules form the contract between the involved parties. The roles distinguished in a DSS, and the contracts between the involved parties are illustrated by Figure 6.1.

The User and the SP have an agreement upon the Quality of Service to be provided by the SP. The SP must be able to store data at multiple locations. It uses several resources (nodes, communication links, disks) to do this. These resources are made available by the NP. The contract between the SP and the NP consists of the resources to be used and the accounting and security policies for these resources.

Chapter 7

Architecture

A system from the computational point of view consists of a configuration of computational objects. A computational specification defines the functional decomposition into these objects, which interact at interfaces[11]. The objective is to abstract from aspects of distribution, i.e., the specification is based on *distribution transparency.*

7.1 Objectives

The computational specification of the distributed storage system does not cover the implementation strategies discussed in the previous chapter. It is presumed that such distribution transparencies as fragmentation and replication are supported by an underlying engineering mechanism (which will be discussed in Chapter 8).

7.2 Computational model

Figure 7.1 shows the functional decomposition of the Distributed Storage System into three computational objects and their interfaces. These objects interact according to the Client/Server-model [1]. The computational objects are *Application*, *Storage Manager* and *Storage Unit.*

The Application and Storage Manager interact at interface ①. The Storage Manager is a server which supports interface ①. At this interface functions to create Storage Units are provided to clients. The Application is a client and requires ① to request storage capacity from the Storage Manager. The Storage Manager creates a Storage Unit and provides the Application with a reference to the created Storage Unit. The Storage Units created by the Storage Manager are servers which support interface ②. This interface enables clients to access the data stored by a Storage Unit. The Application requires ② and interacts at this interface with the Storage Unit to access data stored by the Storage

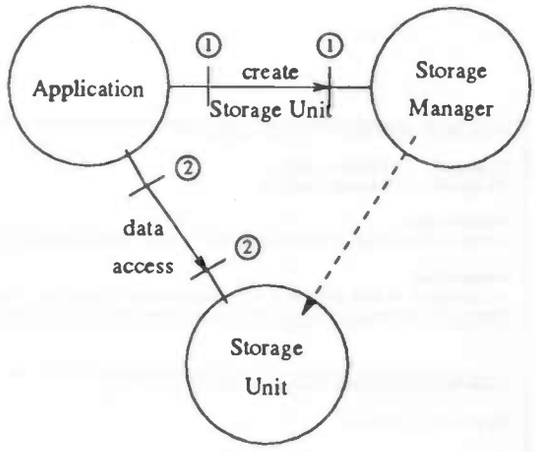


Figure 7.1: Computational Model of the Distributed Storage System

Unit.

The request of the Application to create a new Storage Unit comprises the following information requirements:

- QoS level
- storage capacity
- type of consistency

The Storage Manager needs the QoS requirements from the Application to create a Storage Unit with a performance and reliability matching these requirements. The required storage capacity is needed to create a Storage Unit with sufficient storage capacity. The type of consistency, i.e., weak or strong consistency, is needed to apply the right type of algorithms during transactions related to the Storage Unit.

7.3 Objects and Interfaces specified in IDL

We have used OMG's Interface Definition Language [16] to specify the computational objects and interfaces comprising the computational model. First we define the interface templates which the computational objects require to create their interfaces. Next, we define the object templates from which the computational objects can be instantiated. In the templates, the data types are left open, since they are trivial.

In Table 7.1, the computational interface templates are specified. From these templates the computational interfaces can be instantiated. Table 7.2 specifies the object templates from which the computational objects can be instantiated.

Interface template StorageManagerIF

Typedef ... TPerformability
Typedef ... TStorageUnitRef

Operations
create_container(in TPerformability P, out TStorageUnitRef SU)

Behaviour
An instance of this interface template enables clients to create customised storage units where the desired performability properties can be specified.

Interface template StorageUnitIF

Typedef ... TData

Operations
read(out TData D)
write(in TData D)
empty()
delete()

Behaviour
An instance of this template enables clients to store, modify and clear data. Clients can dispose of the Storage Unit by making a delete request at this interface.

Table 7.1: Computational interface templates

Object template StorageManager

Typedef ... TInterfaceRef

Initialization
Init(out TInterfaceRef StorageManagerRef)

Supported interfaces
StorageManagerIF

Behaviour
An instance of this template is a Storage Manager object which can create storage units. Via the StorageManagerIF clients can request the StorageManager to create a storage unit with specified performability properties.

Object template StorageUnit

Typedef ... TInterfaceRef

Initialization
Init(out TInterfaceRef StorageUnitIF)

Supported interfaces
StorageUnitIF

Behaviour
An instance of this template is a StorageUnit object which is initially empty. It supports the StorageUnit interface through which clients can access the data stored by the StorageUnit.

Table 7.2: Computational object templates

Chapter 8

Implementation

A system from the engineering point of view consists of a configuration of engineering objects. An engineering specification defines the mechanisms and functions required to support the distributed interaction between objects[11].

In ODP, the engineering specification describes how the functionality of the computational objects is distributed and what infrastructure is available to support this distribution.

The engineering specification of the Distributed Storage System describes how the functionality of the *Storage Unit* computational object is distributed using the various fragmentation and redundancy strategies, as discussed in Section 2. The engineering specification is not concerned with the physical components, hence, it abstracts from specific components such as operating systems, data base management systems, and peripheral devices.

8.1 Objectives

The objective is to define a generic engineering model for distributed storage architectures, i.e., the engineering model should apply for various architectures obtained using different fragmentation and redundancy levels. For the Distributed Storage System, we need a general, hardware independent storage function and a way to coordinate a co-operative configuration of multiple objects implementing this storage function.

8.2 Engineering model

In ODP, a number of functions are defined that are either fundamental or widely applicable to the construction of ODP systems[11]. These functions are grouped as follows:

- management functions

- coordination functions
- repository functions
- security functions

In our engineering model we use the *Storage Function* from the repository functions group and the *Group Function* from the coordination functions group.

The Storage Function stores data. The used concepts are *Data Repository* and *Container Interface*. A Data Repository is an object providing the storage function and the Container Interface is an interface of a Data Repository allowing access to data. The Storage Function rules state that a Data Repository stores sets of data. Each set of data is associated with a Container Interface created when data are stored. A Container Interface provides functions to modify, retrieve and delete the associated data.

The Group Function provides the necessary mechanisms to coordinate the interactions of objects in multi-party bindings. The concept of the Group Function, which is a subset of the objects participating in a binding managed by the group function. The rules for the Group Function state that for each Interaction Group the group function manages the *interaction*, *collation*, *ordering* and *membership*.

Part of the engineering model of the Distributed Storage System is shown in Figure 8.1. We compose the engineering model using the concepts of Data Repository, Container Interface and Interaction Group. The objects representing these concepts also interact according to the Client/Server-model [1]. The Interaction Group is modelled by the channel connecting the Data Repositories and the Storage Unit Manager.

Figure 8.1 shows that the functionality of the Storage Unit *computational* object is distributed over multiple Data Repositories and a Storage Unit Manager. The Storage Manager *engineering* object interacts with the Data Repositories at interface ③. The Storage Manager requires this interface to request a Data Repository to create a Container Interface. The Data Repositories interact with each other at Container Interfaces ④ in a multi-party binding. This binding consists of a channel instantiated by the Storage Manager and is controlled by a *Storage Unit Manager* object created by the Storage Manager. The Application interacts with the Storage Unit Manager at interface ② which resembles a Container Interface. The Application uses ② to access the data stored by the group. Hence, the internal fragmentation and replication strategies are masked from the application. The Data Repositories are explicitly drawn in separate nodes. This is essential, because it enables the Storage Manager to create storage at multiple, physically different locations.

