

**FTSS: A FAULT-TOLERANT STORAGE SYSTEM SUPPORTING
HIGH AVAILABILITY AND SECURITY IN A DISTRIBUTED
PROCESSING ENVIRONMENT**

Baron O. A. Grey

**December 1984
CSD-840066**

UNIVERSITY OF CALIFORNIA

Los Angeles

**FTSS: A Fault-Tolerant Storage System Supporting High Availability
and Security in a Distributed Processing Environment**

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy
in Computer Science

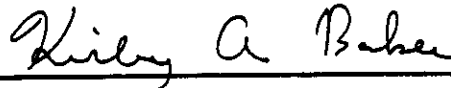
by

Baron Octavius A. Grey

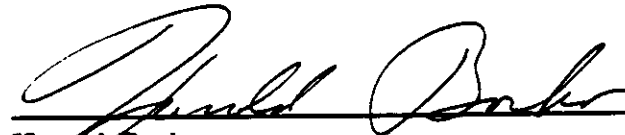
1985

© Copyright by
Baron Octavius A. Grey
1985


The dissertation of Baron Octavius A. Grey is approved.



Kirby Baker



Harold Borko



D. Stott Parker



Algirdas Avizienis



David Rennels, Committee Chair

University of California, Los Angeles

1985

To
Ermine, Gilbert, Thelma, and Connie

TABLE OF CONTENTS

	page
ACKNOWLEDGMENTS	xi
VITA	xii
ABSTRACT OF THE DISSERTATION	xiii
1 INTRODUCTION	1
1.1 The Problems of Distributed Storage	3
1.1.1 Availability	4
1.1.2 Extensibility	6
1.1.3 Complexity	6
1.1.4 Functionality	7
1.1.5 Cost	8
1.2 Centralization as a Solution	8
1.3 The Need for Fault-Tolerance	12
1.4 Related Work	14
1.5 Organization of the Thesis	14
2 ARCHITECTURAL DESIGN ISSUES	17
2.1 Introduction	18
2.2 Separation of Policies and Mechanisms	19
2.3 Application Support	20
2.4 Host Software Minimization	22
2.5 Multiple-Host Support	23
2.6 Reliability and Availability	24
2.7 Extensibility	25
2.7.1 Hardware Extensibility	26
2.7.2 Software Extensibility	26
2.8 Security	27
2.9 Resiliency	28
2.9.1 The Transaction Paradigm	29
2.10 Efficiency	31
2.11 Summary	31
3 SOFTWARE SYSTEM MODEL	32
3.1 Strawman Storage System	32
3.2 A Generalized Software Model	35
3.3 Software Design Overview	36
3.3.1 Security-Driven Models	36
3.3.2 A Kernel-Based Architecture	37
3.3.2.1 The Kernel	39
3.3.2.2 The Subsystem Level	41
3.3.2.3 The SMS Interface	42
3.4 Implementation of SMS	43
3.4.1 Kernel Implementation	43
3.4.1.1 An Overview	44
3.4.1.1.1 Memory Management	46

3.4.1.1.2	Process Management	49
3.4.1.1.3	I/O Management	50
3.4.1.2	Kernel Modules	51
3.4.1.2.1	Capabilities	51
3.4.1.2.2	Objects	52
3.4.1.2.3	Data Abstraction	54
3.4.1.2.4	I/O	55
3.4.2	Subsystem Implementations	55
3.4.2.1	Process Management Subsystem	56
3.4.2.2	Message Subsystem	59
3.4.2.3	Security Subsystem	61
3.4.2.4	Transaction Subsystem	63
3.4.2.5	Object Management Subsystem	65
3.4.2.6	System Monitor Subsystem	67
3.5	Using the System	68
3.5.1	The Execution Environment	68
3.5.2	Abstract Data Types	73
3.6	Building an Operating System Interface	75
3.7	Summary	84
4	SECURITY ISSUES	86
4.1	Introduction	86
4.1.1	Limiting the Scope	88
4.2	The Basic Mechanism	89
4.2.1	Capability Sealing	90
4.3	The Mechanism in Use	95
4.3.1	Protection	97
4.3.2	Revocation	97
4.4	Implementation Issues	101
4.4.1	Kernel Support	101
4.4.2	User Level Support	106
4.5	Summary	110
5	AVAILABILITY ISSUES	111
5.1	Conceptual Framework	111
5.2	Hardware Reliability versus Information Availability	113
5.3	Approaches to Achieving High Availability	114
5.4	An Information Availability Model	116
5.4.1	Model Definition	117
5.4.1.1	Network System Operation	120
5.4.1.2	Hardware and Software Considerations	122
5.4.2	Model Development	122
5.4.3	Reliability Estimation for Storage Sites	123
5.4.3.1	The Processor Subsystem	123
5.4.3.2	The Storage Subsystem	127
5.4.3.3	The Storage Site Subsystem	129
5.4.4	Network Availability	129
5.4.4.1	Simple Topologies	130
5.4.4.2	Periodically Renewed Networks	133
5.4.4.3	Complex Topologies	137
5.5	Discussion of Results	142
5.6	Summary	143

6	FAULT-TOLERANT STORAGE SYSTEM ARCHITECTURES	144
6.1	Introduction	144
6.2	Architectural Considerations	147
6.2.1	Extensible Topologies	147
6.2.1.1	Tree Topologies	148
6.2.1.2	Bus Topologies	149
6.3	A Highly Reliable Architecture for BSN's	149
6.3.1	Hierarchical Tree Systems	150
6.3.2	Fault-Tolerant Hierarchical Tree Systems	152
6.3.3	The Basic Topology	154
6.3.4	Properties of the Topology	156
6.3.5	Routing	159
6.3.6	Extensibility	160
6.4	Fault-Tolerance and Reliability	162
6.4.1	Reliability Analysis	162
6.4.2	Discussion of Results	169
6.5	Applying the Topology in BSN's	170
6.6	Summary	172
7	THE OBJECT STORE: FAULT-TOLERANCE AND OTHER ISSUES	173
7.1	Mapping FTSS onto a Fault-Tolerant Tree Architecture	173
7.2	Object Representation	175
7.2.1	Object Representation in the IL	176
7.2.2	Object Representation in the BL	177
7.2.3	Consistency	177
7.3	Location	181
7.4	Accessing Object Blocks	183
7.4.1	Write/Update Access	183
7.4.2	Read Access	188
7.5	Migration	190
7.6	Deletion	194
7.7	Recovery	196
7.7.1	Recovery Management	197
7.7.2	Site Failures	197
7.7.3	Device Failures	199
7.7.4	Link Failures	200
7.7.5	Media Failures	200
7.7.6	Host Failures	202
7.8	Summary	202
8	IMPLEMENTATION OF FTSS	203
8.1	Introduction	203
8.2	The Basic System	204
8.3	FTSS Hardware	208
8.3.1	The Top Level	208
8.3.1.1	Bus Architecture	212
8.3.1.1.1	The Main Bus Architecture	216
8.3.1.1.2	The I/O Bus Architecture	220
8.3.1.2	Processor Modules	221
8.3.1.3	RAM Buffer Modules	224
8.3.1.4	Network Interface Modules	228
8.3.1.5	Internetwork Interface Modules	230

8.3.1.6 Other Modules	232
8.3.2 The Internetwork	232
8.3.2.1 Storage Site Architecture	233
8.3.2.2 Internetwork Communications	237
8.4 FTSS Software	241
8.4.1 Distributed SMS Architecture	242
8.4.2 Effect of FT on SMS Architecture	243
8.5 Summary	243
9 PERFORMANCE ISSUES	246
9.1 Introduction	246
9.2 Bandwidth Considerations	247
9.3 The Simulation Model	248
9.3.1 Model Description	248
9.4 Using the Model	252
9.4.1 Varying LAN Bandwidth	256
9.4.2 Varying the IL Bandwidth	257
9.4.3 Varying the Processor Bandwidth	258
9.4.4 Varying the RAM Buffer Bandwidth	260
9.4.5 Other Variables	260
9.5 Summary	262
10 CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH	264
Appendix A Diameter of $T_{2,p,1}$ Trees	268
Appendix B A Simple Distributed Routing Algorithm for $T_{2,p,1}$ Trees	270
Appendix C FTSS Data Link Control Procedure	272
Appendix D OBJECT-LEVEL ALGORITHMS	274
Appendix E EXAMPLES OF OPERATING SYSTEM PRIMITIVES	278
REFERENCES	284

LIST OF FIGURES

	page
Figure 1.1: Distributed Local Network Architecture	4
Figure 1.2: Distributed System with File Servers	9
Figure 1.3: Backend System with Distributed Storage System Components	11
Figure 1.4: Backend Storage System with Independent Communication Subsystem	12
Figure 3.1: Architecture of SMS	34
Figure 3.2: Multi-Level Kernel Architecture	39
Figure 3.3: Functional Block Diagram of the SMS	44
Figure 3.4: Structure of a Process Object	71
Figure 3.5: Process Object with Several Contexts	73
Figure 3.6: Tree of Object Types	74
Figure 3.7: Snapshot of Login Authentication	80
Figure 3.8: Object Update	81
Figure 3.9: Snapshot During Subsystem Creation	83
Figure 4.1: Sealed Capability	96
Figure 4.2: Organization of the SYS_SEAL object	109
Figure 5.1: Distributed System of Storage Sites	113
Figure 5.2: Markov State Diagram of Information Availability	119
Figure 5.3: Time Diagram Showing Operation of the LCN	121
Figure 5.4: Markov State Diagram for the Processor Subsystem	124
Figure 5.5: Plot of Reliability and Availability for the Processor Subsystem	125
Figure 5.6: Effect of Crash Coverage on Availability	126
Figure 5.7: Plot of Reliability and Availability for the Storage Subsystem	128

Figure 5.8: Plot of Reliability and Availability for a Storage Site	130
Figure 5.9: Revised PRC Model	134
Figure 5.10: Effect of Site Dependencies on Network Availability	137
Figure 5.11: Multi-Connected Computer Network	139
Figure 5.12: Bridge Subnetwork of an Application Program	140
Figure 6.1: A $T_{2,4,1}$ 1-FT Symmetrical Hierarchical Tree	155
Figure 6.2: Two ways to extend $T_{2,4,1}$ trees	161
Figure 6.3: System reliability under different failure assumptions	167
Figure 7.1: Conceptual Hardware Architecture of FTSS	174
Figure 7.2: Object Block Location Hierarchy	182
Figure 7.3: Object Update Using Multiple Versions	185
Figure 8.1: Logical Organization of the Top Level	209
Figure 8.2: A fault-tolerant busing structure based on distributed buses	214
Figure 8.3: A fault-tolerant bus architecture based on monolithic buses	215
Figure 8.4: FTSS Main Bus Architecture	218
Figure 8.5: Processor Module Hierarchy	221
Figure 8.6: Self-Checking Processor Module	223
Figure 8.7: Logical Architecture of the RAM Buffer	224
Figure 8.8: Modular Organization of the RAM Buffer	228
Figure 8.9: Logical Architecture of a Storage Site	234
Figure 8.10: Storage Site Architecture	235
Figure 8.11: Logical view of the distributed SMS	244
Figure 9.1: Block Diagram for Simulation Model	249

LIST OF TABLES

	page
Table 5.1: ARIES Parameters for the Processor Subsystem	126
Table 5.2: ARIES Parameters for the Storage Subsystem	128
Table 5.3: Time-Dependent Availability for a Bus Network	132
Table 5.4: Steady-State Availability versus Replication Factor	136
Table 5.5: ARIES Parameters for Site Dependencies	138
Table 5.6: Steady-State Availabilities for Various PRC Partitions	138
Table 5.7: Availability for Multi-Connected Topology	142
Table 6.1: Reliability of a $T_{2,4,1}$ Tree	168
Table 6.2: Reliability Improvement of $T_{2,4,1}$ Tree	168
Table 6.3: Mission Time Improvement of $T_{2,4,1}$ over Non-Redundant Tree	168
Table 6.4a: Fault Coverage and Failure Rate at each Level	169
Table 6.4b: $T_{2,4,1}$ Tree with Different Failure Rates and Coverages at Each Level	169
Table 9.1: Simulation Model Variables	252
Table 9.2: Baseline Simulation Values	253
Table 9.3: Baseline Simulation Results	254
Table 9.4: Effect of LAN Bandwidth on Response Time	256
Table 9.5: Effect of IL Bandwidth on Response Time	258
Table 9.6: Effect of Processor Bandwidth on Response Time	259
Table 9.7: Effect of RAM Buffer Bandwidth on Response Time	261

ACKNOWLEDGMENTS

This dissertation is the result of the encouragement, support, and dedication of many individuals. Although it is not possible to acknowledge them all here, they have my heartfelt thanks. The following people I would especially like to thank.

Many thanks to my dissertation advisors, Prof. David Rennels and Prof. Algirdas Avižienis who have guided me through the delicate pathways of fault-tolerance. My gratitude is also extended to the other members of my dissertation committee, Prof. K. Baker, Prof. D. S. Parker, and Prof. H. Borko. I am especially grateful to Prof. Rennels who generously gave of his time through many technical and philosophical discussions.

All the members of the Research in Distributed Processing group and the other Research Assistants with whom I have had the pleasure to work, deserve special thanks for all the assistance and support they have provided.

I am deeply indebted to my parents who motivated me to study abroad, and provided me with mental and financial support in those critical moments. Without their help, this would not have come to pass.

