ELSEVIER

Computer Communications 19 (1996) 30–38

computer communications

Specification of a distributed storage system

Mark A. Nankman, Lambert J.M. Nieuwenhuis

Royal PTT Nederland NV, KPN Research, Groningen, The Netherlands

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 paper, we present the specification of a distributed database function which conforms to the Reference Model of Open Distributed processing (RM-ODP), the ISO/ITU-T standard. The computational model masks the distributed application from the implementation details. Fragmentation and redundancy are used to control the performance and reliability of the distributed storage system. The engineering model is based on the ODP Storage Function and Group Function.

Keywords: Distributed storage; Quality of service; Computational model

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 interest for telecommunications service providers, public network operators and users of the telecommunications services, because the Quality of Service of infrastructures for sharing and distributing information is improved while the cost of communication hardware goes down.

Almost all distributed telecommunications applications include, in one way or another, a distributed database or storage system. In many of these applications, the performance and reliability of this storage system to a large extent determines the Quality of Service (QoS) experienced by the service end-users.

In this paper, we present a distributed storage system specified in accordance with the computational and engineering viewpoints of the Reference Model of Open Distributed Processing (RM-ODP), the ISO International Standard and emerging ITU-T [2–4]. RM-ODP has been adopted by TINA-C (Telecommunications Information Networking Architecture Consortium), which is a worldwide initiative of telecommunications suppliers, network operators and service providers and IT [5,6].

Our objective is to give a general model for distributed storage architectures, in accordance with the ODP Reference Model, based on the basic architectural alternatives: fragmentation and redundancy. These architectural alternatives can both be found, at a hardware level, in disk array architectures [7], and, at a software level, in distributed databases [8].

This paper is structured as follows. First, we propose a classification of distributed storage architectures based on levels of fragmentation and redundancy. We then outline the computational and engineering specification. Finally, conclusions and directions for future research are presented.

2. Distributed storage architectures

This section discusses distributed storage architectures in an open distributed environment. Relevant aspects are the impact of distribution of data on the performance and reliability of these architectures, and the problem of maintaining the consistency of distributed data in the face of concurrent transactions.

To find the common implementation strategies used in distributed storage architectures, we explore how data is distributed in common distributed storage architectures such as Disk Arrays and Distributed Database architectures. Next, we discuss some examples of telecommunications applications using a distributed storage system. Finally, we conclude that there are basically two implementation strategies for distributed storage architectures which can be used to classify these architectures.

SSDI 0140-3664(95)01045-9

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 popularized 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 organize multiple, independent disks 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 on large data accesses and higher I/O rates on small data accesses. Fragmentation also results in uniform load balancing across all disks, eliminating hot spots that otherwise saturate a small number of disks while the majority of the disks sits idle.

However, large disk arrays 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.

2.2. Distributed database systems

According to Ozsu and Valduriez [8], 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 strategies for placing data in distributed databases: 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. On the one hand, data retrieved by a transaction may be stored at a number of sites, making it possible to execute the transaction in parallel. On the other hand, since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized 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.

2.3. Case: Video on demand

An application of a distributed database is a Video on Demand (VOD) service, illustrated in Fig. 1. 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 kbyte/s, and that the average length of a video is one and a half hours. A video would then require a performance of approximately 100 Mbyte/s.

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 Fig. 1, 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 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.

2.4. 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 Fig. 2.

Suppose there are 1,000,000 users of this application over an entire country and the average user makes five transactions per day with an average transaction size of 1 kbyte. Then the average total size of the data flow through the system is approximately 5 Mbyte/day and the required system performance would then be approximately 60 kbyte/s. Also knowing that BS users are generally very patient, you can conclude that the