The purpose of a channel is to support distribution transparent interaction between engineering objects and consists of a configuration of stubs, binders, protocol objects and interceptors [11]. The stubs perform the necessary translations between data formats on the distinct nodes and the binders ensure that data presented by the stubs is transported to the stubs in the correct nodes. A protocol object communicates with the other protocol objects to achieve interaction between the connected engineering objects.

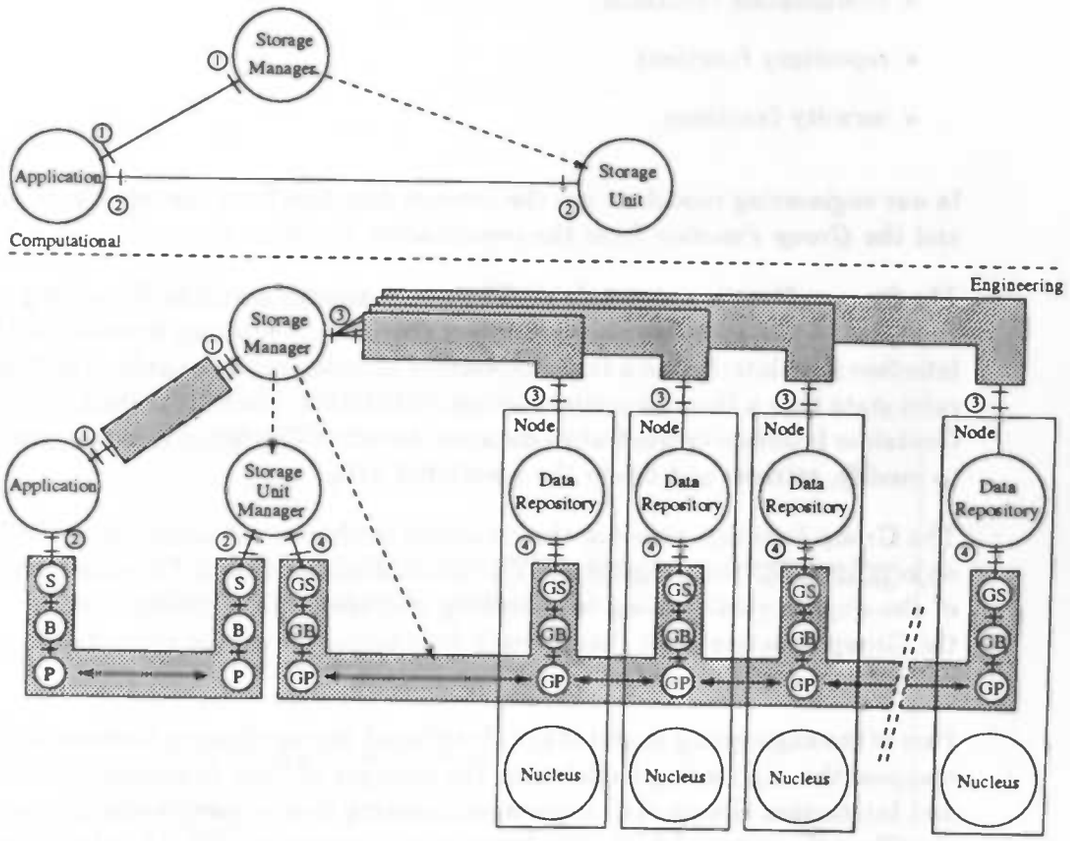


Figure 8.1: Mapping of the Computational Model onto the Engineering Model of the Distributed Storage System

The Group Protocol (GP) objects implement ODP's group function. The desired fragmentation and redundancy strategies are specified as an interaction schema to be used by the Group Protocol objects.

8.3 Consistency with the Computational Model

Figure 8.1 shows how the Computational Model of the Distributed Storage System can be mapped onto the Engineering Model. Each computational object has an corresponding set of one or more engineering objects and each computational interface corresponds to a channel.

8.4 Objects and Interfaces specified in IDL

Interface template DataRepositoryIF

Typedef ... TDataType
Typedef ... TContainerRef

Operations
create(in TDataType DT, out TContainerRef CR)

Behaviour
An instance of this interface template enables clients to create a single container of the specified data type.

Interface template ContainerIF

Typedef ... TData

Operations
read(out TData D)
write(in TData D)
empty()
delete()

Behaviour
An instance of this template enables clients to store, modify and clear data. A client can dispose of the stored data by making a delete request at this interface.

Table 8.1: Engineering interface templates

We have used OMG's Interface Definition Language [16] to specify the engineering objects and interfaces comprising the engineering model. First we define the interface templates which the engineering objects require to create their interfaces. Next, we define the object templates from which the engineering objects can be instantiated.

The interface templates from which the engineering interfaces can be instantiated are specified in Table 8.1. These are new templates to be added to the computational templates defined in Section 3. The engineering Container Interface resembles the computational Storage Unit Interface. Hence, the ContainerIF template replaces the StorageUnitIF template.

Table 8.2 specifies the object templates from which the engineering objects can be instantiated. The templates from which the engineering objects can be instantiated are either new templates to be added to the computational templates, or extended templates to replace computational templates.

The StorageManager object template is an extended computational template. At the engineering level it also has to support the DataRepositoryIF to be able to create new ContainerIF's.

8.5 SDL specification

As mentioned in Chapter 5, IDL includes an informal description of the behaviour of the objects and interfaces. In this thesis, SDL is used for a formal description of the behaviour of these objects and interfaces.

Object template StorageManager (extended)

Typedef ... TInterfaceRef

Initialization
Init(out TInterfaceRef StorageManagerRef)

Supported interfaces
StorageManagerIF

Required interfaces
DataRepositoryIF

Behaviour
An instance of this template is a Storage Manager object which can create Storage Units. Via its StorageManagerIF clients can request the StorageManager to create a Storage Unit with specified performability properties.

Object template DataRepository

Typedef ... TInterfaceRef

Initialization
Init(out TInterfaceRef DataRepositoryIF)

Supported interfaces
DataRepositoryIF
ContainerIF

Behaviour
An instance of this template is a Data Repository object. Data Repositories are factories for container interfaces.

Object template StorageUnitManager

Typedef ... TInterfaceRef

Initialization
Init(out TInterfaceRef ContainerIF)

Supported interfaces
ContainerIF

Required interfaces
ContainerIF

Behaviour
An instance of this template manages a group of container interfaces, and enables clients to access the data stored by the group.

Table 8.2: Engineering object templates

The SDL specification of the Distributed Storage System is discussed in Chapter 9. In SDL, the Distributed Storage System is specified as a set of interacting processes, providing services to the environment and to each other. These processes correspond to the engineering objects: Storage Manager, Storage Unit Manager, and Data Repository. An overview of these processes and services can be found in Figure 9.15.

For objects two types of behaviour can be distinguished: internal behaviour and external behaviour. The internal behaviour of an object is invisible to its environment (other objects). In SDL, the internal behaviour is specified in process diagrams. A process diagram processes input signals which can cause changes to the internal state of the process, and which can cause the process to initiate interactions with other objects.

The external behaviour of an object is defined as the set of all potential interactions with other objects in its environment. In SDL, these potential interactions are specified as sets of signals which can occur at channels between processes. The union of all signal sets of channels from or to a process comprises the external behaviour of this process.

For example, the external behaviour of a Data Repository is defined by the set $\{Create\_ContainerIF, ContainerIF\_Created, Read, Write, Empty, Delete, Data, Submit\}$ (see Figure 9.2). This set contains all potential interactions a Data Repository may take part in.

Of course, the set of potential interactions does not constitute the external behaviour of an object (process) by itself. You also want to know which interactions occur in what situations. The channels at which (subsets of) the interactions occur, indicate the direction for these interactions. Interactions at an input channel of an object indicate messages to the object. As a response to these messages, the object performs certain internal actions and may send messages to other objects via output channels. These internal actions are part of the object's internal behaviour and are masked from the other objects, but cannot be ignored in behavioural analysis of systems.