VITA

- October 3, 1950 -- Born, Kingston, Jamaica
- 1973 -- B.S., Northrop University, Inglewood, California
- 1978 -- M.S., Northrop University, Inglewood, California
- 1975-1981 -- Assistant Professor
Electronic Engineering Department
Northrop University, Inglewood, California
- 1981-1983 -- Associate Professor, Chairman
Computer Science Department
Northrop University, Inglewood, California
- 1983-1985 -- Post Graduate Research Engineer
Department of Computer Science
University of California, Los Angeles

PUBLICATIONS

- [Grey84] B.O.A. Grey, A. Avizienis, and D. Rennels, "A Fault-Tolerant Architecture for Network Storage Systems," in *Proceedings 14th International Conference on Fault-Tolerant Computing*, Kissimmee, Florida: June 1984, pp. 232-239.

ABSTRACT OF THE DISSERTATION

**FTSS: A Fault-Tolerant Storage System
Supporting High Availability and Security
in a Distributed Processing Environment**

by

Baron Octavius A. Grey

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor David Rennels, Chair

In this thesis, a new approach to the design of reliable shared storage systems for local computer networks is presented. Traditional approaches have achieved reliability for the information stored in the system via replication, storing multiple copies at different sites in such a way that site failures will not immediately lead to unavailability. Other traditional approaches provide specialized file server systems and backend networks which attempt to provide higher reliability and performance through dedicated hardware and software. Our approach is quite different. We provide high reliability through the application of hardware fault-tolerance, high performance through the use of a specialized hardware architecture, and simple information management mechanisms through the use of centralization. In the system developed here, which we call FTSS, information is stored as objects, and accessed via capabilities which

are managed independently by the system.

FTSS is intended to be used in environments where fast access to large volumes of information is required. This can range from mainframe systems that only need to use it as a repository from which data is downloaded for local access when necessary, or at the other extreme, simple workstations that do not possess any online storage of their own. FTSS provides facilities that allow each host to independently define a storage environment tailored to the characteristics of its local operating system. It is arbitrarily extensible so that more storage can be added as needed. Information is automatically managed in a user-transparent way; an object is stored or accessed without any regard for its actual location or device-dependent characteristics in the system.

An important motivation for the development of FTSS is the need to provide reliable and easily-accessible storage for large numbers of distributed workstations, especially given the current trend towards workstations in engineering and office automation. FTSS portends to be an important means of allowing machines of widely differing storage and computational requirements to share an important resource with all the cost benefits that accrue from economies of scale.

CHAPTER 1

INTRODUCTION

As the cost of computing and computers decreases, there is more demand for local autonomy by system users. This has partially resulted in a proliferation of distributed processing systems consisting of a set of autonomous nodes or computing sites typically interconnected with a high-bandwidth communication subsystem over which the nodes may communicate and share information. As this decentralization takes place, new problems arise which require novel solutions in order to provide high performance and a reasonably efficient level of service to the user.

The need for decentralization is driven by many factors. Traditionally, computing resources have been provided in a centralized computing facility independently managed by some central authority. The reasons for this centralization was primarily the cost benefits of sharing expensive resources and the high performance that resulted from large mainframe computers. Decentralization was an effort to allow diverse users the opportunity to control and manage their own resources autonomously. Despite the decentralization of resources, there is still a need for information and resource sharing; hence, the interconnection of the computing sites in local computer networks.

Despite the trend towards decentralization, there are still cases where resources are so expensive that they must be shared. One such case is in the processing capabilities of the system. It is often the case that an autonomous processing site does not possess the capability to execute processes either because of the inherent size or the speed of the processor. In such cases, it is common to establish mechanisms whereby a processing site may execute its processes at another site, either directly (that is, with *a priori* knowledge of the mechanisms) or indirectly. Another case is in the storage capacity of a site. Often processing sites do not possess enough reliable online storage for the semi-permanent repository of information. In such cases it is common to share storage between several sites. Perhaps the most important reason for shared storage is the requirement for large online databases; such databases usually have constraints on size and performance which precludes their implementation on the existing computing sites. The unreliability of storage often results in mechanisms whereby information is replicated at several sites in the hope that storage-related failures will not make information unavailable for long periods of time.

The need to share storage system resources is even more important when one considers the recent trend towards *workstations*. Workstations are low-cost high-performance computing nodes that, because of their small physical size, are ideally suited for decentralized resource management. Since it is not usually cost-effective to provide large amounts of online storage for each workstation, there is an evolution towards *diskless* workstations which, of necessity, must obtain their storage by sharing from some centralized repository.

The decentralization of computing resources has brought about a need for information sharing which requires techniques not found in centralized facilities. Primary among these techniques are mechanisms for managing shared storage distributed among several computing nodes. This thesis examines these mechanisms and proposes alternative solutions to those that have been developed in the past. The primary issues are storage system reliability and availability, and information security. The solution proposed is to centralize storage system resources and to provide high reliability by using fault-tolerance design techniques for a storage system architecture. We propose a storage system architecture based on both file-server and backend network principles. Security is provided by extending existing techniques for the design of secure operating systems using the notions of *objects* and *capabilities*.

1.1 The Problems of Distributed Storage

The problems associated with *distributed storage* are best illustrated by considering Fig. 1.1 which shows a typical local network architecture of autonomous computing sites. Each computing site consists of a Network Interface (NI), a Host Processor (H), and a Storage Subsystem (S). The storage sites are interconnected with a high-speed contention-based bus to facilitate information sharing. By *distributed storage* we allude to the fact that if information is to be shared between the various computing sites, each host must take part in this sharing for that portion of the information which is stored in the storage subsystem under control of the host. For example, if a host wishes to access information which is stored at another site, not only must the "local" host be involved, but the host at the remote site as well. If one imagines the sum of all the information stored in each storage site as the totality of information in the

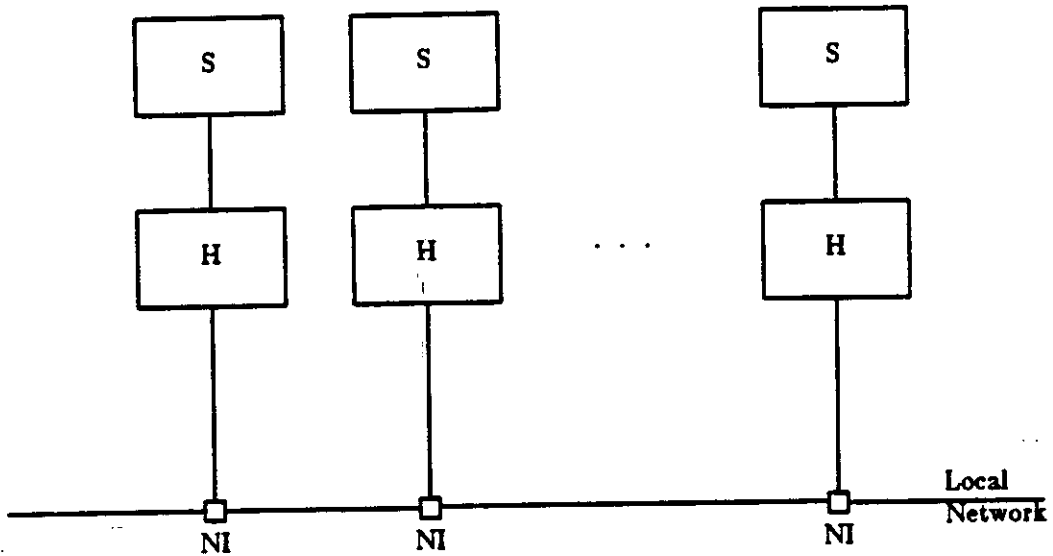


Figure 1.1: Distributed Local Network Architecture

system, it is easy to see the nature of the distribution of storage - each physical storage subsystem S represents a (distributed) part of the system's information. It is the very nature of this physical distribution of storage in distributed processing systems that lead to a plethora of problems not encountered¹ in centralized storage systems.

1.1.1 Availability

In general, users would like their stored information to be available a high percentage of the time, if not ALL the time. If we consider a single computing site for the moment, it is interesting to note that the reliability of *information* in the storage subsystem S is *lower* than the reliability of either NI, H, or S. The reason for this is because NI, H, and S are in *series* as far as reliability is concerned. Thus, the reliability of information in S is the *product* of the reliabilities of NI, H, and S. The significance of this is the fact that informa-

¹At least, not to the same extent.

tion in S can become unavailable either because of a failure in NI (due to hardware or software), a failure in H (either due to hardware or an operating system crash), or a failure in S itself (again due to hardware, such as a disk controller failure, or software). If information is not replicated in other computing sites, then the availability of this information is clearly a function of the reliability of the site components, assuming that system repair is not immediate.

One way to obtain higher availability in the presence of unreliable site components is to replicate information at two or more computing sites; if a site fails, there is some non-zero probability that the desired information can be accessed from another site. This, of course, depends upon the *replication factor*, and on the probability that not all the sites at which a replicate is stored will fail simultaneously. However, replication introduces the problem of maintaining consistency among the replicates. Several novel schemes have been proposed for maintaining data consistency in the face of site or component failures and network partitioning. However, these schemes depend to a large extent on the compatibility of storage system architectures between the storage sites. If, in the worst case, each host H has a unique storage system architecture, then it is virtually impossible to store information remotely at another site. Since we naturally expect a diversity of processor types in any autonomous network, the success of replication as the sole means of achieving high availability is questionable.

1.1.2 Extensibility

One of the key requirements of any storage system is the ability to easily extend the amount of online storage without perturbation of the system or loss of availability. This is not easy to do with a distributed storage architecture. The problem is caused by the close coupling of the storage subsystem S and the host processor H. In general, the capacity of S is a function of the host processors' operating system. That is, once the capacity of the operating system to address online storage is reached, there is no possibility of extending the storage subsystem's capacity without significant hardware and software modifications.¹ In view of the trend towards autonomous workstations, it does not appear that distributed storage architectures are practical in view of their lack of extensibility.

1.1.3 Complexity

Distributed storage architectures require complex storage management mechanisms which offer little opportunity to optimize or "tune" for performance and efficiency gains. If information is being shared between two computing sites, it is necessary to provide mechanisms at both sites that must cooperate in information transfers. Assuming that the hosts run identical software processes, there is a fixed processing overhead which cannot be avoided; each computing site will observe a degradation in performance which is the overhead of remotely accessing this information. Said in another way, there is no way in which a remote access can appear to have been performed locally to a host processor. While it has been argued that the incidences of

¹Some systems solve this problem by simply adding another compatible computing site to the network and sharing storage with it.

remote access are few, there are applications in which the overhead of a remote access is a high price to pay indeed.

The added complexity in the storage management components of host operating systems is contrary to the philosophy that simplicity often leads to improved reliability. It would seem logical that schemes for offloading storage management responsibilities from host processors in local computer networks should be pursued in order to allow higher reliability in operating system software.

1.1.4 Functionality

In any computing system, there is a need to provide automatic archiving and incremental backup as a means of preserving the state of the system. These tasks are complicated by a distributed storage system in which naming and location are local to each site; this necessitates complex intersite protocols for mapping names to locations.

Besides the sharing of physical resources, it is useful in a distributed processing system to share application programs as well. For example, large database management programs can be shared, resulting in a significant savings in the use of system resources. Another possibility is the ability to perform dynamic load balancing by automatically migrating processes between homogeneous hosts. Again, these services are not easy to implement in a system in which naming and location mechanisms are not centralized.

1.1.5 Cost

Unlike the case for centralized computing centers, distributed storage architectures do not allow one to take advantage of the benefits of the economies of scale possible through resource sharing. Since each autonomous processor might use resources which cannot be directly used by other sites, there is no advantage to "pool" resources economically - this is often dictated by the requirements of the host operating system for resources with prespecified characteristics. Because of the decreasing cost per bit of online storage, it would seem that every effort should be made to take advantage of the economies of scale in storage system configurations.

1.2 Centralization as a Solution

There are two possible solutions to the problems of distributed storage architectures. The first is to create a *logically centralized* storage system on top of the distributed storage. Storage management is distributed among the computing sites in such a way that the naming and location of information in the storage system is transparent to the host processors. That this is not an elegant solution is immediately obvious when one realizes that all the attendant problems of distributed storage remain, only the details are hidden from the user. This approach has been taken by several researchers because it is relatively easy to retrofit in existing distributed systems.

The second solution is to create *physically centralized* facilities for storing and managing information. This approach allows one to unbundle storage from the host processors at the computing site and to manage this storage as an independent entity. By centralizing storage system resources in this way, it

is possible to either eliminate or significantly reduce most of the problems associated with distributed storage systems. There are two major directions that have been taken by researchers in this area: The first gives rise to what are called *File Server* systems, and the second to *Backend* systems.

The attributes of file server systems is the unbundling of storage from host processors to specialized *servers* attached to the same communication network. This is illustrated in Fig. 1.2.