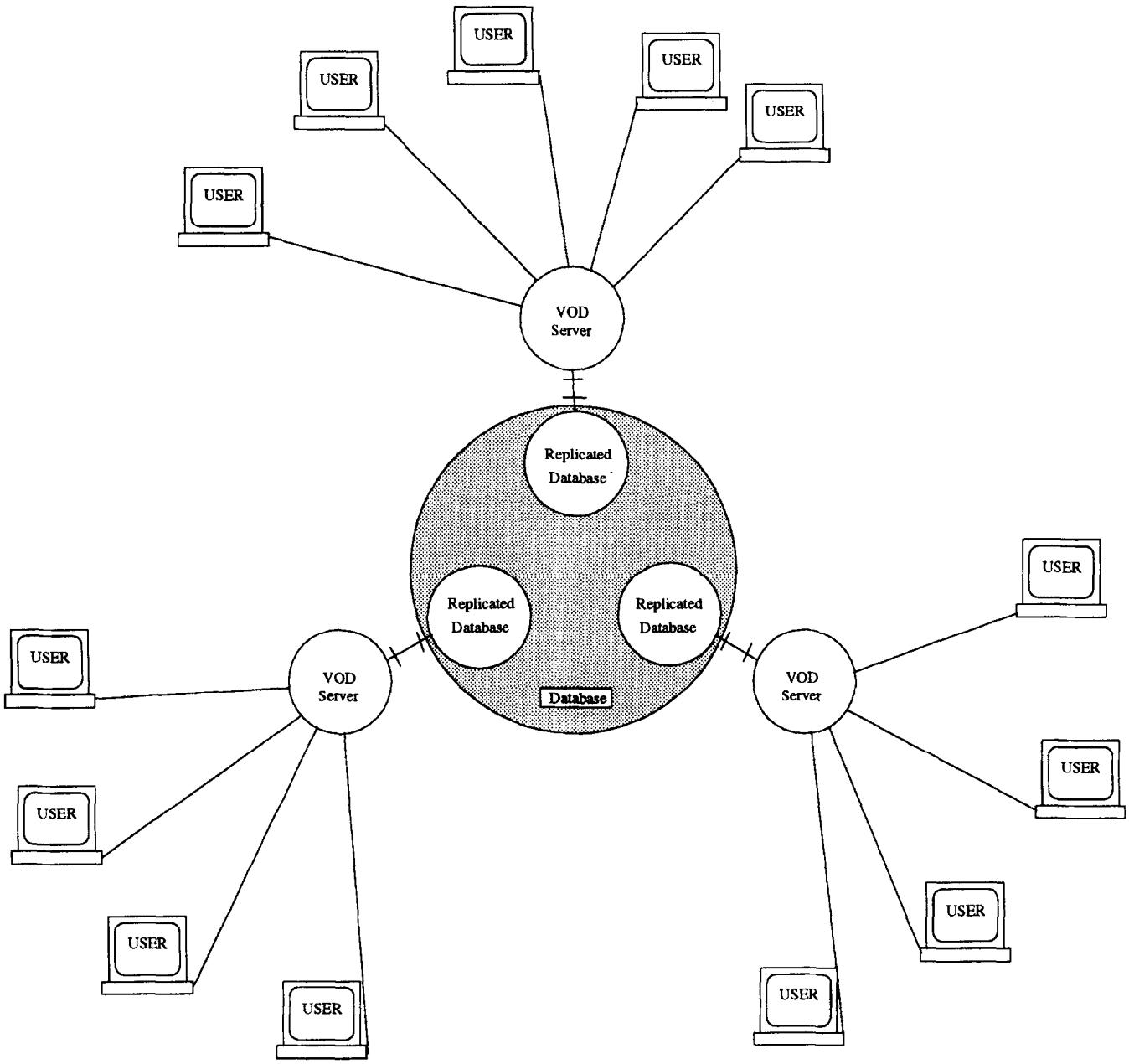

Fig. 1. Video on demand system.

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 Fig. 2, 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 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.

2.5. Implementation strategies

Basically, two strategies can be identified to satisfy the requirements for distributed storage systems used by a distributed application. The first implementation strategy is fragmentation , i.e. the database is subdivided into a number of partitions 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.