A Data Repository, for example, creates a new ContainerIF, as a response to the message "Create_ContainerIF", and sends the message "ContainerIF_Created" together with a reference to the newly created ContainerIF to the sender of the input message.

In SDL, both internal and external behaviour of objects can be specified in detail. The external behaviour of objects is controlled by their internal behaviour which process input signals and initiate new interactions. For the analysis of the performance and reliability of existing systems, detailed information about both types of behaviour is required. For systems, fully specified in SDL, we can use the behaviour specification for performance and reliability analysis of implementations of the system.

Chapter 9

SDL Specification

In this chapter, the engineering model, as specified in the previous chapter, is specified using ITU-T's Specification and Description Language (SDL). In the SDL specification, the formal behaviour of the engineering objects: Storage Manager, Storage Unit Manager, and Data Repository is specified. The SDL symbols used in this specification are listed in Appendix A.

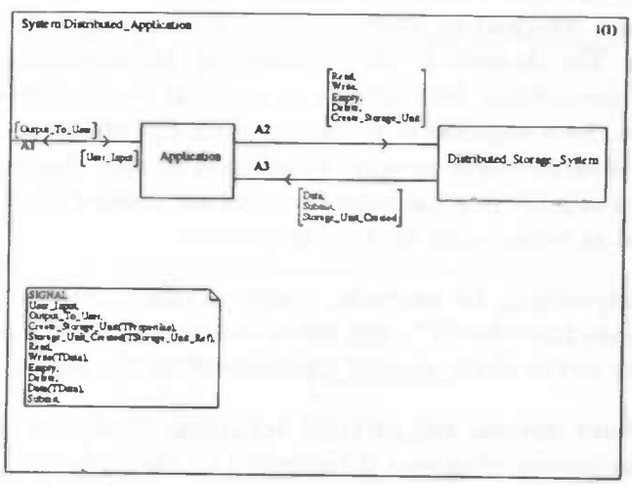


Figure 9.1: Global specification of a distributed application using the Distributed Storage System.

In Figure 9.1, a global view of a distributed application using the Distributed Storage System is given. The application can request the Distributed Storage System to create a Storage Unit which it can use to store its data. With this request, the application can make certain requirements upon the Quality of Service (QoS) of the Storage Unit. The application directly communicates with the Storage Unit in order to store and retrieve data (using the Read, Write, Empty, and Delete messages).

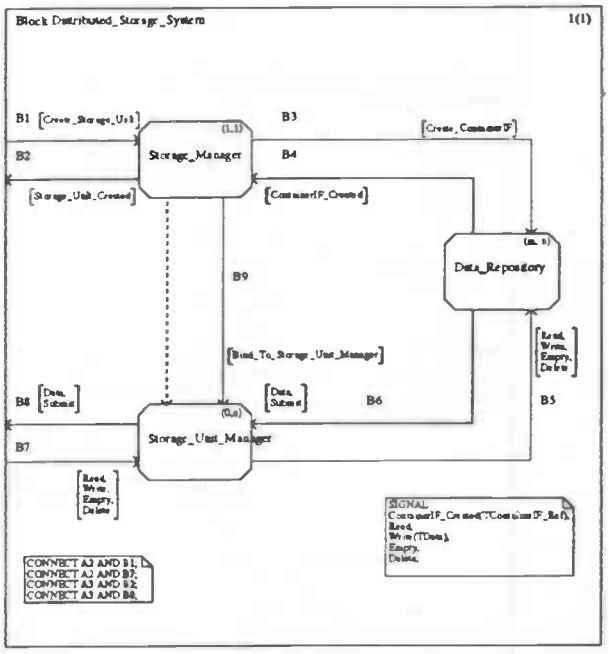


Figure 9.2: The internal processes of the Distributed Storage System.

Figure 9.2 shows the internal processes of the Distributed Storage System. There exists exactly one Storage Manager, multiple Data Repositories (at least one), and multiple Storage Unit Managers (initially zero) which are created by the Storage Manager.

Figure 9.3 shows the services offered by the Storage Manager. The Storage_Unit_Factory service responds to the application's request to create a new Storage Unit, i.e., the Create_Storage_Unit message. It uses the Container_Factory service and Container_Binder service to respectively create a number of containers and to bind these containers to a newly created Storage_Unit_Manager process.

The process graph of the Storage_Unit_Factory service is specified in Figure 9.4. It receives the message Create_Storage_Unit with the required properties (QoS) for the Storage Unit. As a response, it determines suitable values for the level of fragmentation and the redundancy factor to be used for the Storage Unit. This procedure is not furtherly specified in this thesis. As mentioned in Chapter 2, finding optimal values for fragmentation and redundancy is an NP-hard problem which can be approached with AI methods or predifined tables.

After the values for fragmentation and redundancy are determined, a request to create a suitable number of containers (Create_Containers) is sent to the Container_Factory process, and waits until it receives the Containers_Created message. When this message is received, the Storage_Unit_Factory requests the Container_Binder service to bind the containers to a new Storage_Unit_Manager process, and waits until it receives the Binding_Created message. Finaly, when this message, containing the reference to the Storage

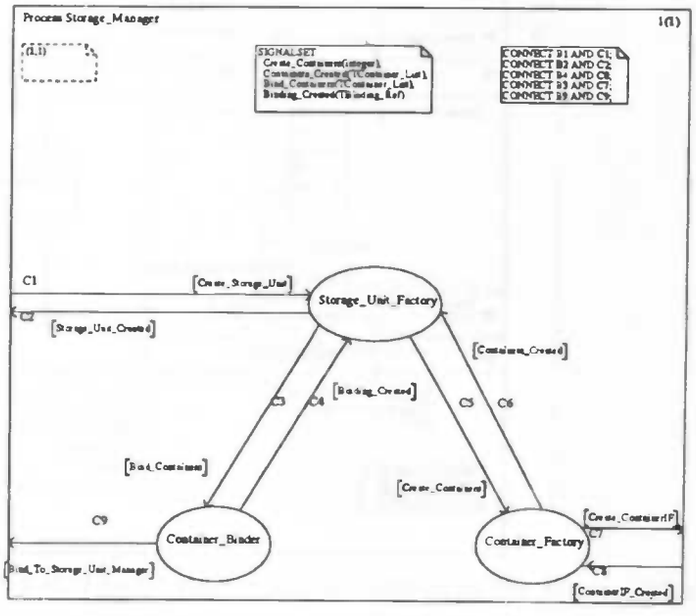


Figure 9.3: The services of the Storage Manager.

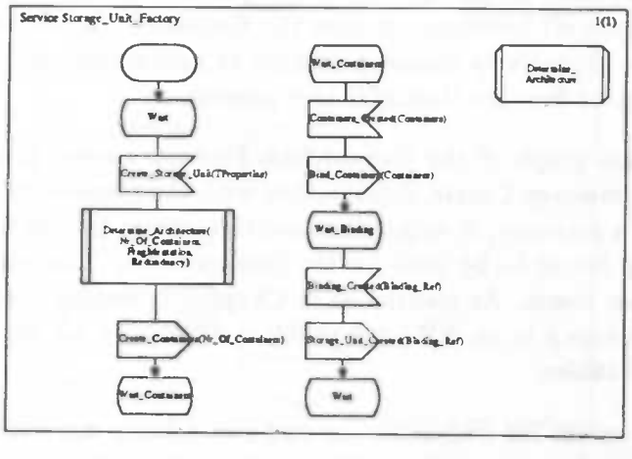


Figure 9.4: The Storage_Unit_Factory service of the Storage Manager.