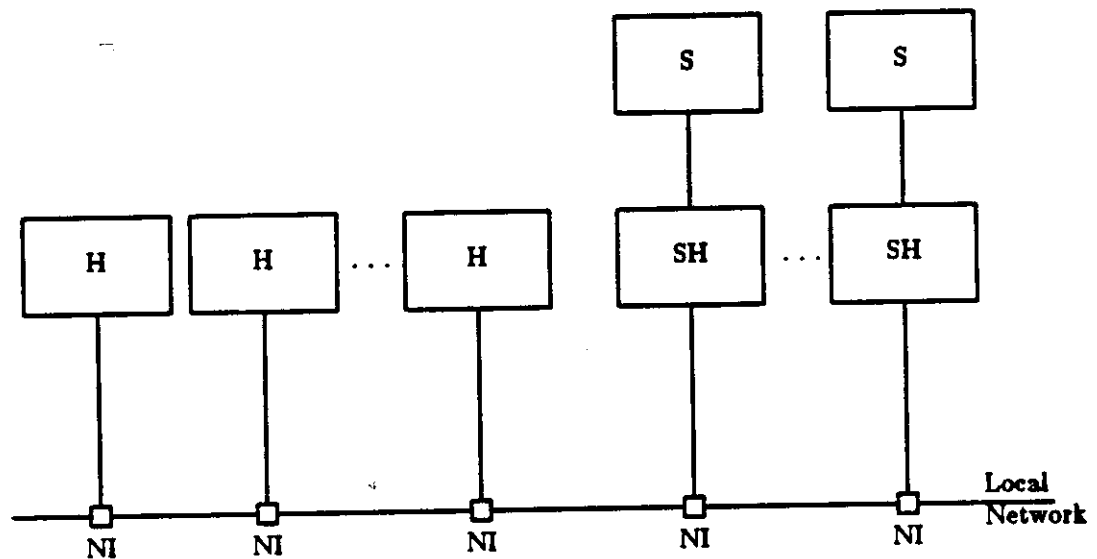


Figure 1.2: Distributed System with File Servers

Each file server is a series connection of NI, Server Host (SH), and storage S. The set of file servers in the network can indeed be treated as logically centralized from the rest of the network. Thus, host processors no longer need to take part directly in storage management (they simply make requests to the servers). The file server approach has some disadvantages however. First, storage is still isolated from the network by the SH's and NI's (a failure in either will lead to unavailability). Second, it is common practice to use off-the-shelf computers as SH's (at times, the same types as the host processors to take

advantage of cost benefits); there is therefore very little opportunity to improve the overall reliability of the system except by either replicating information in several servers, or to replicate the entire host processors if they should prove to be a reliability bottleneck. In short, it is difficult to apply reliable design in file server systems. Third, they do not completely solve the problems of extensibility, functionality, and cost although they do represent a significant improvement over the distributed storage case.

Backend storage systems are in the same spirit as file server systems - they unbundle storage from the host processors so that the hosts do not have to perform global storage management. The significant difference between the two approaches is that backend systems attempt to optimize the performance, extensibility, and functionality of the resulting storage system by distributing functionality in such a way that critical functions are relegated to specialized hardware components; these components are then directly connected to the network. An example of this structure is shown in Fig. 1.3 where SH represents Server Host processors and C the specialized backend components. It is typical, for example, to have specialized components such as authentication and directory servers, Random Access Memory (RAM) buffers, job allocators, and so on. Storage components S are connected directly to the network via NI's. The primary advantage of this organization is the ease with which the system might be extended since new system components can be directly attached to the network (possibly without perturbing existing components or resulting in loss of availability). There is a disadvantage in this particular organization however: it stems from the fact that since all components are connected directly to the network, network bandwidth is consumed whenever the components must com-

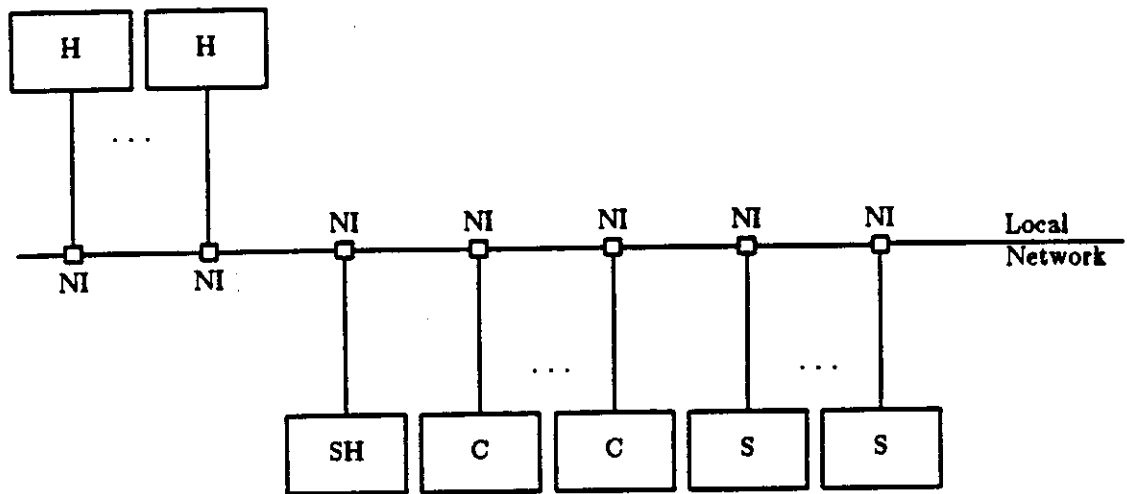


Figure 1.3: Backend System with Distributed Storage System Components

municate. Since backend storage management implies significant communication overhead, this might lead to some compromise in the bandwidth requirements of the computing sites. Since, in most contemporary local network architectures, there is a limited bus bandwidth, some effort should be made to minimize the impact of the storage subsystem upon bus bandwidth. One scheme for achieving this is illustrated in Fig. 1.4.

The compromising of network bandwidth is avoided by providing an independent busing structure for the backend components. This structure has all the advantages of the previous structure, and more, since it permits arbitrary extensibility and performance tuning without affecting the ability of the host processors to meet critical intersite communication requirements.

Much of the work that has been done on file server systems is applicable to backend systems, particularly in the software processes needed for storage management. We present in a later portion of this thesis an architecture which

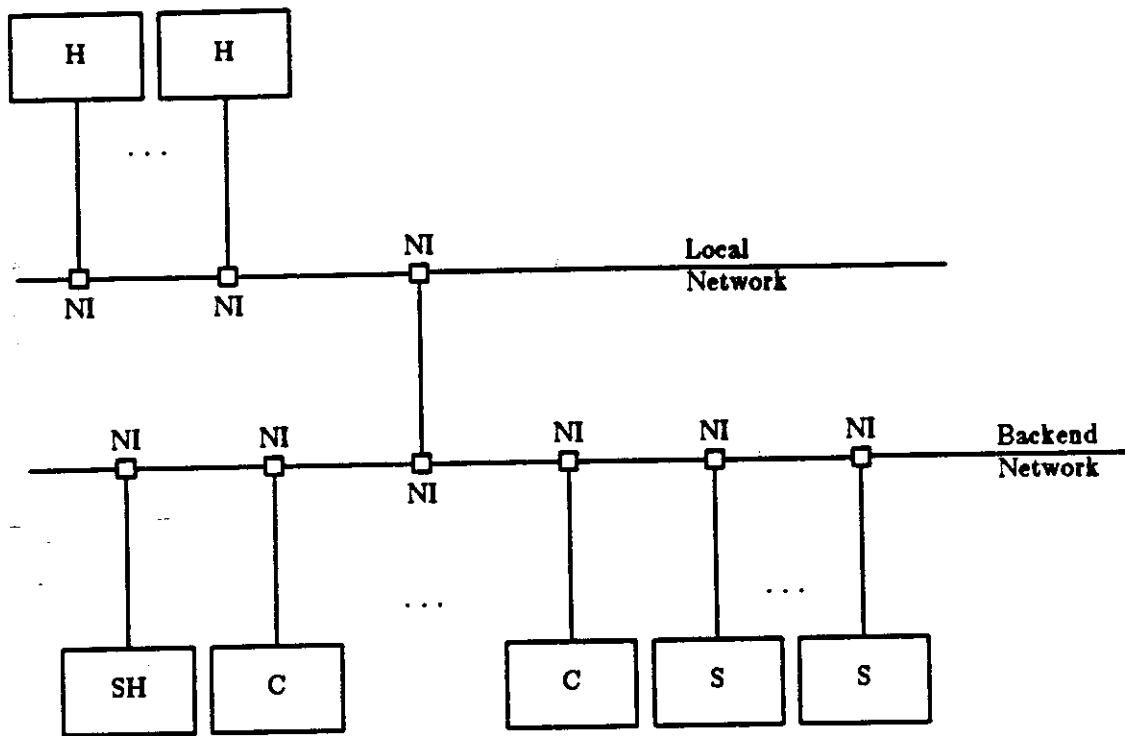


Figure 1.4: Backend Storage System with Independent Communication Subsystem

is somewhat of a cross between a file server and a backend system. It uses the organizational principle of Fig. 1.4 along with established storage management principles developed for file server systems.

1.3 The Need for Fault-Tolerance

The goal of any shared centralized storage facility is to provide an environment in which human, software, and hardware resources can intercommunicate and be shared in a coherent way. Because of the centralized nature of the storage, there is a greater need for reliability and availability since unavailability of the facility affects all hosts (this was not the case for distributed storage where a failure at one or more sites might affect only a few hosts). Certainly, in the case where the hosts are autonomous workstations that depend

upon the availability of storage, it is important that the facility maintain a high steady-state availability.

There are two approaches to achieving high availability in a distributed system. One is to use components that never fail, and the other is to provide some form of redundancy. Since it is impossible to design systems that never fail, given a finite amount of time and resources, the latter approach is universal. One approach to the application of redundancy in reliable design is through the use of fault-tolerant design principles.

The approach taken to achieve high availability in distributed and file server systems is to replicate information in locations which have independent failure probabilities for the classes of faults expected. If a failure occurs, information may be accessed from one of the replicated sources until the failed unit is repaired, at which time an effort is made to reconcile any inconsistencies that resulted from updates during the failure interval. The fallibility of this approach is that it is not always easy to perform these reconciliations when information is spread out over multiple computing sites. The technique, however, works well for those classes of failures in which information is not lost during the failure interval (as, for example, when a host operating system crashes). However, if recovery from hardware failures in which data might be lost is critical to the system's operation, then some form of hardware fault-tolerance is required. In this thesis we shall investigate an approach to the design of a Shared Centralized Fault-Tolerant Storage System (FTSS) that can be used to realize the benefits of information sharing in a centralized repository.

1.4 Related Work

A number of works have markedly influenced the direction taken by this research. First, the choice of a centralized architecture was directly related to our first-hand observation of and experiences with our own distributed system LOCUS [Pope81]. LOCUS, however, espouses a logically centralized file system built on top of a distributed system of autonomous machines, each of which manages a portion of the file system. LOCUS is an excellent example of the importance of transparency; it has also influenced our approach in the organization of the Storage Management System in chapter 3.

Our choice of an object model in chapter 3 was heavily influenced by the implementations of Hydra [Wulf81], the IBM System/38 [IBM78], and iMAX [Khan81], and the excellent work by Jones [Jone73] on protection in operating systems. Various backend and file server systems have influenced our implementation approach to FTSS. Notable among these are Swallow [Svob81], the Cambridge file system [Dion80], the early work done at Xerox on WFS [Swin79], and the Octopus network at Lawrence Livermore [Thor80]. Our approach to fault-tolerance using off-the-shelf components where possible, and the use of self-checking designs is influenced by the work of Avižienis [Avi78] and Rennels [Renn80]. Finally, the material on capability sealing in chapter 4 is derived from the works by Redell [Rede74] and Gifford [Giff82].

1.5 Organization of the Thesis

The rest of this thesis is organized in the following way. Chapter 2 discusses some key issues relating to the hardware and software architecture of shared centralized storage systems, with emphasis upon software issues; the

discussion is followed by the presentation, in chapter 3, of a kernel-based software architecture based on the abstraction of *objects* and *capabilities*, which we find to be ideally suited to the requirements developed in chapter 2. This material sets the stage for the description of a hardware architecture suitable in meeting the requirements of the abstract system developed in chapter 3; this we present first in Chapter 6, and then again in chapter 8.

The issue of protection and security in shared storage systems is sufficiently important that we have addressed the topic exclusively in chapter 4. Here we extend extant protection and security mechanisms to meet the unique requirements of our abstract model. Chapter 5 explores issues relating to information availability in distributed processing systems; it is both an experimental and analytical study of the role of information redundancy in meeting the availability goals of shared storage systems. The results of this study sets the stage for a hardware implementation of our abstract model from chapter 2.

In chapter 6 we develop a class of fault-tolerant architectures that are eminently suitable for implementation of centralized storage systems. Much of the resulting hardware discussions in the rest of the thesis is based on these architectural concepts. One might say that this chapter represents the *beef* of this thesis, for it is here that fault-tolerance is applied, in a way which is both cost effective and functional; cost-effective because it makes use of a natural form of redundancy, and functional because it provides the necessary functionality of a centralized storage system in a very stylized way.

Chapter 7 extends some of the abstractions developed in earlier chapters by discussing low-level mechanisms in terms of the fault-tolerant architecture developed in chapter 6. The algorithms developed impose certain constraints on the implementation of the architecture, a task which we undertake in chapter 8. Chapter 9 discusses various issues relating to system performance, largely based on our implementation approach. Chapter 10 presents a summary, with conclusions we have been able to draw from our research, and offers many suggestions for further research.

CHAPTER 2

ARCHITECTURAL DESIGN ISSUES

This chapter investigates and discusses various issues fundamental to the design of a shared centralized storage facility. Key among these is the need for fault-tolerance since it is required to provide adequate reliability and availability in a resource shared by many people. As work progressed it became clear that, given highly reliable hardware, many new services could be made available, simplifying operating system functions in host processors, and providing orderly mechanisms for security and for sharing information between hosts in the network. This is, of course, an intelligent storage system which applies fault-tolerant design principles in the design of the storage system itself. This added digital logic is expected to be relatively inexpensive in comparison to the shared secondary storage devices, and allows us to create unique services that can only be made available in a centralized facility of this type.

This chapter discusses the services which should be performed in the storage system. It is used to derive a set of requirements and a basic implementation approach used in the remainder of this thesis.