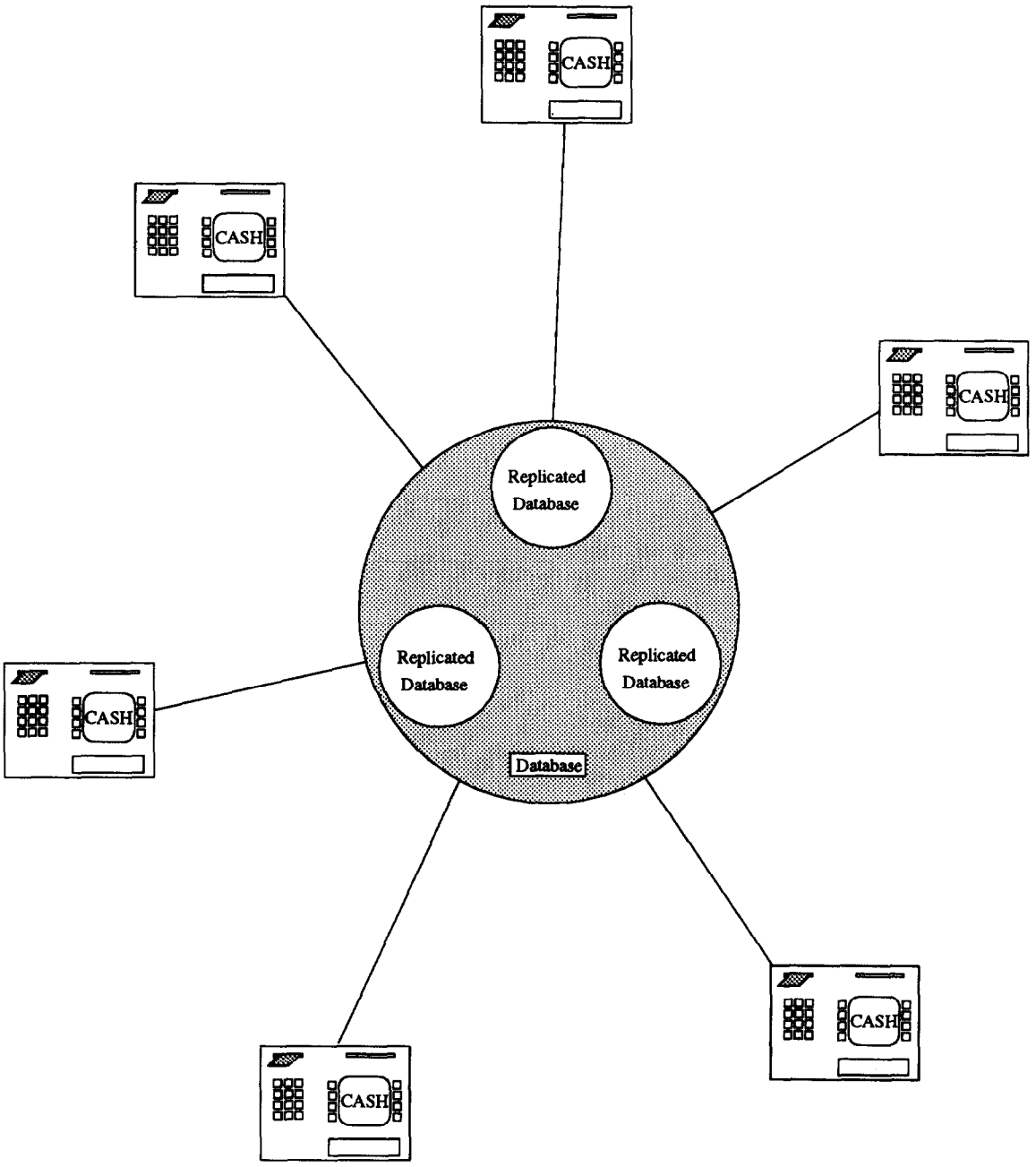

Fig. 2 Banking service.

The second implementation strategy is adding 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 redundancy is applied when the service without redundancy would have failed. In practice, combinations of both strategies can be used, e.g. an increase of performance is achieved if replicas are accessed in parallel for simultaneous read-only transactions.