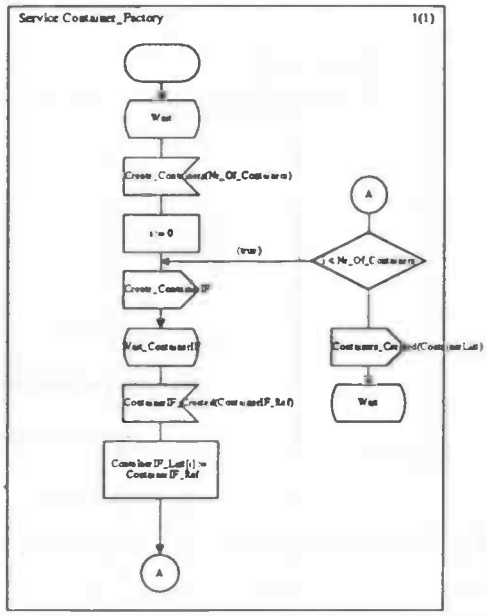


Figure 9.5: The Container_Factory service of the Storage Manager.

Unit (Binding_Ref), is also received, the application is informed that the Storage Unit is created and retrieves the reference to the created Storage Unit.

The process graph of the Container_Factory service is specified in Figure 9.5. It responds to the Create_Containers message sent by the Storage_Unit_Factory service. It creates multiple containers by requesting multiple Data_Repositories to create new container interfaces using the Create_ContainerIF message. When the requested number of container interfaces is created the message Containers_Created with the references to the created container interfaces is sent back to the Storage_Unit_Factory.

The process graph of the Container_Binder service is specified in Figure 9.6. In responds to the Bind_Containers message sent by the Storage_Unit_Factory service it instantiates a new Storage_Unit_Manager process and binds the containers created by the Container_Factory service to the newly created Storage_Unit_Manager by sending the message Bind_To_Storage_Unit_Manager to the newly created Storage_Unit_Manager. Finaly, the message Binding_Created including a reference to this binding (OFFSPRING), is sent back to the Storage_Unit_Factory.

In Figure 9.7, the services offered by a Data_Repository process are specified. The ContainerIF_Factory service can create new container interfaces, and the Container_Interface service provides a way to access these container interfaces.

The process graph of the ContainerIF_Factory is specified in Figure 9.8. It responds to the Create_ContainerIF message by creating the container interface and returning the reference to this container.

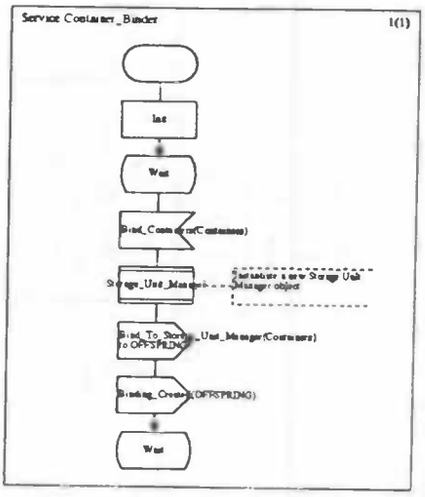


Figure 9.6: The Container_Binder service of the Storage Manager.

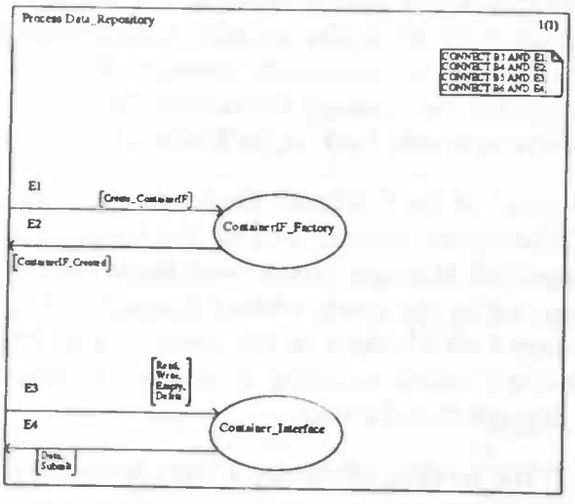


Figure 9.7: The services of a Data Repository.

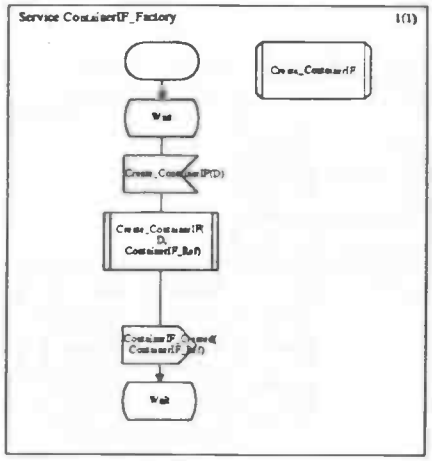


Figure 9.8: The Container_Interface_Factory service of a Data Repository.

The process graph of the Container_Interface service is specified in Figure 9.9. This graph is straight forward, it responds to the messages Read, Write, Empty and Delete by performing the corresponding operation and returning the requested data in the case of a Read operation and returning a Submit message in the case of a Write, Empty, or Delete operation.

In Figure 9.10, the services offered by a Storage_Unit_Manager process are specified. A Storage_Unit_Manager has four services: the Read service, the Write service, the Empty service, and the Delete service. Each of these services performs the corresponding operation on the group of container interfaces the Storage Unit Manager is bound to. The process graphs of these four services are specified in the Figures 9.11 through 9.14. How, for example, data is written to the container interfaces, depends on the policy used for this operation. Such a policy specifies how the orriginal data should be fragmented, how redundant information should be computed, and how the resulting data should be distributed across the container interfaces.

Figure 9.15 gives an overview of the distributed application, showing how the functionality of the Distributed Storage System is distributed over multiple interconnected processes, and how the functionality of these processes is distributed over multiple interconnected services.

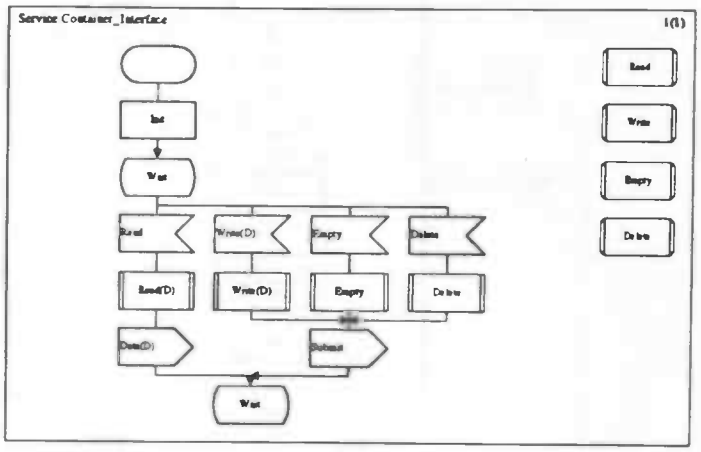


Figure 9.9: The Container_Interface service of a Data Repository.