2.1 Introduction

As described in Chapter 1, the goal of a shared centralized storage system is to provide an environment in which human, software, and hardware

resources can intercommunicate and be shared in a coherent way. In meeting this goal, there are many requirements imposed upon the logical architecture of such a system. Experiences with the design of Backend Storage Networks (BSN's) [Wats80] and File Server (FS) systems [Svob82] point out the need for:

- A logical separation of policies and mechanisms so that users can build new services out of existing ones, or define their own storage management policies.
- Application support, so that users may make use of facilities that they do not inherently possess.
- Host software minimization, so that the storage management component of host operating systems can be effectively removed to the storage system, resulting in less costly and more reliable host operating systems.
- Multiple-host support, so that several users may access the system simultaneously, each observing an environment which is unique to a particular host.
- Reliability and availability, so that the system has a high probability of performing a particular operation to successful completion without interruption, and that it has a high probability of being *able* to perform a particular operation at some arbitrary time in the future.
- Extensibility in both hardware and software, so that the capabilities of the system can be increased arbitrarily with little or no perturbation upon its existing operations.
- Security, in order to prevent the unauthorized disclosure of information

among users of the system.

- Resiliency, so that the system has a high probability of performing an operation which results in a consistent state irrespective of hardware or software failures.
- Efficiency, so that the system can perform operations in a manner which makes it appear to be local to the host making the request.

In the following sections we will develop these requirements more fully, discussing as we go along various issues that are significant in their application to autonomous shared centralized storage systems.

2.2 Separation of Policies and Mechanisms

The first requirement we shall discuss is the need for a dichotomy between what is considered a *policy*, which is subject to arbitrary interpretation by a user, and a *mechanism* which is an inflexible entity provided by the system. Experience with the design of operating systems show this to be a powerful tool in allowing a relatively simple system to meet a diverse set of user needs and/or requirements [Wulf81, Ritc74]. Such a concept is ideally suited for a shared storage system where each user might (and usually does) have different policies regarding the way information is stored and accessed. Instead of trying to satisfy all policies *a priori*, a mechanism which allows each user to define his/her policy is eminently more suitable. Of course, the provision of such a mechanism should in no way constrain the user as to what kind of policies might be implemented.

The principle of separation of policies and mechanisms applies to the centralized storage system in the following way: Since the system is designed to support multiple users with diverse requirements, what is needed are mechanisms with which each user can define those policies that apply to the way in which one's information is to be accessed and manipulated. The mechanisms need to be quite general so that an arbitrary number of policies can be in effect simultaneously. One such mechanism which has proven to be quite useful in operating systems and in programming languages is the notion of *abstract data types* [Lisk74]. An abstract data type mechanism permits a user to define a new operation, and to specify the rules that govern the use of the operation. By providing an abstract type mechanism, users can define new operations that implement their own policy decisions. This is an important concept which is used to satisfy several of the other requirements, and is developed more fully (from an implementation point of view) in the next chapter.

2.3 Application Support

A centralized storage system not only permits the sharing of hardware resources, but application programs as well. By having a centralized storage system which permits the management of information in a coherent way, it is possible to provide services for users that would either be very difficult or impossible to provide otherwise. This is particularly true when there is a mix of host types in the network. Experiences with distributed operating systems supporting data transparency show that added services can be provided by judicious use of system resources [Pope81]. For example, a process may be migrated from site to site without concern for the data which the process mani-

pulates, since this data is globally accessible from any site.¹ This allows the establishment of special *Process Servers* on the network to which host processes can be automatically migrated and serviced. This would not be possible if the location of a process is site-dependent. As another example, text files can be shared between dissimilar host types by providing automatic mapping structures in the storage system to convert between various host formats. Therefore, a user need not be concerned with the origin of a text file, as long as it can be accessed.

In a similar way, application programs can be shared by providing automatic mapping functions which are transparent to the hosts. For example, several users might share a common database without any particular knowledge of the physical location of the database, or even how the information in the database is actually represented. It is also possible for inhomogeneous hosts to share executable program modules transparently. For example, there might exist a host on the network with a specialized language translator; other hosts may make use of the translator simply by referencing it through the storage system and providing the necessary input data. The storage system could take care of all the work necessary to perform the translation and pass the appropriate results back to the user. It is not particularly difficult to provide the necessary functionality in the storage system to meet these needs. Conceptually, all that is required is the mechanism to generate the necessary mappings when they are needed -- this can be easily accomplished by a data abstraction mechanism.

¹Process migration implies that the process migrates among sites that are capable of executing the process.

Besides the direct sharing of application programs, a centralized autonomous storage system allows hosts to augment their capabilities by making direct use of storage system facilities. For example, a host might implement a more capable file management system in the storage system than would normally be possible given the limited resources of the host.

2.4 Host Software Minimization

One of the important objectives of centralizing resources is to minimize the impact of resource management on hosts. This is true not only for the hardware, but the software as well. If hosts do not need to know the details of storage management, then a significant portion of the host's operating system mechanisms dealing with storage management can be removed. For example, since a host no longer needs to know the details of mapping between logical data structures and physical storage devices, device drivers can be orders of magnitude simpler. One goal of this centralized storage system is to remove the burden of storage management from the hosts, thereby making their operating systems simpler.

One approach to the solution of this problem is to provide a mechanism in the storage system with which each host can interface in such a way that host storage management primitives can be mapped into storage system primitives in a user-transparent way. For example, if a host has a primitive called *read_file* which reads the contents of a file in a host-specific format, this could be passed directly to the storage system where it is mapped into equivalent storage system primitives. Normally, the host would define these mappings to ensure that the correct operations are being performed. Again, this can be

satisfied with a data abstraction mechanism.

2.5 Multiple-Host Support

In order to provide as much generality as possible to meet the diverse requirements of various host systems, it is desirable to design the storage system in such a way that it can support several host types. This can range from a mix of simple diskless workstations to large mainframe computers which simply use the storage system to augment their own mass storage systems.

In view of this requirement, at issue here are the ways in which this support should be provided. The main problem is: how does one provide an environment in which multiple autonomous hosts may share a centralized storage system in a coherent way using a minimum of interface mechanisms? This problem stems from the autonomous nature of the hosts; each host type requires a unique interface with the centralized storage system. If there are a large number of such types, and if each type requires unique protocols to communicate with the storage system, the diversity of the resulting mechanisms can lead to designs which are not easy to modify or extend. Ideally, what is needed is a single mechanism upon which can be built the interface requirements of the workstations.

One approach to the solution of the problem is to use a data abstraction mechanism which permits each host to define a virtual machine in the storage system which maps host commands into storage system primitives. In principle, a host would define an abstract type, which we call a *subsystem*, using primitives provided by the storage system. In communicating with the storage system, the host communicates with its defined subsystem through a suitable

subsystem interface. The advantage of this approach is that only a single mechanism (data abstraction) provided by the storage system is required. Using this mechanism, several virtual machines can coexist simultaneously, each defining *exactly* the interface that each host type requires (note that it is not necessary to have more than one virtual machine for each host type in the network).

2.6 Reliability and Availability

Reliability and availability are quantitative terms used to specify, respectively, the probability of successful completion of an operation given that it was successfully started, and the probability that it will be possible to start some operation at some arbitrary time in the future. It is always desirable to have both high availability and high reliability in any storage system, whether it be centralized or distributed. There is, in fact, no reason to believe that distributed systems are any more reliable or available than centralized ones, or vice versa. Although distributed systems evolved primarily because of the need to share resources, it soon became evident that this was also a natural approach to higher reliability because of the inherent redundancy of distributed resources. The important point is not distribution, but redundancy.

While it is possible to achieve high reliability and high availability through fault-avoidance, it has been consistently demonstrated that fault-tolerance through redundancy is both a viable and cost-effective methodology of doing so [Avi78, Renn80]. The reliability and availability, then, of centralized or distributed storage systems are specified in exactly the same way; it is only the application methodology which might differ.

The application methodology in an autonomous centralized storage system of the type we are proposing is not necessarily easier than in a distributed storage system, but it is certainly less constrained. In a distributed storage system, one is constrained by the characteristics of extant storage resources, and by the network configuration itself. This is not the case with a centralized facility where one is more-or-less free to choose an architecture and the devices which are used for storage. In chapter 6 we avail ourselves of this right by proposing a fault-tolerant architecture which we believe to be ideally suited to the requirements of network storage systems. This architecture espouses the principle of redundancy while being arbitrarily extensible at the same time.

2.7 Extensibility

With respect to shared storage, extensibility implies the ability to increase the level of service that the storage system provides for the users, *in a transparent way*. The issue of transparency is an important one. It should be possible to change the physical configuration of the system (for example, adding disks and processors, or removing hardware for repair) without disturbing ongoing services. As another example, we would like the user to be able to create new services (or perhaps new policies for existing services) in a more-or-less arbitrary manner. Therefore, there is a need for both hardware and software extensibility.

2.7.1 Hardware Extensibility

Hardware extensibility is achievable through architectures that permit online hardware modifications. For example, hardware can be designed in such a way that replaceable or new components are electrically isolated from existing

components, achievable through the use of transformer coupling or optical isolation. This requirement is particularly important in view of the high-availability requirement of the system; if a component fails, it should be replaceable without resulting in system downtime.

2.7.2 Software Extensibility

One way to achieve software extensibility is to provide an environment in which users can define their own abstractions. This is commonly referred to as *abstract data typing*, or simply *typing*. A user creates an abstract data type (or simply, a *type*) by defining an *operator* and the rules governing the use of that operator -- the *operations* of the type. By allowing users to create (or delete) abstract data types at will, software extensibility is achieved. What is therefore needed are mechanisms that will permit this dynamic creation and destruction of abstract data types.

Data abstraction alone is not sufficient to provide true software extensibility. The software must in fact be designed to be easily modifiable and upgradable. It is desirable for example to be able to install software updates online while the system is still in operation. Furthermore, if the software is distributed throughout the system, with each component communicating with others over communication links using a protocol, it is necessary to design the protocols so that new services can be added or deleted without requiring protocol revision. One approach to meeting these needs is *modularity*. By designing the software as a set of modules with well-defined interfaces, new modules can be added or the internal details of a module changed without significant perturbation on the overall function of the system's software.

2.8 Security

In an environment where multiple users must share a common resource, the prevention of unauthorized access of information is vitally important. There has recently been much furor over the security of information in online systems such as banking and military systems. There is no reason to believe that systems such as that proposed in this thesis will not find applications in these areas. Therefore, security mechanisms should be an integral part of the system's design.

There are two aspects to security: *information security*, and *physical security*. Information security pertains to the unauthorized disclosure of information in a computer system, while physical security concerns the susceptibility of the system to physical attacks. Although the notion of centralization tends to support physical security, since the storage devices can be kept under common surveillance, we do not consider aspects of physical security in this thesis. On the other hand, information security must be given serious consideration because of the inherent properties of contemporary computing systems. Because of the need for high performance in information management, the density of information on storage devices is continually increasing. As this density increases, the possibility of data interaction between independent users increases, particularly when failures manifest themselves. However, interactions due to failures are not the only threats to security. As pointed out in [Denn82], information security can be subverted by eavesdropping, tampering, browsing, searching, inference, wanton or accidental destruction, collusion and masquerading.

In an autonomous centralized storage system, what is ideally needed is a *single* security mechanism whereby arbitrary security policies can be implemented. One of the biggest problems with designing secure systems is the diversity of security mechanisms that are used; each such mechanism requires independent verification in addition to the complexities of interfacing them. It should be possible for each autonomous host to use this mechanism to *extend* those security mechanisms already extant in the host, or alternatively to use the mechanism of the storage system to augment that already provided. In chapter 4 we show how a single mechanism based on access controls can be used to implement arbitrary security policies.

2.9 Resiliency

Resiliency is concerned with the *consistency* of information in the storage system. It introduces the notion of *atomicity* which has the property that either an action is performed to completion, or its effects are not observable in the system [Lamp81]. There are two conditions under which information consistency is desirable: 1) whenever software failures occur (such as operating system crashes in either the hosts or the storage system), and 2) whenever hardware failures occur. What is therefore needed is a mechanism which will, with high probability, provide for consistency under the specified conditions. One useful paradigm for providing consistency under these conditions is the notion of a *transaction*.

2.9.1 The Transaction Paradigm

The classical model of a transaction, as described by [Gray78], defines a mechanism for constructing reliable database systems. In this model, transactions are defined as arbitrary collections of operations, or *actions* delimited by two markers: *Begin _ Transaction* and *End _ Transaction*, and have the following special properties:

- Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity*.
- If a transaction completes successfully, the system state always satisfies its invariant predicate. This property is usually called *internal consistency*.
- If a transaction completes successfully, the system state corresponds to external perceptions of it. This property is usually called *external consistency*.
- If a transaction completes successfully, the system state reflects, as closely as possible, the correct response to external stimuli. This property is usually called *congruity*.
- If a transaction completes successfully, the results of its actions will never subsequently be lost. This property is usually called *permanence*.
- If several transactions execute concurrently, they affect the database as if they were executed serially in some order. This property is usually called *serializability*.

- An incomplete transaction cannot reveal results to other transactions, in order to prevent *cascading aborts* if the incomplete transaction must subsequently be undone.

If a transaction is performed successfully, it is said to have *committed*; if it was started but failed, then it is said to have *aborted*.

Transactions are extremely useful in the design of any system since they reduce the burden on application programmers by simplifying the treatment of failures and concurrency. The property of failure atomicity guarantees that when a transaction is interrupted by a failure (operating system crash or hardware failure), its partial results are undone. Serializability insures that other concurrently executing transactions cannot observe any inconsistencies brought about through the temporary violation of consistency constraints by programmers. Prevention of cascading aborts limits the amount of effort required to recover from a failure. If it were possible to design a generalized transaction mechanism in a centralized storage system, issues relating to concurrency control, sharing, and failure recovery of information could be handled by a single mechanism [Spec83]. has studied some of the requirements for the applications of transactions in general-purpose distributed systems, and a large portion of the significant results are applicable in this context. The reader is referred to that work for a more in-depth discussion of the transaction methodology in distributed systems. Svobodova [Svob82] discusses the importance and design issues of transactions in file server systems, but not in the context of a generalized mechanism supporting abstract types; it is however, a useful report since it discusses various approaches to the application of transactions in contemporary file server systems (some of which have been actually con-

structed).