Basically, the fragmentation and redundancy implementation strategies result in different distributed database architectures. Fig. 3 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 $(r + d)/d$. An architecture based on Triple Modular Redundancy (TMR) and the RAID (Redundant Arrays of Inexpensive Disks) architectures, as described by Chen *et al.* [7], are drawn in the architecture space shown in Fig. 3.

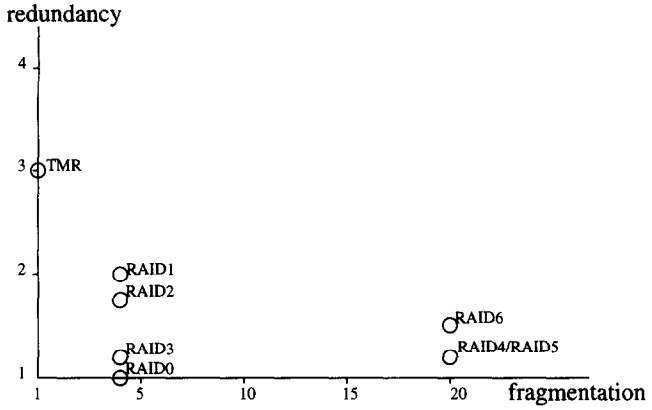

Fig. 3. Architecture space.

3. Computational specification

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. The objective is to abstract from aspects of distribution, i.e. the specification is based on distribution transparency.

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

3.1. Computational model

Fig. 4 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. 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 Unit.

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

- QoS level

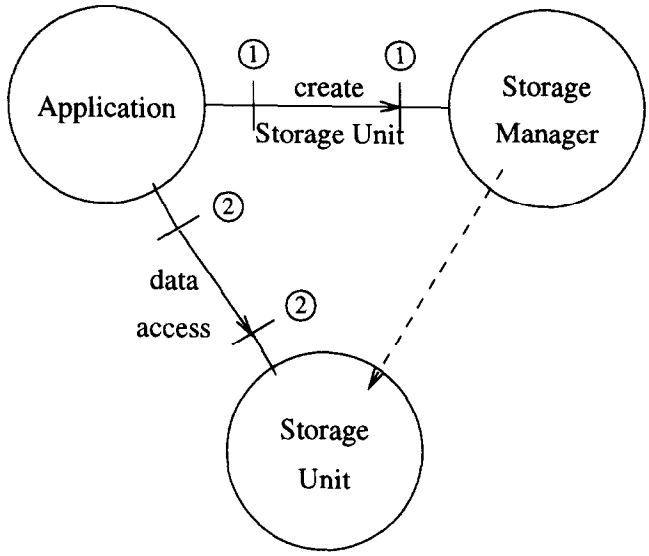

Fig. 4. Computational model of the distributed storage system.

- 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.

3.2 Objects and interfaces specified in IDL

We have used OMG's Interface Definition Language (IDL) [9] 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.

Below, the computational interface templates are specified. From these templates the computational interfaces can be instantiated.

Interface template Storage ManagerIF

Typedef ... Tperformability
Typedef ... TStorageUnitRef

Operations
create_container(bn Tperformability P, **out** TStorageUnitRef SU)

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

Interface template StorageUnitIF

Typedef ... TData

Operations
read(**out** TData D)
write(**n** 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 through this interface.

The templates below specify the object templates from which the computational objects can be instantiated.

Object template StorageManager
Typedef ... TInterfaceRef

Initialisation
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

Intialisation
Init(**out** TInterfaceRef StorageUnitRef)

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.

4. Engineering specification

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 [4].

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 above. The engineering specification is not concerned with the physical components, hence it abstracts from specific components such as operating systems, database management systems and peripheral devices.

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

4.1. Engineering model

In ODP, a number of functions are defined that are either fundamental or widely applicable to the construction of ODP systems [4]. 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. These multi-party bindings are called interaction groups. The rules for the Group Function state that for each Interaction Group the Group Function manages the interaction, collation, ordering and membership.