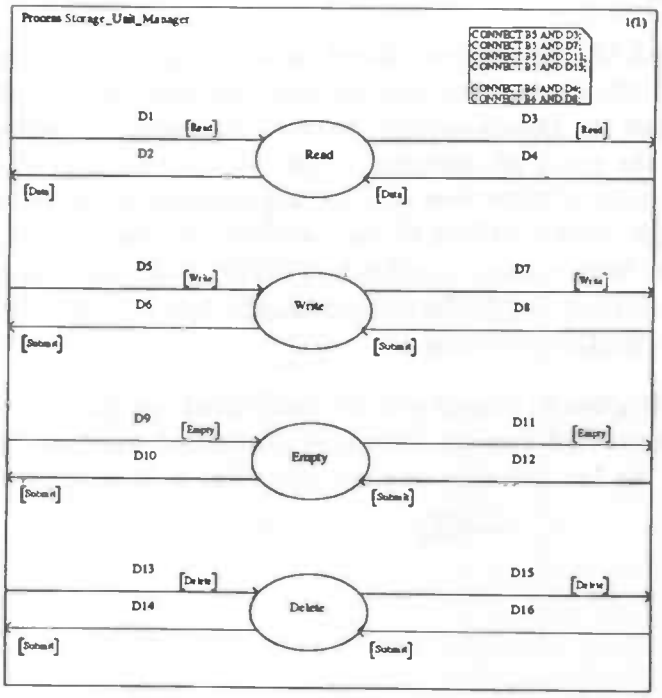


Figure 9.10: The services of a Storage Unit Manager.

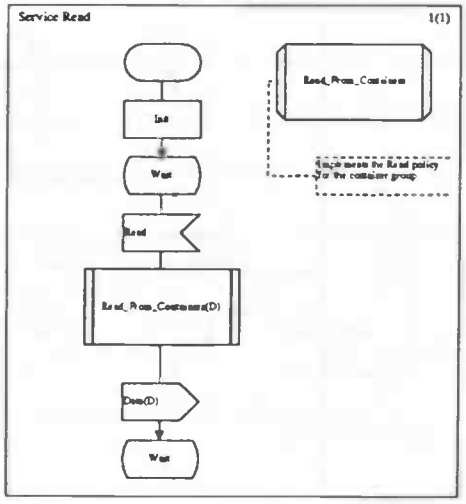


Figure 9.11: The Read service of the Storage Unit Manager.

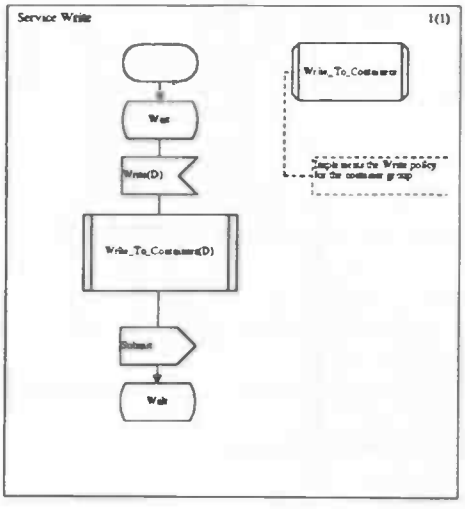


Figure 9.12: The write service of the container manager.

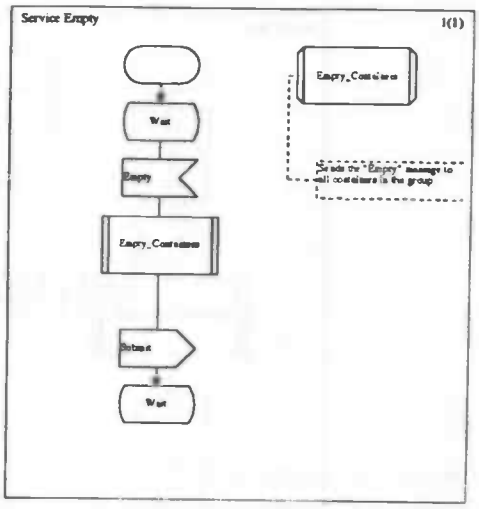


Figure 9.13: The Empty service of the Storage Unit Manager.

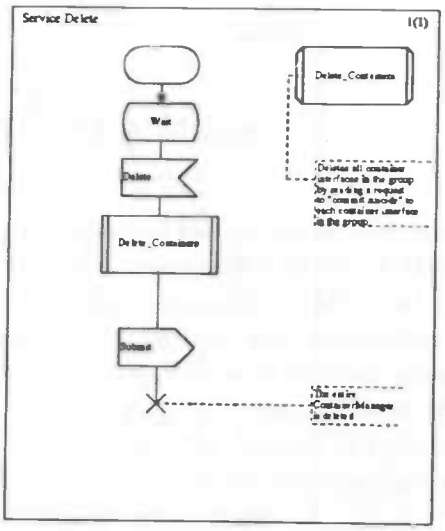


Figure 9.14: The Delete service of the Storage Unit Manager.

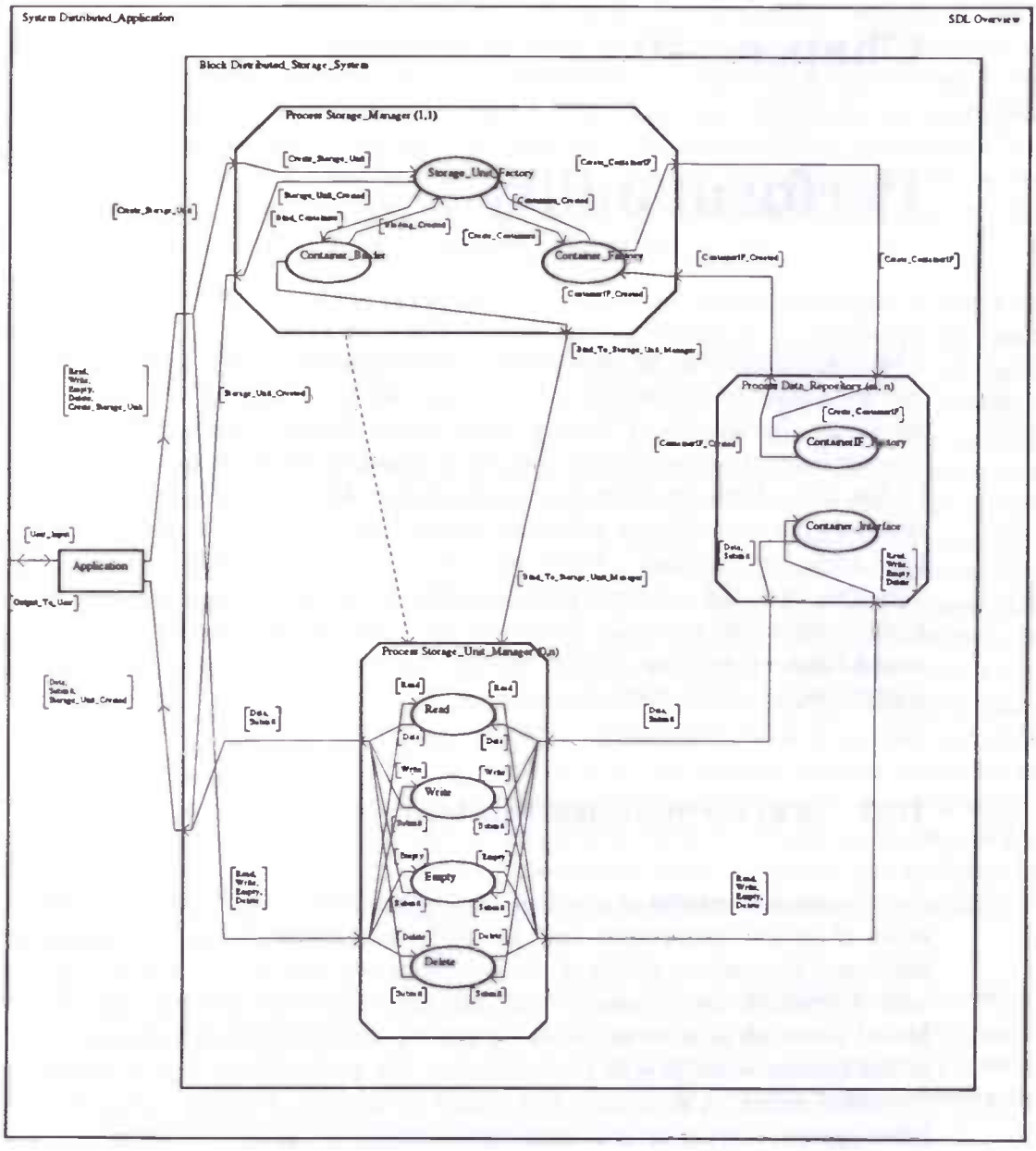


Figure 9.15: Overview of the Distributed Application.