2.10 Efficiency

Centralizing storage resources should not unduly impact the performance of the network. In fact, it would be desirable to design the system in such a way that accesses appear to be local to each host so that the system is essentially transparent (from the user's viewpoint). There are several potential bottlenecks to performance in these kinds of systems. First, the need to transport information across a shared network could be a bottleneck, depending upon the bandwidth of the network, and the characteristic and volume of the offered traffic. Second, the performance of the storage system itself; especially so in view of the need to maintain interfaces for each host which are largely implemented in software. In chapter 8 we propose hardware and software architectures which are meant to provide a system which will meet the performance requirements of a wide range of host systems.

2.11 Summary

In this chapter we have investigated a number of issues important in the design of autonomous shared storage systems. In some cases we proposed approaches which provide viable solutions to an issue. In the remainder of this thesis, we explore these solutions in more detail. In the next chapter we develop a system model and propose the implementation of a storage management system which meets all of the software requirements discussed in this chapter.

CHAPTER 3

SOFTWARE SYSTEM MODEL

In this chapter, the goal is to describe a software architecture for a Storage Management System (SMS) that attempts to meet the requirements discussed in the previous chapter. In so doing, we first postulate a strawman hardware architecture which serves as a framework for the development of the software architecture. A generalized software model is discussed, after which we present a software design overview, and finally an implementation of the architecture.

3.1 Strawman Storage System

In general, it is very difficult to structure a software architecture without some knowledge of the hardware on which it will eventually run. This is true for such areas as memory management, process management, and I/O. However, it is still possible to base the design on abstractions which can be mapped onto whatever hardware is chosen in the final implementation. It is normally the case, however, that hardware peculiarities will severely impact the software architecture. For example, the design of the paging mechanism in Hydra [Wulf81] and the I/O mechanism of the UCLA Data Secure Unix [Pope78] were direct consequences of the underlying hardware. PSOS [Neum77], on the other hand, appears to be an operating system which was designed with very few hardware dependencies; one suspects, however, that its

implementation was not without compromise, for reasons of efficiency.

The approach we will take, is to postulate a strawman hardware architecture on which to specify the software architecture. This hardware architecture hides many of the detailed fault-tolerance techniques and multi-computer implementations which will be developed later; but it is idealized in such a way that the software architecture can be transferred to the detailed design in later chapters. This architecture is shown in Fig. 3.1.

The strawman storage system architecture as shown in Fig. 3.1 consists of a set of hosts connected to a local network, and the storage system connected to the same network. All connections to the network are assumed to be via a suitable network interface. We assume the network to be of the "ethernet" variety [Metc80], with whatever transmission speed is needed to do the job. We assume that such a network and its interfaces provide perfectly reliable communication at the link level, assuming that low-level fault-tolerance to be added later will not have a major structural or functional impact on the software architecture. Furthermore, we assume that the storage system proper consists of a *server* computer to which is attached a number of high-capacity disk storage units. This computer is assumed to have as high a speed as is needed and is fault-free. The disk units communicate with the server CPU via a very high rate fault-free communication subsystem. Each disk unit is managed by a single fault-free processor which resides logically between the communication subsystem and the unit. These processors will be used primarily to manage the information stored at each disk unit. The one thing that we do not assume to be fault-free is the disks. They may fail, but we impose the requirement that whenever such failures occur they must be detected by the

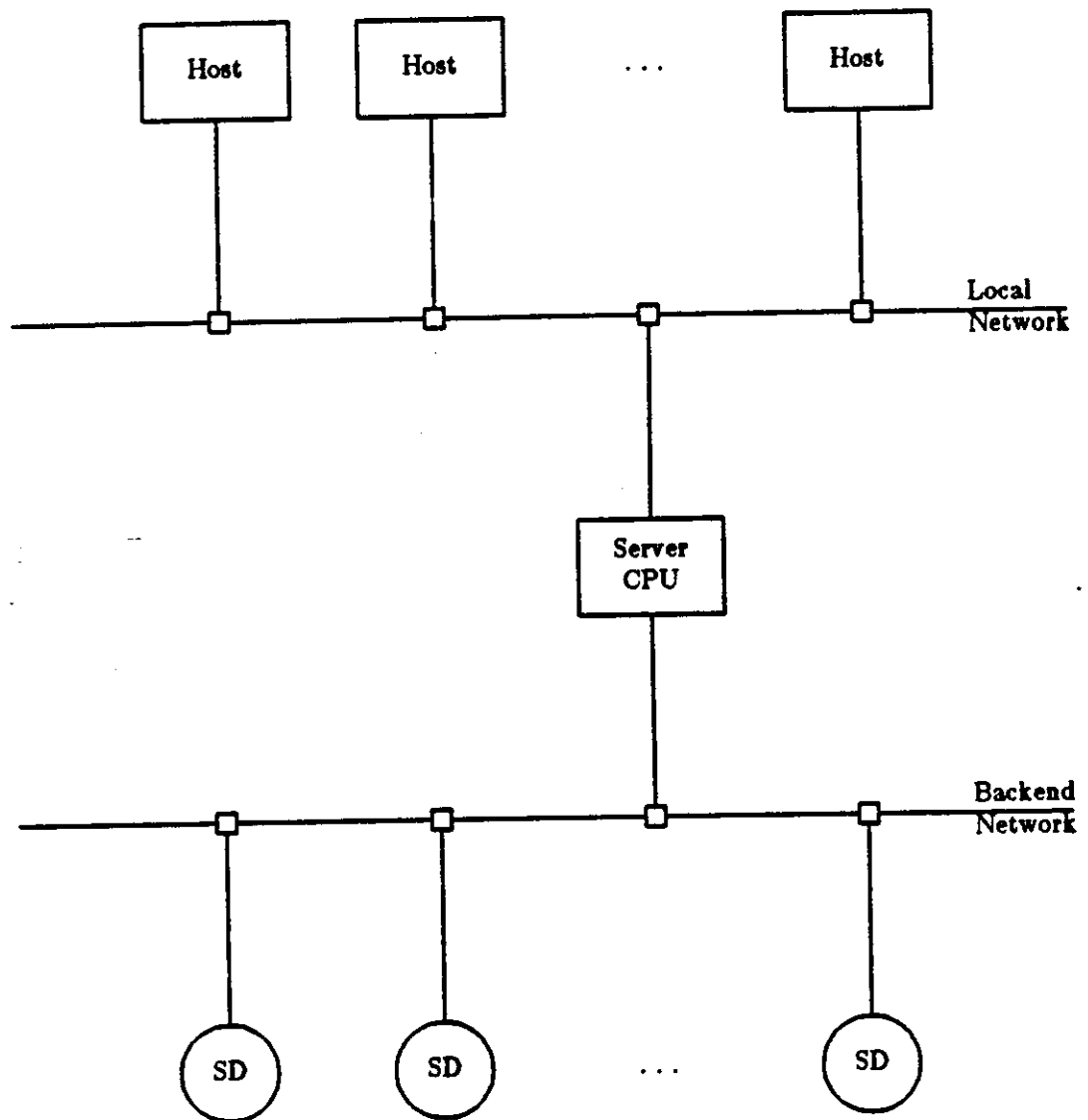


Figure 3.1: Architecture of SMS

processors which will prevent damaged data from propagating. The need to replicate information on independent disks and the ability to recover is a fundamental requirement of the SMS which must be dealt with in the strawman design to be transferable to the final architecture.

3.2 A Generalized Software Model

In the previous chapter several requirements of a software model for a centralized storage system were presented. In this section we discuss a software model which allows all these requirements to be met. In this model, all protectable resources -- virtual or physical -- are represented as *objects* and accessed via *capabilities*. This model follows very closely the object model as typically applied to resource-based operating systems [Jone79]. In the model, which we will henceforth refer to as the "object model", each object has a unique name, a representation, and a type. A capability, which serves as both an access and a naming mechanism, has a unique name (the same as the unique name of the object to which it points), and a set of access rights.

The salient feature of the object model is the implementation of security via access control. The representation of an object cannot be accessed unless a capability is provided for the object which contains the appropriate rights for the type of access being requested. Any implementation of the model must enforce this mechanism. In addition to security, the model can easily be extended to support abstract data types which is essential for software extensibility. This is accomplished via the technique of *type extension* with which users are able to create new object types by combining existing ones, or by creating entirely new ones. The interested reader is referred to Jones [Jone73] for an excellent analysis of protection using the object model, and to Wulf *et al* [Wulf81] for an implementation of abstract types in the object model. In FTSS, all information stored by users are encapsulated as objects under the model and their security is enforced via the capability mechanism. In the rest of this chapter we describe the implementation of the object model which

forms the basis of the SMS.

3.3 Software Design Overview

The following paragraphs describe some tradeoffs and the basic approach recommended for the design of software for the idealized storage system.

3.3.1 Security-Driven Models

One of the primary motivating forces that will drive the software architecture of the SMS is the need for security. Within this context, there are two further requirements: First, the need for verification that the resulting architecture does meet security goals; and second, the system provides enough functionality to meet the proscribed requirements, and that it does so efficiently.

There are two approaches to the design of architectures based on the need for verifiable security. The first is to base the design on a small nucleus of code which implements all security-related actions in the system. This nucleus is a virtual machine which resides directly on top of base hardware, providing extended instructions which form the basis of all security mechanisms in the system. This nucleus is commonly referred to as a *security kernel*, and gives rise to security kernel-based architectures [Pope79, Sche80]. The second approach is to specify the entire software architecture in such a manner that it is possible to prove the security of the system using automated tools. It is common in such approaches to base the resulting design on a hierarchy of modules in which intermodule dependencies is strictly top-down. This approach was used in the design of the PSOS operating system [Neum77].

Given the current state-of-the-art in specification techniques and verification tools, especially where parallel, asynchronous processes abound, the first approach has significant advantages over the second. If it is indeed possible to structure one's architecture on security-kernel principles, cogent arguments have been made in support of this approach, especially where extensibility and separation of policies and mechanisms are concerned [Pope79]. Furthermore, since only a relatively small body of code need be proven secure, the design is naturally modularized and therefore simpler. For these reasons, and others,¹ we choose to base our software architecture on security kernel principles. In the next sections we discuss various architectural decisions based upon the system requirements.

3.3.2 A Kernel-Based Architecture

In designing any kernel-based architecture, one must first decide on what should be included in the kernel. As far as security kernels are concerned, the general principle is to include all security-related *mechanisms*², and to exclude all others. By including only security-related functions, the size of the kernel can be kept small which increases its verifiability. There is sufficient experience in the design of operating system kernels that pave the way. For example, in the design of the Hydra kernel, all generic operations on objects and capabilities are included in the kernel. In UCLA Data Secure Unix, only those operations that are directly related to security are implemented in the

¹Especially in mapping the architecture onto fault-tolerant hardware.

²Upon which arbitrary policies can later be built.

kernel.³ The lesson provided by Hydra is that it is much better to design a flexible system, which can be mapped onto efficient hardware (or made efficient through different codification), than to design a system which is initially efficient but is very difficult to change. From these arguments, it is clear that there are tradeoffs between verifiable security, functionality, and efficiency. Our approach is a compromise between the first two which, we hope, provides enough flexibility to meet our system requirements.

The SMS is a kernel-based architecture. The objective of this system, to reiterate, is to serve as an intelligent centralized interface between a number of distributed hosts and a (possibly) distributed set of storage resources. The SMS is internally structured as shown in Fig. 3.2. It consists of a kernel which is encapsulated logically by a set of *subsystems*. These subsystems perform system operations that do not pose a threat to system security, but are nevertheless an important part of the system's functionality. The subsystem "level" is encapsulated by one or more *user-level* subsystems that form an interface with users; these users represent the outermost level of the system. There are several advantages to structuring the SMS in this hierarchical fashion. First, the hierarchy provides natural modularization which makes the design task less onerous. Second, it allows the functions and even the properties in each level to be modified without unduly affecting other levels, providing that there are well-defined interfaces between them. Third, it affords the establishment of natural "protection boundaries" which serve as firewalls for error propagation. Finally, it helps to support the notion of abstract types by allowing users to

¹There are, however, security-related functions that are relegated to "trusted" processes that execute in a verified "sub-kernel" outside of the main kernel. This dichotomy was purely for verification modularity.