The engineering model of the distributed storage system is shown in Fig. 5. 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. The Interaction Group is modelled by the channel connecting the Data Repositories and the Storage Unit Manager.

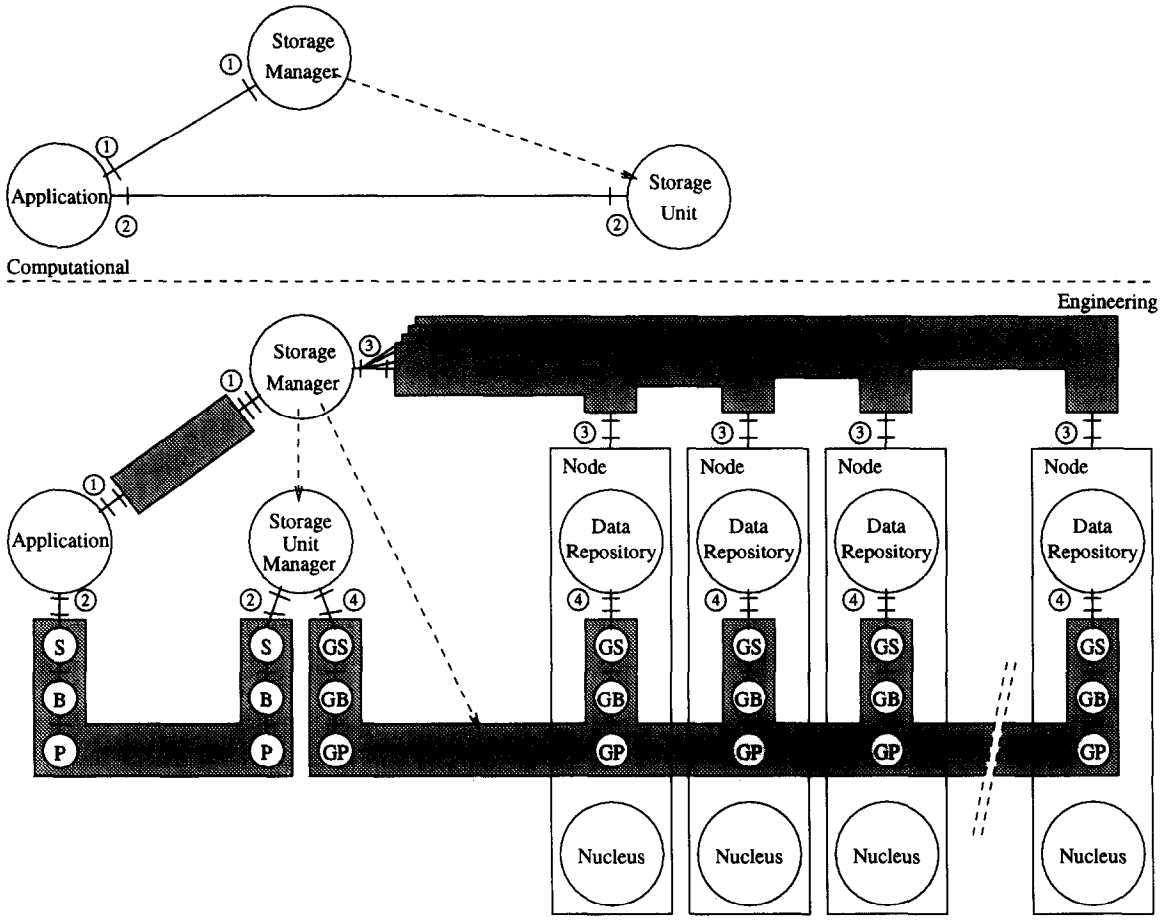

Fig. 5. Mapping of the computational model onto the engineering model of the distributed storage system.

Fig. 5 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 [4]. 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. 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.

4.2. Consistency with the computational model

Fig. 5 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.