Chapter 10

Performability

This chapter provides an answer to the third question of the problem definition: "how can performability be modelled?". This answer contributes to the solution of the problem of validating the Quality of Service (QoS) of the Storage Units created by the Distributed Storage System as specified in Chapter 7, 8 and 9. The QoS is the performance perceived by the user. Performability is a good measure for QoS, because it obtains performance measures also considering reliability events [20]. It is a more meaningfull representation than strict performance. To be able to verify if Storage Units created by the Storage Manager have the required performability properties, a general method for modelling performability, is required. Furthermore, a general method to obtain a performability model from an existing implementation of a system is needed. In this chapter, only a suggestion for such a method is given.

10.1 Performability Modelling

A performability model of a (distributed) system shows the relation between certain properties of system components and the architecture in which they are embedded on the one hand, and the performability of the system on the other hand. It is used to model systems with degradable performance, fault tolerance and availability [6, 20]. A Performability Model shows all possible states of the system and for each state it shows the performance of the system when it is in that state and the probabilities that it will switch to one of the other states. A state switch is caused by failures of system components or repairs of failed system components. A state switch causes the system's performance to decrease in case of a failure or to increase in case of a repair.

For performability modelling the question "How much work will be done or lost in a given interval including the effects of failure, repairments and contention?" is asked [6]. More precise is the question "What is the expected amount of work that is done or lost by the system in a given interval including the effects of failure, repairs and contention?". Answering these questions results in the systems performability. The questions can be

answered if, for each state of the system, the expected probability of residing in that state during the interval, is known.

A performability model should be evaluated in order to draw conclusions from it. Three basic types of evaluation techniques can be distinguished [6, 20, 18, 13]: analytical evaluation techniques, numerical evaluation techniques, and evaluation techniques based on simulation. Of course, combinations of these techniques also exist. The results of analytical techniques are exact, but many restrictions are imposed on the models to be evaluated. The models which can be evaluated by numerical techniques are much less restricted, but the results are less accurate than the results of analytical techniques. The models evaluated by simulation techniques virtually have no restrictions, but the results are only statistical estimates. The models used for these evaluation techniques can be based on, for example, Petri Nets, or Queueing Networks.

Another distinction that can be made between various evaluation techniques is based on whether the technique is based on behavioural decomposition or an integrated approach [6]. Behavioural decomposition is based on difference in time-scale between types of events [6]. Performance events, e.g., arrival of jobs, take place far more often then dependability events, e.g., failures or repairs of components. Performance events can change the systems operational state, e.g., arrivals of jobs may cause the system to change from idle state to a processing state. Dependability events can change the systems structure state, the overall performance of the system degrades in the case of a failure, or increases in the case of a repair. Looking at the length of the time interval between two successive dependability events, and the large amount of performance events in that interval, it seems reasonable to assume that the performance of the system between these successive structure state changes is in steady state most of the time. Thus the percentage of time the system is not in steady state is assumed to be negligible. Of course, this assumption is approximate. First of all, the transient phase between successive operational states is omitted, i.e., it is assumed that the system switches instantaneously from one state to another. Secondly, it is assumed that the system always reaches a steady state between two successive structure state changes. This does not have to be true. It is possible for a system to operate for some time in an instable way. Both assumptions have their impact on the usability of behavioural decomposition. The integrated approach does not suffer these drawbacks, but the model becomes more complex.

This chapter discusses an evaluation technique which is based on behavioural decomposition. Behavioural decomposition leads to a model consisting of two parts: a stochastic process $\{X(t), t \geq 0\}$ describing the structure state changes of the system, and a reward function r defined on the state space M of $\{X(t), t \geq 0\}$. In the case that the stochastic process is Markovian, a Markov Reward Model is obtained.

As mentioned above, for performance evaluation, queueing network models are widely used and results from queueing theory and simulations of queueing networks are generally applied. However for models that do not satisfy product form requirements nor have some other closed form solution, the most general approach is still to built and numerically solve a Markov performance model. However, simulation is also used [20]. The next section presents a short overview of stochastic processes and the special case of a stochastic

process: the Markov process.

10.2 Markov models

A stochastic process can be described as a collection of random variables $\{X(t) \mid t \in T\}$ defined on some probability space, and indexed by the parameter t which can take values in some set T. The values that $X(t)$ assumes are called states. The set of all possible states is called the state space and is often denoted by I. This state space can be continuous or discrete. Therefore, stochastic processes can be divided into *continuous* stochastic processes and *discrete* stochastic processes. A continuous process has a continuous state space, and a discrete stochastic process has a discrete state space and is also called a *chain*. Also for the index set T, the set can be of the positive integers N^+, in which case we have a discrete time stochastic process, or the indexed set T can be continuous Z^+, in which case we have a continuous time stochastic process.

A special case of these stochastic processes are the Markov processes. A stochastic process $\{X(t) \mid t \in T\}$ is called a Markov process if for any $t_0 < \ldots < t_n < s < t$ the distribution of $X(t)$, given the values $X(t_0) < \ldots < X(t_n) < X(s)$ only depends on $X(s)$. That is, $P[X(t) \leq x \mid X(t_0) = x_0 < \ldots < X(t_n) = x_n, X(s) = x_s] = P[X(t) \leq x \mid X(s) = x_s]$. This is generally denoted as the *Markov property*. This means that the next state of the process only depends on the current state. Hence the Markov property is also called the *memoryless property*. A Markov process is called *homogeneous* if the process is invariant to time shifts, i.e., it satisfies

$$P[X(t) \leq x \mid X(s) = x_s] = P[X(t-s) \leq x \mid X(0) = x_s].$$

Because of the memoryless property the state residence times are in the discrete time case geometrically distributed and in the continuous time case exponentially distributed.

In this thesis, only Markov chains are considered. The interest is in finding the state probabilities

$$\pi_j(t) = P[X(t) = j],$$

where $\pi_j(t)$ is the probability that the system is in state j at time t. In the case of homogeneous Markov processes, these would depend on their initial distribution $P[X(0) = i]$, which is assumed to be given.

Besides the state probabilities we have the transition probabilities

$$p_{i,j}(u,t) \doteq P[X(t) = j \mid X(u) = i].$$

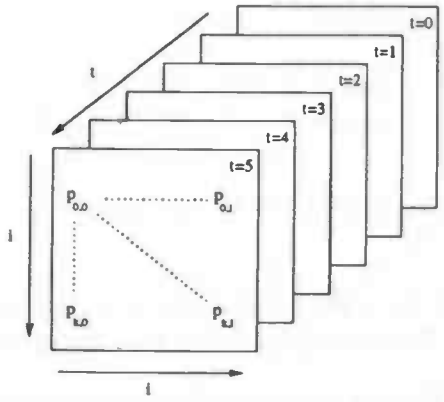


Figure 10.1: The one-step transition probability matrices in time for the discrete time Markov chain

This is the probability that the system is in state j at time t, given that it was in state i at time u. We can then write

$$\pi_j(t) = \sum_{i \in I} \pi_i(u) P_{i,j}(u,t)$$

which can be rewritten as

$$\Pi(t) = \Pi(u) P(u,t).$$

In this notation $\Pi(t)$ denotes the row vector $(\pi_0(t), \pi_1(t), \ldots)$ and $P(u,t)$ denotes the matrix

$$\begin{bmatrix} p_{0,0}(u,t) & p_{0,1}(u,t) & \ldots & p_{0,l}(u,t) \\ p_{1,0}(u,t) & p_{1,1}(u,t) & \ldots & p_{1,l}(u,t) \\ \vdots & \vdots & & \vdots \\ p_{k,0}(u,t) & p_{k,1}(u,t) & \ldots & p_{k,l}(u,t) \end{bmatrix}.$$

For a discrete time Markov chain (DTMC), it is fruitful to think of the process as making transitions at times $t = 0, 1, 2, \ldots$. At $t = 0$ a discrete time Markov chain starts in an initial state i, $X(0) = i$, and makes a state transition at the next step, i.e., at $t = 1$, so that $X(1) = j$, etcetera.