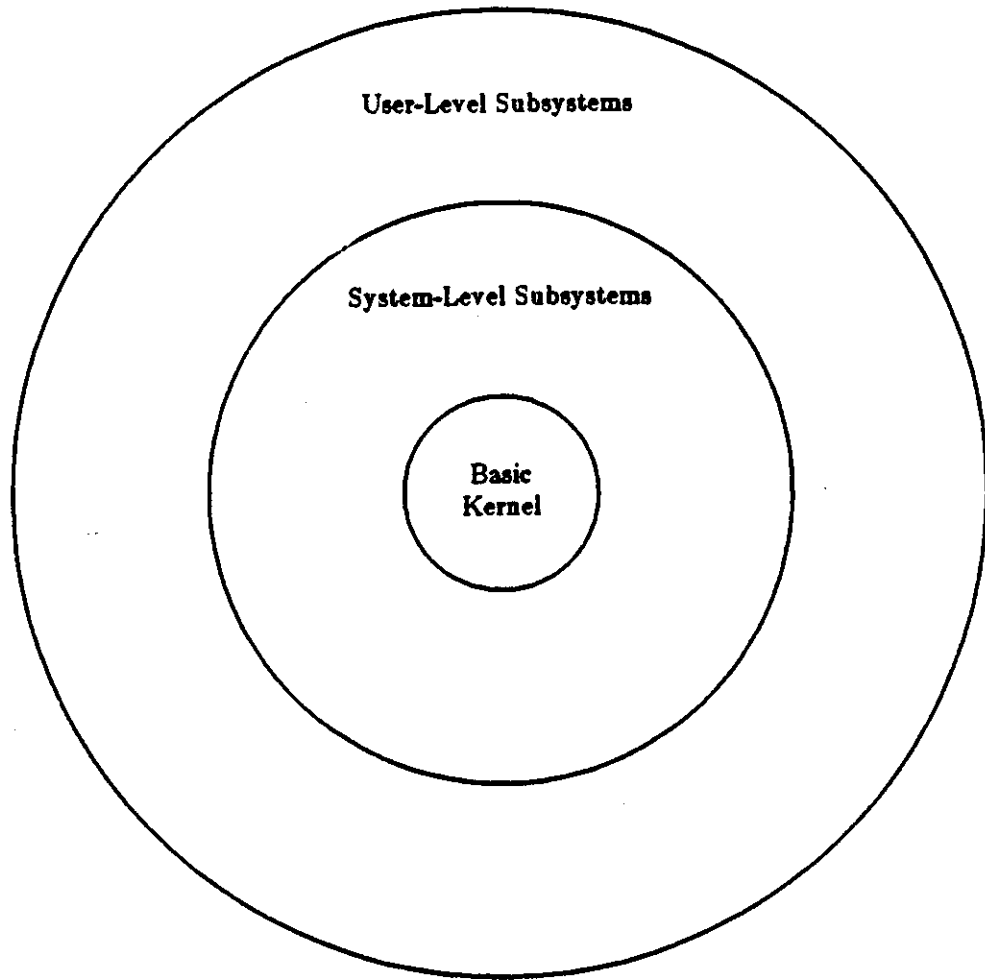


Figure 3.2: Multi-Level Kernel Architecture

define the environment most suitable for their own needs. In the next sections we will describe the primary attributes of each level in the figure.

3.3.2.1 The Kernel

The kernel implements the object model. The reason is simple: in the object model, security is enforced via objects and capabilities -- this is the only mechanism required; all other higher-level security policies can be built on top of it. In order to properly create the abstraction of the model, one must

understand how objects and capabilities are represented, how they are created and destroyed, how protection is enforced in the execution domain, and how abstract types are created and destroyed. The kernel is organized into a set of modules. There is a module that deals with the creation, manipulation, and destruction of capabilities, a module that deals with the creation, manipulation, and destruction of objects, and a module that deals with the enforcement of protection in the execution domain (of a program). However, for reasons of efficiency, the kernel contains other modules that provide support for some of the system functions defined below in the subsystem level. The rationale for this modular organization is twofold. First, it simplifies the design task since one can hide implementation details in the body of the module without fear of interaction with other software in the kernel. Second, by making only those functions that are needed outside the module visible, there is a much higher probability that the system will perform correctly; this approach also assists in verification since it significantly reduces the number of states in the system.

At this level, the design of the kernel parallels, in principle, the design of the Hydra kernel [Wulf81]. We support the notion of small protection domains, as in Hydra, resulting in an object grain at the level of "procedures".¹ While this might seem unnecessary in SMS, it allows great flexibility and provides better security in the system. In general, the finer the object grain, the better control one has over the unauthorized access of information. Therefore, the kernel supports a procedure invocation mechanism which provides a protectable domain in which the procedure executes. The details of this invocation mechanism will not be presented at this time.

¹The term "procedure" here is in direct analogy with the notion of a procedure in high-level languages such as Pascal.

3.3.2.2 The Subsystem Level

The subsystem level implements a number of abstract types that either provides support functions for the kernel or provides additional services necessary to meet the system's requirements. They are separated from the kernel because, to a large extent, their operation is not a threat to system security and this helps to keep the kernel small.¹ These system functions are as follows:

- a. *Message System*: A communication facility for cooperating sequential processes supporting the user/server model of communication.
- b. *Scheduler*: A facility for the scheduling and synchronization of user-level processes.
- c. *Transaction Manager*: A facility for the reliable execution of a single process, or multiple sequential processes.
- d. *Object Manager*: A facility for managing information storage objects in the storage system.
- e. *System Monitor*: A facility which provides general system monitoring capabilities for administrative uses.
- f. *Security Manager*: A facility which allows arbitrary security policies to be implemented.

While these subsystems are a minimum set necessary to meet system requirements, others can be easily defined as abstract types and added when necessary. While it is ideal to implement the object model in the kernel and all

¹They are, in fact, implemented on top of the object model.

other non-security related system functions above it, this can be achieved only if the performance of the resulting system is acceptable. More than likely, it will be the case that the kernel must provide some additional support for operations that need to be performed quickly, without the overhead of the call interface between the two levels. This is the reason, for example, why it is common to find device drivers, processes, and so on implemented as a part of the kernel. We find a similar situation in SMS; for example, all the I/O mechanisms are implemented in the kernel because of the need to avoid the kernel call interface - since a large part of FTSS's function deals with I/O, it is logical to optimize it as much as possible.

3.3.2.3 The SMS Interface

This level of the SMS provides a mechanism which users might use to create an interface with their local operating system. Let us consider the case of UNIX¹ as an example. In order to make the storage system transparent to a UNIX user, one would create a virtual machine in the storage system which emulates the behavior of the UNIX file system. That is, a user should be able to store and access files in the storage system exactly as if they were "local" to his machine (not considering performance issues at the present time). One way to do this is to simulate the actions of the UNIX file system within the storage system through a suitably defined abstraction. Clearly, this could also be done by implementing the virtual machine in the host, but that would violate our requirement that the storage management of host computers should be removed, or made considerably simpler. In the case of UNIX, this would mean removing all block I/O management and device drivers, and implementing

¹UNIX is a trademark of Bell Laboratories.

them (or their equivalent) in the storage system via kernel mechanisms. An advantage of this strategy is that all hosts that make use of this UNIX interface could share the same subsystem, significantly reducing the software overhead and reliability of the storage system. The objective of the SMS interface, therefore, is to provide an environment in which this is possible.

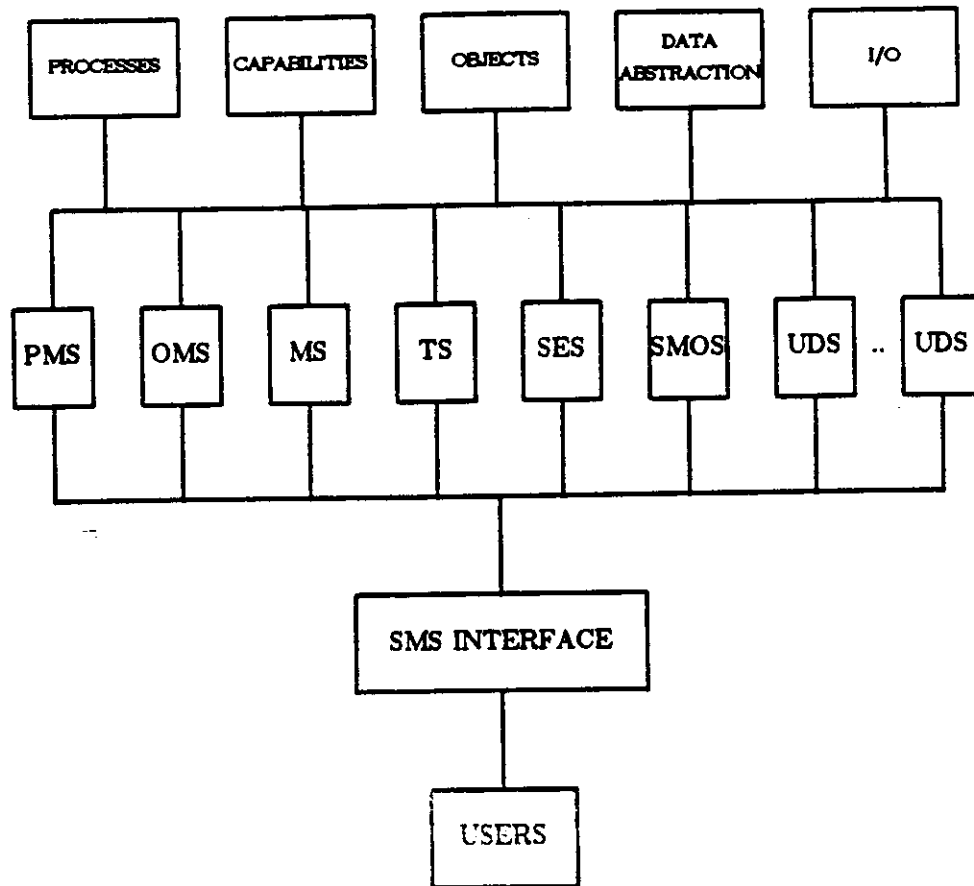
The SMS interface provides two basic services. First, it provides a subsystem interface which makes it possible for users to create new abstract types to tailor their requirements. Second, it provides an operating system interface for those users who do not wish to build their own subsystem interfaces.

3.4 Implementation of SMS

In this section we present a high-level view of an implementation approach to the SMS based on the logical structure of the previous section. We show the major functional blocks of each of the levels of the SMS and describe the visible operations that are available at that level. These operations form a set of primitives which are sufficient to implement all the intended functions of the storage system. At this point the interested reader is referred to Fig. 3.3 which shows the relative location of each functional block in the SMS.

3.4.1 Kernel Implementation

The kernel implements the abstractions of the object model plus additional mechanisms needed to support higher-level abstractions. These are implemented "below" the protection level of the system. For example, the abstractions of capabilities, processes, devices, and other kernel "types" are created "out of thin air" by kernel code.



UDS: User-Defined Subsystem

Figure 3.3: Functional Block Diagram of the SMS

3.4.1.1 An Overview

From an implementation point of view, the kernel is an executable piece of code that resides permanently in memory (in-core), providing the basic services of the system. These include process, I/O, interrupt, and memory management, as well as protection via the object/capability paradigm. In general, it is difficult to completely dissociate each service in the implementation without replicating large sections of the code. However, it is still possible to discuss each independently since each has a unique function.

In the following discussion, we will assume that the kernel is to be implemented on conventional uniprocessor hardware with the following characteristics:

1. demand paged virtual memory,
2. hardware protected address spaces; as a minimum, system versus user space, and
3. a rich repertoire of instructions.

Characteristic of such processors are the VAX-11 series, the Motorola MC68020, and the National Semiconductor 32032, all of which are widely available in the public domain.

The kernel is implemented in its entirety in system space. It is assumed that the processor has alternate register sets, one for the system and the other for users such that the execution of a program in one space will not affect another program's execution in the other. The switch from one space to another is accomplished through a privileged instruction executable only from system space (that is, from the kernel). Since it is not possible for programs running in user space to access the kernel in system space (the hardware will not allow this), the kernel is designed to have a standard set of *entry points*. The SMS kernel has a number of standard entry points of which the following are the most important:

1. The kernel call entry point; a program running in user space may call upon routines provided in the kernel - the actual implementation is hardware dependent; however, it is normally accomplished by a software

interrupt from the user program; the interrupt invokes an interrupt service routine in the kernel which then determines which service the user requires.

2. The memory management entry point; user programs that cause memory faults such as a page or segmentation fault (assuming the processor supports a segmented address space) will trap to the kernel through this entry point.
3. The hardware interrupt entry point; hardware interrupts such as I/O completion or clock interrupts are handled through this entry point.

3.4.1.1.1 Memory Management

Memory management is responsible for the management of two virtual address spaces: the first is the address space associated with physical memory in the processor (where processes execute), and the second is the address space in the storage system where semi-permanently stored objects reside. In SMS, the two virtual spaces are managed independently. There is a kernel memory management component responsible for allocating/deallocating in-core memory, and an external memory management component in the storage system for managing objects stored there. An interface is provided in the kernel which allows communication between the two. When an object is currently being accessed by kernel mechanisms, the object is said to be *active*, its representation (or a part of it) is in the processor, and the capability which points to it contains the real address of an in-core data structure used for locating the various incore components (more detail of this will be discussed in chapter 7). When an object is not being accessed, it is said to be *passive*, its representation

is in the storage system, and the capability which points to it contains the unique identifier for that object. An active object may be passivated at any time (writing its representation to the storage system) via an appropriate kernel mechanism; a passive object is activated simply by referencing it in the kernel. In addition to in-core (memory) space, the kernel also manages space for swapping processes in and out of the processor, as well as space for demand paging.

The determination of how the physical (in-core) address space is divided between system and user is typically established by the system builder at system initialization. The kernel is aware of this division and will map objects to the proper space in conjunction with the processor's memory management hardware. In processors such as the VAX-11, the hardware provides segmentation registers which can be used to partition the real memory space into a number of (possibly overlapping) segments - it is the responsibility of the kernel to establish segment boundaries and to switch the processor from one space to the other (system versus user) as a function of which segments are being used. When an object is created by the kernel (as all objects initially are), incore memory space is allocated for the representation of the object (more details on this is presented in chapter 7). Since object creation is related to an executing process (except from some specialized kernel objects), the object's representation must be stored in either system or user space, depending on the space in which the process is currently running. The kernel keeps track of all objects in a global kernel data structure indexed by the unique identifier of the object.

Although the management of the storage system's address space is independent of the processor's address space, the kernel provides facilities that permit a storage system object to be read into core, or for an in-core object to be written to the storage system. The primary kernel mechanism from a memory management standpoint is a set of in-core buffers for holding object blocks; these buffers are allocated in system space and are maintained as a circular list with a least recently used allocation discipline. When a block of a storage system object is to be read, a buffer is allocated from this buffer pool and the object block is transferred to this buffer - if the buffer was previously modified, it is written back to the storage system before it is reused. Additional details of this mechanism will be presented in chapter 7.