4.3. Objects and interfaces specified in IDL

We have used OMG's Interface Definition Language [9] to specify the engineering objects and interfaces comprising the engineering model. Our objective is to implement this system in our experimental environment based

on ANSAware. The IDL templates, as specified below (although details concerning data types also have to be specified), can be directly translated to stubs for the engineering objects and their interfaces. It also provides the opportunity to migrate to future experiments using CORBA 2.0.

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 below. These are new templates to be added to the computational templates defined in above. The engineering Container Interface resembles the computational Storage Unit Interface. Hence, the ContainerIF template replaces the StorageUnitIF template.

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 through this interface.

Below, the object templates from which the engineering objects can be instantiated are specified. 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.

Object template StorageManager (extended)

Typedef ... TInterfaceRef

Initialisation
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

Initialisation
Init(out TInterfaceRef DataRepositoryRef)

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

Initialisation
Init(**out** TInterfaceRef StorageUnitManagerRef)

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.

5. Technology specification

From the ODP technology viewpoint, the system consists of a configuration of physical technology objects. These objects concern software components as well as hardware components. A technology specification describes the implementation of the ODP system in terms of these components, and defines the constraints that should be satisfied by these components.

A technology specification of the distributed storage system describes the properties and constraints for the hardware and software components which represent the abstract engineering objects of the engineering specification.

The implementation of the system was planned for the second half of 1995. Our primary goal is to implement the distributed storage system using ANSAware on a PC-based platform. Besides the distributed storage system, a telecommunications demo application, using the distributed storage system, will be developed.

To be able to use the telecommunications demo application for meaningful measurements of the system's performance and dependability, the demo application should have the following properties. First, for meaningful performance measurements, the demo application should concern sufficiently large transactions. And second, for meaningful dependability measurements, the data concerned with the demo application should be distributed across a sufficiently large number of sites.

6. Conclusions and Recommendations

We have used the RM-ODP to present a specification of a distributed storage system in a distributed environment. The resulting model provides a framework for different implementations using various fragmentation and redundancy strategies. Our next step is to implement the specified distributed storage system on a distributed computing environment, i.e. ANSAware running on PCs and SUN workstations. Our next objective is to develop a parameterized performability model [1,10], which can be validated by experiments based on telecommunications demo applications.

References

[1] B.R.M.H. Haverkort, *Performability modelling tools, evaluation techniques, and applications*, PhD thesis, Twente University, The Netherlands (January 1991).

[2] 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).

[3] 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).

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

[5] W.J. Barr and B.V. Inoue, 'The TINA initiative', *IEEE Comm. Mag.* (March 1993) pp. 70–76.

[6] G. Nilsson, F. Dupuy and M. Chapman, 'An overview of the Telecommunications Information Architectre Networking architecture', *Proc. TINA'95 Workshop*, Melbourne, Australia (February 1995).

[7] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz and D.A. Paterson, 'RAID: High performance, reliable secondary storage', *ACM Comput. Surv., 26* (2) (June 1994).

[8] M.T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs, NJ (1991).

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

[10] A.P.A. van Moorsel, *Performability Evaluation, Concepts and Techniques*, PhD thesis, Twente University, The Netherlands (December 1993).

Mark A Nankman studied computer science at the University of Groningen in The Netherlands. In June 1995 he participated in the First International Workshop on High Speed Computing and Open Distributed Platforms in St. Petersburg. In July 1995 he received his Master's degree in object oriented techniques and telematics, and he is now working at the Department of Communications Architectures and Open Systems of the Research Laboratory of the Royal Dutch PTT.

Lambert J M Nieuwenhuis received a BSc and MSc in electrical engineering and a PhD in computer science from Twente University in 1977, 1980 and 1991, respectively. He joined the Computer Science Department of KPN Research in Leidschendam in 1980. He worked on fault tolerant computing and software reliability. Currently, he is senior scientist with the Department of Communications Architectures and Open Systems of KPN Research in Groningen. His current research interests include IT architectures and platforms and open distributed computing. Dr Nieuwenhuis is also part time professor in Tele-Informatics at the Computer Science Department of the University of Groningen.