In the case of a DTMC, let $u = n$ and $t = n + 1$. This results in the following equation:

$$\Pi(n+1) = \Pi(n) P(n),$$

where $P(n)$ denotes the matrix

$$\begin{bmatrix} p_{0,0}(n,n+1) & p_{0,1}(n,n+1) & \dots & p_{0,l}(n,n+1) \\ p_{1,0}(n,n+1) & p_{1,1}(n,n+1) & \dots & p_{1,l}(n,n+1) \\ \vdots & \vdots & & \vdots \\ p_{k,0}(n,n+1) & p_{k,1}(n,n+1) & \dots & p_{k,l}(n,n+1) \end{bmatrix},$$

where $p_{i,j}(n,n+1)$ is the probability of going from state i to state j at the next time instant after n. We call $p_{i,j}(n)$ a one step transition probability.

A discrete time Markov chain is called homogeneous if all $p_{i,j}(n)'s$ are independent of the time parameter n. We shall further consider only homogeneous chains, which simplifies to $\Pi(n+1) = \Pi(n)P$. A discrete time Markov chain is said to have a *stationary probability distribution* $\Pi = (\pi_0, \pi_1, \ldots)$, where each $\pi \geq 0$ and $\sum_{i \in I} \pi_i = 1$, if the matrix equation $\Pi = \Pi P$ is satisfied. A stationary probability distribution is called stationary since the probabilities $\pi_j(n)$ do not change within time, i.e., when $\pi_j(0) = \pi_j$ $(j \geq 0)$, for such a distribution, then $\pi_j(n) = \pi_j$ for all n and j. A Markov chain is said to have a *limiting* probability distribution $\Pi = (\pi_0, \pi_1, \ldots)$ if

$$\lim_{n\to\infty} \pi_j(n) = \lim_{n\to\infty} P[X(n) = j] = \pi_j, j = 0, 1, \ldots.$$

This property is also known as *ergodicity*. A Markov chain with this property is called an *ergodic Markov chain*. Ergodic Markov chains satisfy $\Pi = \Pi P$, hence they have a stationary probability distribution.

Figure 10.1 shows the one-step transition probability matrices in time for the discrete time Markov chain. In the case of an Ergodic Markov chain, these matrices will have no differences for $t \to \infty$.

For continuous state Markov processes we have a transition rate between different states. The transition rate matrix of the continuous time Markov chain is represented by

$$Q = \begin{bmatrix} q_{0,0} & q_{0,1} & \dots & q_{0,l} \\ q_{1,0} & q_{1,1} & \dots & q_{1,l} \\ \vdots & \vdots & & \vdots \\ q_{k,0} & q_{k,1} & \dots & q_{k,l} \end{bmatrix},$$

where $q_{i,j}$ represents the transition rate from state i to state j. Define q_i as $\sum_{i \neq j} q_{i,j}$. The diagonal elements of Q, $q_{i,i} = -q_i$. Let $\pi_i(t)$ be the unconditional probability of the continuous time Markov chain (CTMC) being in state i at time t, then the row vector $\Pi(t)$ represents the transient state probability vector of the CTMC. The behaviour of the CTMC can be described by the following Kolmogorov differential equation:

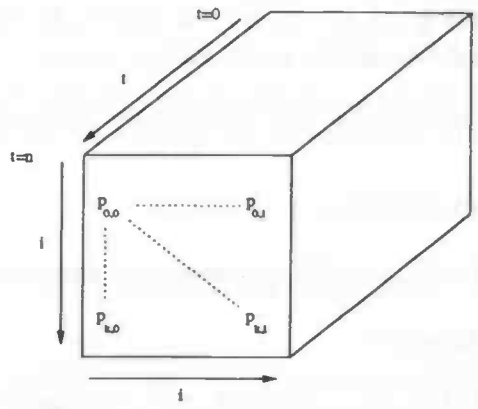


Figure 10.2: The one-step transition probability matrices in time for the continuous time Markov chain

$$\Pi'(t) = \Pi(t)Q.$$

For the limiting probability distribution it follows that $\lim_{t\to\infty} \Pi Q = \lim_{t\to\infty} \sum_{i\in I} \pi_i q_i = 0$, because the system is in steady state.

Figure 10.2 shows the one-step transition probability matrices in time for the continuous time Markov chain. In the case of a Markov chain with a limiting probability distribution, $\Pi'(t)$ will be equal to 0 for $t \to \infty$.

10.3 Markov Reward Models

Markov chains are extended by assigning rewards to the states or to the transitions. In the former case we speak of rate based Markov Reward Models (MRM), in the latter we speak of impulse-based Markov Reward Models. Combinations of both types are of course also possible.

The reward function defined on the states is represented by:

$$r : I \to R.$$

For every state $i \in I$ the reward $r(i)$ signifies the gain received or the reward that will be obtained per unit of time of a system X. Note that we deal here with the rate-based MRM.

The rewards based on the transitions are represented by:

$$r' : I \times I \to R,$$

where $r'(i,j)$ signifies the instantaneous reward accrued whenever a transition from state i to state j, where $i,j \in I$, occurs of a system X. Extensions to cases where the reward functions are time-dependent are possible.

Four types of measures that can be obtained from an MRM can be distinguished:

- Steady state measures
  - $E[X] = \sum_{i \in I} \pi_i r(i)$, the expected reward rate in steady state
- Transient measures
  - $E[X(t)] = \sum_{i \in I} \pi_i(t) r(i)$, the expected reward rate at time t (instantaneous reward)
- Cumulative measures
  - $Y(t) = \int_0^t r(X(\tau))d\tau$
  - $E[Y(t)] = \int_0^t (\sum_{i \in I} \pi_i(\tau) r(i))d\tau$, expected cumulative measures
- Distributions of cumulative measures
  - $F(t,y) = P[Y(t) \leq y]$
    The quantity $1 - F(t,y)$ expresses the probability that the system has done better then y over the specified interval $[0,t]$.

Performability is the amount of work a system can do, including the effects of failures, repairs, and contention. A Performability Model has rewards for each state of the system. For each state this reward indicates the amount of work the system can do when it resides in that state. During a certain period of operation, the system switches from state to state. The average reward of the states in which the system resided in this period, is the amount of work the system has done in this period. The above mentioned measures are predictive measures. The steady state measures, for example, provide information of the expected reward, i.e., amount of work that can be done by the system, when the system is in steady state.