Another important aspect of memory management is concerned with the paging aspects of the system. In systems that do not support paging, the entire executable module (representing a program to be executed) must be core-resident for execution to take place. In a paged system, only those pages that form the working set of the executing program needs to be in core - the rest is paged in as desired. To reduce the overhead of paging, it is necessary to have the pages of the executable module in relatively fast storage. A number of systems provide special paging devices such as head-per-track disks for this purpose. In FTSS, a similar approach is suggested; when an executable module is to be executed, the entire module is transferred to a paging device at the top level of the system before execution begins.

Since processes may be swapped out of the processor to free physical memory resources, the kernel memory management component also manages a swap area. The swap area is again, usually, a fast special device that permits

relatively rapid swapping. Head-per-track disks is a good choice here. The paging device is sometimes partitioned so that it can also be used as the swap device as a cost saving measure. The structure of FTSS (as will be discussed in greater detail in chapter 8) is such that the paging/swapping devices are implemented at the topmost level of the system in close proximity with the processor.

3.4.1.1.2 Process Management

Processes are the active elements in SMS. A process is an instance of the execution of a program. Some processes may exist and execute entirely in kernel space, while others will execute for the most part in user space. In order to take advantage of the protection afforded by the object model (which the kernel implements), processes are built on top of it; that is, a process is represented as an object. When a process is created, the kernel maintains two data structures on it - one which is resident in the kernel as long as the process exists which maintains information needed by the kernel even when the process is swapped out, and the other which is dynamically loaded in the kernel when the process is running. A process which is incore but currently not running is a candidate for being swapped out onto a high-speed swapping device in order to free physical memory space for other processes. A process which is swapped out will be swapped in again when it is scheduled to run.

Processes begin their existence in the kernel. If it is a user process, the kernel instructs the hardware to switch address spaces and updates the processor registers in user space so that the process can run. Note that only a single process runs in the processor at any time. If, during the execution of a user

process, kernel mechanisms are needed, the process is switched back to system space, the operation performed, and then switched back to user space. Since termination of a process is a kernel operation, processes always terminate in system space. All objects that a process accesses are rooted in the process object itself (more details of this will be presented later in this chapter). The protection mechanisms of the object model insures that a process cannot access an object for which it does not possess a capability. The details of object access during execution are presented later in this chapter and again in chapter 7.

3.4.1.1.3 I/O Management

That part of the kernel that deals with I/O is largely concerned with communication between the kernel, the storage system, and users on the local network. The basic mechanism is to buffer data between various I/O devices and to establish lines of communication between the kernel and external devices. For this reason, the I/O component is mainly a collection of I/O drivers and configuration routines. Kernel memory management interacts with I/O through the buffer pool mentioned previously. When I/O is to be performed, a buffer from the pool is assigned (either for reading or writing) and the I/O is allowed to proceed. Although it is not fundamental that the I/O component be implemented at kernel level, it is done so because of the high performance usually needed; for example, a typical disk will supply data at a rate of approximately 2 million bytes per second - in order to minimize processor overhead, the transfer is usually done using Direct Memory Access (DMA).

3.4.1.2 Kernel Modules

In order to simplify the design task, and as an assistance to possible verifiability, we have chosen to discuss the implementation of the kernel in terms of a set of modules. The modules are logically independent, providing communication through well-defined interfaces. In the following discussion, we will not be concerned with the details of implementation of each module. Rather, we shall concentrate on the interface to each module and the interrelationships between them. Each module represents one functional level of abstraction for the kernel.

3.4.1.2.1 Capabilities

The capabilities module represents the abstraction of capabilities in the object model. Each capability is a bit vector consisting of two fields. A 64-bit field representing the unique identifier (UID) of the capability,¹ and a variable-length bit vector representing the set of access rights for the capability. The set of visible operations on capabilities are:

```
C_restrict(capability,mask):capability | signal
C_copy(capability):capability | signal
C_destroy(capability):Boolean | signal
C_compare(capability, capability):bit_vector | signal
```

The first operation `C_restrict` allows some or all of the rights in a capability to be restricted (i.e., removed), and either returns a capability or a signal representing some error condition. The second operation allows a capability to be copied, and the third destroys the representation of a capability.

¹The choice of a 64-bit field is somewhat arbitrary. However, it will permit the generation of UID's for over 500,000 years, assuming that a new one is generated every 1 microsecond.

`C_compare` compares the representation of two capabilities and returns a bit vector of the result. It should be noted that there is no explicit means of creating a capability - capabilities are created in concordance with objects; that is, a new capability is created only when a new object is created. While other operations may be possible (or even desirable), these are sufficient to handle most of the anticipated operations of FTSS. No other module may manipulate the representation of a capability.

3.4.1.2.2 Objects

The object module represents the abstraction of objects in the model. It deals with their creation, manipulation, and destruction. Each object is structured in the following way: it consists of a *capability list* and a *data part*. Each part is logically distinct.¹ The capability list (CL) is logically structured as a variable number of *slots*, each of which holds exactly one capability (if a slot is "empty", it holds the special *NULL* capability). The data part (DP), on the other hand, is logically represented as a contiguous block of storage, quantized in some convenient way (in this case, we assume that the quantization factor is the *byte*). Either the CL or the DP may be null. The following are the visible operations on objects permissible by the kernel:

```

O_create *(capability_index):capability | signal
O_create(type_capability,capability_index):capability | signal
O_copy(capability,capability_index):capability | signal
O_get_c(capability,capability_index):capability | signal
O_put_c(capability,capability_index):boolean | signal
O_get_d(capability,buffer,offset,count):count | signal
O_put_d(capability,buffer,offset,count):count | signal

```

¹If we do not assume this, then the underlying hardware must have a means of distinguishing capabilities from data. A tagged architecture has been suggested for this purpose [Fabr74]. However, since such architectures are not readily available off-the-shelf, we do not make this assumption.

```

O_info(capability): structure | signal
O_compare_c(capability, capability_index,
             capability_index):mask | signal
O_destroy(capability):boolean | signal
O_put_object(capability):boolean | signal

```

Briefly, the semantics of the operations are: `O_create_*` is a generic create call which is used to create predefined kernel types; "*" is replaced by the name of the type and `object_type` specifies the type of object to be created - a capability with the right of creation is required. `O_create` creates a new object of the type specified by the `type_capability` - this call is used to create extended types where `type_capability` points to an object which defines the structure of the type. `O_copy` makes a copy of an existing object. `O_get_c` and `O_put_c` get and put capabilities into or from CL, respectively. `O_get_d` and `O_put_d` get or put data into or from the DP, respectively, "buffer" is used to hold the data, while "offset" and "count" specify the location relative to the start of the DP and the number of bytes to transfer, respectively. A count of the number of bytes actually transferred is returned. `O_info` returns a structure which contains useful information about the current state of the object specified. `O_destroy` destroys the representation of an object (both CL and DL), including any capabilities in CL. `O_put_object` writes the incore representation of the object to the storage system. No other module in the system is permitted to manipulate the representation of an object. Also, notice that each operation requires that a capability be presented; the operation is allowed only if the rights in the capability are appropriate.

3.4.1.2.3 Data Abstraction

The data abstraction module provides support for the implementation of *small protection domains* in the SMS. However, on a larger scale, it is the mechanism by which types (abstract or predefined) are executed by SMS. Protection in SMS is bound to the notion of an *execution domain*. This domain might be a single procedure, or a collection of procedures. Protection enforcement mandates that the only objects which an executing domain might access are those that are either explicitly defined in the domain, or those passed to the domain as parameters. This is in support of the *principle of least privilege* which states that a procedure (executing in a domain) should possess no more privilege than it needs to carry out its intended function. The following are the visible operations of this module.

```
D_call(capability_indexes, capability, parameter_list):  
    capability | signal  
D_call_type(capability_indexes, type_capability, capability,  
    parameter_list): capability | signal  
D_return(capability_list, parameter_list) signal
```

D_call takes a capability for an instruction object, a list of parameters to be passed to the called environment; if capabilities are to be returned, they are stored in the indexes specified by capability_indexes. D_call_type takes a capability for a type-definition object, a capability for an instruction object related to the type, and a parameter list to be passed to the called environment; capability_indexes serve the same function as in D_call. D_return returns control from an executing function to the calling environment, and may optionally return values as specified in the parameter list. The reader should note that direct kernel calls do not create new environments.

3.4.1.2.4 I/O

The I/O requirements of FTSS can be quite intense, depending upon the level of activity in the system. Since it is expected that there will be heavy traffic between the server and the disks, this section of the SMS is implemented at kernel level. The major part of the I/O system supports message traffic between the users and the server, and between the server and the disks. The actual device-dependent characteristics of disks are hidden from the kernel; information is handled independently by each of the intelligent processor on each disk - this simplifies the I/O structure of the kernel considerably. To mesh with the protection mechanisms of the system and to provide a uniform access structure, each I/O device is implemented at kernel level as an object of type *device*. Thus, if a user wishes to access a particular device, only a capability for it is needed. The kernel handles all buffering of I/O so that users do not have to handle this. In an alternative approach, the message system (described subsequently) can be used to allow users to connect with device objects; however this would require that each user perform his own buffering. In view of the need for efficiency, this does not seem like a reasonable approach; instead, users make direct calls to the kernel when I/O is to be performed.

3.4.2 Subsystem Implementations

The subsystem level is made up of a number of subsystems which provide higher levels of abstraction of system resources. They create, out of the base hardware and the kernel operations, those operations that are typically found in contemporary resource-based operating systems. Although SMS is not designed to be a general-purpose computing environment, it is necessary to

include as much generality as possible in order to meet the system requirements discussed in the previous chapter.

3.4.2.1 Process Management Subsystem

The SMS view of process management is concerned with the creation, manipulation, and destruction of *processes*.¹ Processes execute on a virtual machine provided by a suitable combination of hardware and software. The design allows for a relatively large number of these virtual machines to coexist, executing processes in a fully asynchronous manner.

One of the crucial decisions facing the implementation of the process abstraction concerns the distribution of the implementation between kernel and user space. On the one hand, it is desirable to build PMS entirely in user space in order to minimize kernel code; on the other hand, for reasons of efficiency, it is desirable to build it in the kernel so that it does not incur the overhead associated with the kernel call interface. Most implementors compromise by building speed- and/or security-sensitive mechanisms in the kernel, and other mechanisms outside. There is, however, no clear-cut approach to what should or should not be included in the kernel. The view we take in PMS (as we do in much of the other subsystems) is that if the kernel call interface can be optimized (performance-wise), then virtually all of the subsystem can be implemented in user space. One cannot ignore security however. If the process abstraction is built on top of the object model (by treating processes as objects), then all security is handled by the (kernel-implemented) object model. We have actually made provision for this by implementing the data abstraction

¹A *process* in this context is a schedulable piece of software, or, alternatively, a "site of execution".

mechanisms in the kernel.

As explained previously, a process needs a virtual machine (or virtual processor) on which to execute. This virtual processor typically provides an instruction pointer, a set of registers, and an execution stack, among other things. In view of the necessity to execute processes as quickly as possible (and therefore the ability to support a large number of concurrently executing processors), it is desirable to build the virtual machine as close to the real machine as possible. PMS takes this viewpoint. It builds the process' virtual machine in the kernel. However, all decisions regarding the creation, manipulation, and destruction of processes are made outside the kernel. PMS therefore consists of a set of kernel calls and a subsystem for managing processes. Below we discuss the kernel and subsystem functions, and the interface between the two.

The kernel provides the following operations on processes:

```
P_create():capability | signal
P_start():boolean | signal
P_interrupt():boolean | signal
P_kill():boolean | signal
```

P_create creates a process object (out of thin air) representing the virtual machine on which the process will later execute, and returns a capability for it -- this could, in fact, be treated as one of the generic kernel create operations (O_create_*), but we choose to separate it for clarity. P_create also associates with the process object all those instruction objects that are needed during its execution; capabilities for these instruction objects are inserted into the process object by the kernel, and the process is made ready to run. P_start actually starts the process executing at whatever virtual address is assigned by

the processor; it is usually an error to start a process that has not been properly initialized. P_interrupt allows the process to be interrupted in its normal course of execution; in effect, an interrupt call causes the process to "return" to the control of the subsystem. P_kill causes the kernel to terminate execution of the virtual processor, deleting the process object (and, effectively, all objects associated with the process that the kernel knows about -- from the process' capability list).

The subsystem, on the other hand, performs all scheduling and synchronization decisions on processes. The two important implementation decisions are: the choice of a scheduling algorithm, and the choice of a synchronization mechanism. The choices are arbitrary at this point, since their implementation is hidden in the body of the module. However, given freedom of choice, we would choose a simple priority-based scheduling algorithm which selects the process with the highest priority ready to run, and a synchronization mechanism based on semaphores [Dijk68].

It is debatable whether the PMS should provide visible operations for other subsystems at the same or higher levels. The only such possible operations are those that would affect scheduling decisions on processes. Although it might be useful to permit users to define their own scheduling policies, we do not feel that this is an important enough issue to justify its inclusion. Therefore, we have opted not to provide it at this time since it serves to complicate the implementation for dubious benefits. We may have to revise this decision if it turns out to be wrong.

The kernel interfaces with the PMS via a special kernel-derived object called a *process coordination* object. When a process (object) is created, the kernel also creates a process coordination object and returns a capability for it to the PMS. All communication between the kernel and the PMS is via the process coordination object.¹ Once the kernel creates a process object which is properly initialized, it is passed to the PMS where it is scheduled for execution. As the process executes, contention for system resources among parallel processes is similarly handled by the PMS. This design approach clearly illustrates how the functions of an important system operation can be split between kernel and user space. This, of course, is all in the interest of minimizing kernel code. We will see similar approaches in some of the other subsystems described below.