As an example of steady state measures we look at the following simple system. Suppose we have a system with two structure states: *up* (state 1), and *down* (state 0). This system is described by the process $\{X(i) \mid i \in \{0,1\}\}$, where $r(0) = 0$, and $r(1) = 1$. Failures cause the system to switch from state 1 to state 0, and system repairs cause the system to switch from state 1 to state 0. The system's failure rate is denoted by $\frac{1}{\lambda}$, and the repair rate is denoted by $\frac{1}{\mu}$. The system's initial distribution is given by $\Pi_0 = \{1-q, q\}$. The one step transition probability matrix P of this system is given by

$$\begin{bmatrix} 1-\mu & \mu \\ \lambda & 1-\lambda \end{bmatrix}.$$

The expected reward rate at time t, $E[r(X(t))] = \pi_1(t)$. Since we deal with a homogeneous process, $\pi_1(t) = \pi_0(0)P_{01} + \pi_1(0)P_{11} = (1-q)\mu + q(1-\lambda)$. The expected reward rate in steady state, $E[X]$, equals Π_1, where Π is the system's steady-state probability distribution. Intuitively, we can see that our system spends λ of its time in state 0, and μ of its time in state 1. Thus, intuitively, the steady-state probability distribution, Π, is $\{\frac{\lambda}{\lambda+\mu}, \frac{\mu}{\lambda+\mu}\}$.

10.4 Obtaining Performability Models

In this thesis, a reference model for distributed storage architectures is specified in conformity to the RM-ODP. The result is a Distributed Storage System which is capable of creating Distributed Storage Architectures that reflect the performability required by the requesting distributed applications. The Distributed Storage System is specified using ITU-T's Specification and Description Language (SDL). The interest is in finding a general method to translate an SDL specification into a Markov Reward Model. In SDL, systems are modelled as communicating finite state machines. This means that the state of one of these finite state machines, at a certain moment, can depend on states of other finite state machines at that moment. It will be difficult to find the one step probability matrix for these systems, since SDL systems can become very complex. Also, an SDL system is a model of a real system. To be able to evaluate the real system's performability, we need more knowledge of the components that will be used in the real system. This means that the final step of the design process used in this thesis (see Chapter 4) also has to be taken. The parameters of the system's components that affect the system's performability have to be known in order to get realistic performability values. Although the general method to translate an SDL specification into a Markov Reward model will be difficult to find, it is an interesting item for future research.

Chapter 11

Conclusions and Future Research

11.1 Conclusions

In this thesis OSI's Reference Model of Open Distributed Processing ([9, 10, 11, 12]) is used to present a specification of a Distributed Storage System in an open distributed environment. The resulting model provides a frame work for different implementations of distributed storage architectures using various strategies for fragmentation and employment of redundancy. This model can be used for an integrated analysis of performance and reliability, i.e., performability[6, 20], and validation of the performability models through implementations in an open distributed environment, e.g., TINA-DPE or ANSAware.

11.2 Future Research

Finding optimal values for fragmentation and redundancy for a distributed storage system is an NP-hard problem [14]. This problem is generally approached by using predefined tables or heuristic search algorithms. Both approaches require research. Predefined tables record commonly applied values for fragmentation and redundancy in known cases. These cases need to be examined. Heuristic search algorithms require research in the field of AI-based algorithms.

In this thesis ITU-T's Specification and Description Language (SDL) is used to specify the Distributed Storage System from ODP's Engineering Viewpoint. A general way to convert an SDL specification to a Performability Model would ease the construction of performability models of systems specified using SDL. Basically, this requires a general way to convert a Finite State Machine to a Markov Reward Model [6, 20]. How this can be done is an open issue and requires research in the fields of theory of computer science and performability modelling.

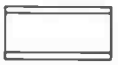
To validate the specified Distributed Storage System it needs to be realised. This requires

the implementation of the objects comprising the Distributed Storage System in an open distributed environment like TINA-DPE or ANSAware. This requires a distributed environment with multiple storage media (e.g., disks, or DBMS's), the realisation of the Storage Manager, and the design and realisation of a distributed demo application, e.g. Video on Demand, to validate the distributed storage system.

Appendix A

SDL Symbols

The next table shows the SDL symbols used in the SDL specification of the distributed storage system in Chapter 9. The names and descriptions of the symbols are borrowed from [4]. The last four columns indicate at which abstraction levels the symbols are used in the specification of the distributed storage system.

Symbol	Description	System level	Block level	Process level	Service level
	Block symbol functional unit, connected to its environment or other blocks by channels	X			
	Channel symbol	X	X	X	
	Create line symbol		X		
	Signal list symbol Lists all possible signals conveyed through a channel	X	X	X	
	Text symbol used to define signals and data types, and to connect channels	X	X	X	X
	Process (m,n) = initial and maximum nr of instances processes are connected to their environment and other processes by channels		X		
	Service symbol functional unit within a process, connected to its environment or other services by channels			X	
	Comment symbol	X	X	X	X
	Start symbol				X
	Task symbol used for assignments				X
	State symbol named state of a process				X
	Input symbol receives specified signal (from specified sender)				X
	Output symbol send specified signal (to specified receiver)				X
	Procedure call symbol				X
	Create request symbol instantiate a new process				X
	Decision symbol				X
	In/Out-connector symbol connects parts of a process graph				X
	Stop symbol terminates the process				X
	Procedure symbol declares a procedure				X

Bibliography

[1] A. Berson. *Client/Server Architecture*. McGraw-Hill Series on Computer Communications, 1992.

[2] Maarten Brugman. Modelleren, specificeren en implementeren van een Virtual Private Network. Master's thesis, University of Utrecht, 1993.

[3] CCITT. Blue Book: Annex F.1 to recommendation Z.100: SDL Formal Definition; Introduction. Technical Report 91.12.1, International Telecommunication Union, November 1988.

[4] CCITT. Blue Book: Functional Specification and Description Language (SDL); Criteria for using Formal Description Techniques. Technical Report 91.12.1, International Telecommunication Union, November 1988.

[5] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma. *SDL Aplications from Protocol Specification*. BCS Practitioner series - Prentice Hall, 1993.

[6] B.R.H.M. Haverkort. *Performability Modelling Tools, evaluation techniques, and applications*. PhD thesis, Twente University, January 1991.

[7] J. Jensen, M. F. Jørgensen, B. B. Nørbæk. Specification of Distributed Telecommunications Services Using SDL. *Proceedings of the TINA'95 Workshop*, February 1995.

[8] M.A. Jacobs. Audio and Video FLows in an Open Distributed Environment. Master's thesis, Twente University, 1995.

[9] JTC1/SC21/WG7. Open Distributed Processing Reference Model, Part 1: overview. Technical report, Draft ITU-T Recommendation X.901 and ISO/IEC 10746-1, July 1994.

[10] JTC1/SC21/WG7. Open Distributed Processing Reference Model, Part 2: Foundation. Technical report, Draft ITU-T Recommendation X.902 and ISO/IEC 10746-2, February 1995.

[11] JTC1/SC21/WG7. Open Distributed Processing Reference Model, Part 3: Architecture. Technical report, Draft ITU-T Recommendation X.903 and ISO/IEC 10746-3, 8 February 1995.

[12] JTC1/SC21/WG7. Open Distributed Processing Reference Model, Part 4: Architectural semantics. Technical report, Draft ITU-T Recommendation X.904 and ISO/IEC 10746-4, February 1995.

[13] K. Kant. *Introduction to Computer System Performance Evaluation.* McGraw-Hill, Inc., 1992.

[14] M.T. Özsu, P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall International Editions, 1991.

[15] L.J.M. Nieuwenhuis. *Fault Tolerance through Program Transformation.* PhD thesis, Twente University, December 1991.

[16] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report 91.12.1, The ASK Group, Inc., September 1991.

[17] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Paterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2), June 1994.

[18] S.G. van der Wal. *Computer Prestatie Analyse; basisrelaties.* Academic Service, 1990.

[19] A.T. van Halteren. Realisation of a Multimedia Stream Object in ANSAware. Master's thesis, Twente University, 1994.

[20] A.P.A. van Moorsel. *Performability Evaluation, Concepts and Techniques.* PhD thesis, Twente University, December 1993.