3.4.2.2 Message Subsystem

The Message Subsystem (MS) represents an alternative way in which processes in SMS may communicate other than making procedure calls. A general message-based communication system is very useful since it permits an arbitrary interconnection scheme between communication components in much the same way that general network architectures do. Furthermore, since it keeps messages out of the address space of processors, it has the desirable effect of reducing memory requirements. SMS implements the MS by providing *messages* and *message servers* - both are implemented as objects.

¹This is actually an implementation of shared memory, with much more control, since all access is mediated by the protection mechanisms of the object model.

The MS is implemented totally in user space as a user-level subsystem. Since messages are based on the protection mechanisms of the kernel, and since we assume that the kernel-call interface can be optimized, there was no apparent reason why kernel code should be complicated by the additional mechanisms of a message system.¹ The MS implements messages and message servers as abstract data types. To send or receive a message, the sender or receiver must have a capability for a message server object. Thus, message servers act as both a source and a sink of messages.

```
M_create_message():capability | signal  
M_send_message():boolean | signal  
M_receive_message():capability | signal
```

M_create_message creates a message object and returns a capability for it. Obviously, since messages are objects, they can be shared. This is extremely useful when the same message must be "broadcast" to a number of objects. M_send_message sends a message to the designated message server. M_receive_message receives a message from the specified message server object. Since the MS is implemented as abstract types, these operations are not available unless one has a capability for each object type; it is normal to make such capabilities available to processes and other entities that wish to communicate.

¹In the implementation of Hydra, the message system was implemented in the kernel. In retrospect, it was thought that it would have been better to implement it in user space.

3.4.2.3 Security Subsystem

In view of the nature of information storage systems, a fairly general and unconstrained mechanism is required whereby users may define their own security policies. Contemporary systems are somewhat restricted in the type of security policies that might be dictated on the use of "files"; for example, in the UNIX operating system, the only protection afforded to files is global read, write, and execute permissions for the owner, a predefined group, or all other users. A more flexible mechanism should allow each owner to apply protection controls, not on an individual file basis, but rather by individual users of that file. This is what we try to accomplish via the Security Subsystem (SES). The SES is a set of modules that provide a higher level of abstraction for the implementation of security policies by users. It is an interface whereby users may define their own security policies based upon the security mechanisms provided by the kernel. In this section we outline the basic architecture of the subsystem; In chapter 4 we provide much more details in view of the importance of this subsystem to system security.

The SES is distributed between kernel and user space. Those operations that manipulate the representation of a capability are implemented in the kernel; other operations are implemented in user space. The mechanism which we use to provide the user with the ability to define his own security policy is based on the idea of *capability sealing*, whereby a user can place arbitrary information in the representation of a capability which can be used at capability-use time to determine if the intended access is permitted (beyond those permitted by the access rights in the capability itself). A simple example will suffice to illustrate the concept. If a user wishes to restrict the number of

operations that can be performed with a capability, the user would seal the capability with this limit information before passing a copy of the capability to another user. Each time the capability is used, the person who sealed the capability is "consulted" before the operation is performed. If the imposed limit has been exceeded, the sealer has the option of denying the request by removing the appropriate access rights from the capability (the reader is referred to chapter 4 for a better and more complete treatment).

The kernel provides the following operations for SES:

```
S__seal():status | signal  
S__unseal(): capability | signal
```

The S__seal operation seals user-specified information in a capability; the S__unseal operation reverses this effect. In order for the kernel to communicate with the user-level portion of SES, sealed information (from the S__unseal) operation is passed back to the calling subsystem via a kernel-defined object; hence the S__unseal operation returns a capability for this object -- the object contains the sealed information.

The portion of SES in user space provides a suitable interface whereby users might avail themselves of the kernel facilities. In general, when a user seals a capability, a type definition object must also be provided which specifies how to interpret the information sealed in the capability. Therefore, when a sealed capability is unsealed, the type definition object's code is used to manipulate this information. A module is provided which allows users to define their type definition object. Another module which extends the seal and unseal kernel operations is also provided, typically allowing the user to manipulate or

review the information sealed in the capability. The details of these modules are elided since they do not impact the overall architecture of the system; they can be implemented in any number of ways using the programming tools provided with, or by, the system.

3.4.2.4 Transaction Subsystem

The Transaction Subsystem (TS) is an abstract data type that implements the nested transaction model [Moss81, Muel83]. The model fully supports the notion of a transaction (called a top-level transaction) composed of other transactions (called subtransactions) that may either execute sequentially or concurrently. As with other subsystems, we find it appropriate to build the transaction model on top of the object model to take advantage of the protection and naming mechanisms provided there.

The nested transaction model consists of a set of rules for the initiation and termination of transactions and nested transactions, a set of rules for maintaining synchronization of concurrently executing transactions, and a state restoration algorithm when transactions or subtransactions abort. The interface to the transaction abstraction provides a set of operations that implement these rules. The primary objective of the transaction abstraction in FTSS is to provide reliable concurrent operations on information storage objects (that is, those objects stored semi-permanently on disks), rather than to provide a truly general-purpose mechanism. In view of this, it is not expected that the overhead of the transaction mechanism will pose a significant obstacle to performance. Therefore, while it is tempting to implement the abstraction in kernel space, we choose to implement it in user space at the subsystem level.

Our nested transaction model is not radically different from those used by Moss and Mueller. In fact, our model is considerably simplified since we are operating in a centralized, as opposed to a distributed, environment. In the usual way, a transaction begins as a single process (called a *top-level* transaction). This transaction may invoke other processes within the same transaction. If one of these processes (or the top-level transaction) invokes another transaction, then this transaction is a subtransaction. Thus a single transaction (or subtransaction) might be composed of several *member* processes. The visible part of the TM provides the following operations (see [Moss81] for a more detailed description of the mechanism):

```

T_create_transaction():TID | signal
T_create_subtransaction(TID):TID | signal
T_commit_transaction(TID):boolean | signal
T_abort_transaction(TID):boolean | signal
T_create_object(TID):capability | signal
T_get_object(capability,TID):data | signal
T_put_object(capability,TID,data):status | signal
T_update_object(capability,TID):boolean | signal
T_get_parent(TID):TID | signal
T_is_committed(TID):boolean | signal
T_is_aborted(TID):boolean | signal

```

TM implements these operations as an abstract data type, coordinating with the kernel or other subsystems when necessary. In particular, support for locking objects is provided by the OMS which supports the call `OM_lock(lock_type)`; this will obtain a read or a write lock on an object. The complementary call `OM_unlock(lock_type)` will free the appropriate lock. A subtransaction commits only if each of its member processes terminate successfully. A top-level transaction (and hence the entire transaction) commits only if all its subtransactions commit, and its member processes terminate success-

fully. Otherwise, the top-level transaction aborts.¹

3.4.2.5 Object Management Subsystem

The Object Management Subsystem (OMS) assists the kernel in managing objects stored on disks in the system. For convenience, we shall refer to this collection of objects as the *Object Store*. It handles general maintenance of the object store by performing location, migration, archiving, and garbage collection of objects.

The OMS consists of the following modules:

- Object Location Module
- Garbage Collection Module
- Archive Module
- Migration Module

Below, we give a brief description of the responsibility of each module:

- The Object Location Module is used to locate objects in the object store. Its primary responsibility is: given this object's unique identifier, return its location. Or, with a finer grain, given an object's unique identifier and a logical block number within the object, return the block. The details of object location is presented in a later chapter after we have developed an architecture for FTSS. However, to give the reader an idea of what is involved, the following paragraph describes a simplified loca-

¹One of the advantages of nested transactions is the fact that each subtransaction can act as a firewall, preventing outside influences from affecting the internals. If the subtransaction fails, it can be successively retried until it commits; thus, a failure of a subtransaction does not imply the immediate failure of the entire transaction. Although more mechanism is required, we feel that this is closer to the real state of affairs and therefore adopt this philosophy in FTSS.

tion mechanism.

An object may exist in one of two states: it may be either *active* or *passive*. An active object is one which is currently being accessed for either read or write and portions of it are in "core"; otherwise, it is passive and resides entirely in the object store. To assist in locating objects, this module maintains a Global Object Directory (GODI) which maps each object's unique identifier into its location in the object store. However, since the disks are intelligent (with their own local processors), each disk maintains local information in a Local Object Directory (LODI) which maps the unique identifier into a physical location on disk. Thus there is a two-level mapping structure inherent in the location mechanism. The advantage of this scheme is that the GODI entries only need to point to a disk, and is therefore simplified. Every object whether passive or active has an entry in both the GODI and one or more LODI's, depending on the replication factor of the object.¹ The module provides assistance to the kernel in the following way. When an object block is to be located in the object store, or when a location must be found for it, the kernel transfers control to the module to resolve the problem. The module will use the GODI to perform the necessary mapping and return the results to the kernel. More details of this is presented in Appendix D.

- The Garbage Collection Module is responsible for performing parallel garbage collection operations in the system to determine unreachable or

¹Since we assumed that disks can fail, we provide high availability by replicating objects on two or more disks that have independent failure probabilities.

circularly reachable objects. A garbage collector is needed in order to solve the "lost object" problem, despite the use of mechanisms such as reference counts.

- The Archive Module is responsible for the automatic or manual archiving of objects in the object store. It presents a visible interface typically only to system administrators, although it is quite feasible that a user can manually request that a particular object be archived. It is expected that archiving would be run periodically by a system daemon in an actual implementation. This module is implemented in user space in its entirety.
- The Migration Module is called upon whenever an object must potentially be migrated either for temporal or spatial efficiency. The visible interface permits the source and destination of an object to be explicitly specified, or that use of an internal algorithm be used to determine an appropriate destination, given a source. This module is implemented in user space in its entirety.

3.4.2.6 System Monitor Subsystem

The System Monitor Subsystem (SMOS) provides a convenient interface for monitoring the use of the system. It is intended primarily for the use of system administrators that must be in contact with system operations. In general, the system administrator is the highest level of authority in the system, and requires a convenient interface to kernel mechanisms. This subsystem is not generally available for casual use. Details of its implementation are elided since they have little bearing on the information storage issues of the system.

3.5 Using the System

The following sections are meant to illustrate how the SMS is used. The kernel and system-level subsystems provide a basic set of primitives sufficient to implement any scenario that a user might want. Objects and capabilities are provided to form the basis of security, processes are provided to support the implementation of a process-oriented operating system, messages are provided to support interprocess communication, transactions are provided to support indivisibility of operations, a security subsystem is provided to allow users to implement their own security policies, and object and I/O management support is provided to manage the object store. For each service, a set of primitive operations are provided; while they may not be an independent set, they are sufficient to implement all the functionalities of FTSS discussed in chapter 2. This section should give the reader additional confidence in the mechanisms we have developed.

3.5.1 The Execution Environment

The most important concept is that of a *program* and how it is executed by the primitives. Assuming that a user can write a sequence of instructions that describes an algorithm, if we can provide a mechanism that can execute them, then *anything* that the user can express as an algorithm can be implemented. This includes such things as text editors, a compiler, a statistics gatherer, etc., which are typically found in a more general computing environment. If the primitives provide support for executing programs, then anything can be built on top of them. In FTSS, sometimes the user will want to build something of his own, at other times he will want to use something which is already

built for him. For the remainder of this section, we will discuss how programs in general execute.

In order for anything to execute in SMS it has to be written as a sequence of instructions for the machine on which SMS itself is built. The trick is for the user to give this sequence of instructions (the program) to SMS and then have SMS pass it on to the base machine. For example, suppose a user wishes to read a file out of the object store, a set of instructions that describes how this is to be done must first be obtained; this is then passed to SMS, SMS will then pass it on to the base machine, making sure that the program satisfies any constraints imposed by the requirements of SMS (security, for example). We will first assume that the user has written his program in a high-level language such as the C language [Kern78]. C provides a set of statements (primitives) that can be used to implement an algorithm. We augment the language with one very important feature - we permit the user to invoke the primitives of SMS directly from the language. When one of these primitives is invoked as the program is executing, a special jump is made to the appropriate code in the SMS; when this code finishes executing, a return is made to just after the invocation, and the user's program continues. It is quite possible that the language provides a different abstraction from those provided by SMS. For example, the language might support the notion of files (consider C again); the compiler writer must make the necessary transition from file operations to object operations, and this is where the SMS primitives come into play. Therefore, for simplicity, we will assume that SMS provides a facility that allows users to write code in a high level language and compile this code for execution.

If it is accepted that any user can write and/or invoke any program, then we only need to discuss how SMS takes this program and executes it for the user. The abstraction that accomplishes this is the process abstraction. Each executing program runs as a process. In theory, the process abstraction is not required, systems have been designed without it, but it makes things more modular, and therefore easier to design, implement, and debug; we use it in SMS because it supports data abstraction and concurrency. Processes are built on top of the object model so that they may use the protection features of the model. A process is implemented as an object that represents the virtual machine on which user code is executed. This virtual machine usually consists of an instruction space, a data space, and an execution stack - it uses the same instructions as the base machine. The structure of a process object is shown in Fig. 3.4. The process object is a distinguished node in a graph of objects that are linked by capabilities (often referred to as the capability graph of the process). This graph is not a tree since it may have cycles. The set of objects in the graph is the set of reachable objects formed from the transitive closure of all capabilities in the graph. The data part of the process object contains process control information, a list of resources being used by the process, and synchronization primitives (such as locking primitives). The purpose of the `P_create()` call is to create an empty process object and to initialize it. Initialization involves the establishment of state vectors maintained in the data part, and installing capabilities for objects that might be needed during execution of the process, for example, a capability for a port object to communicate with other processes or with the I/O subsystem. The `P_start()` primitive establishes the initial addressability of the process by establishing an initial context (described below) and initializing the instruction pointer to point to the first

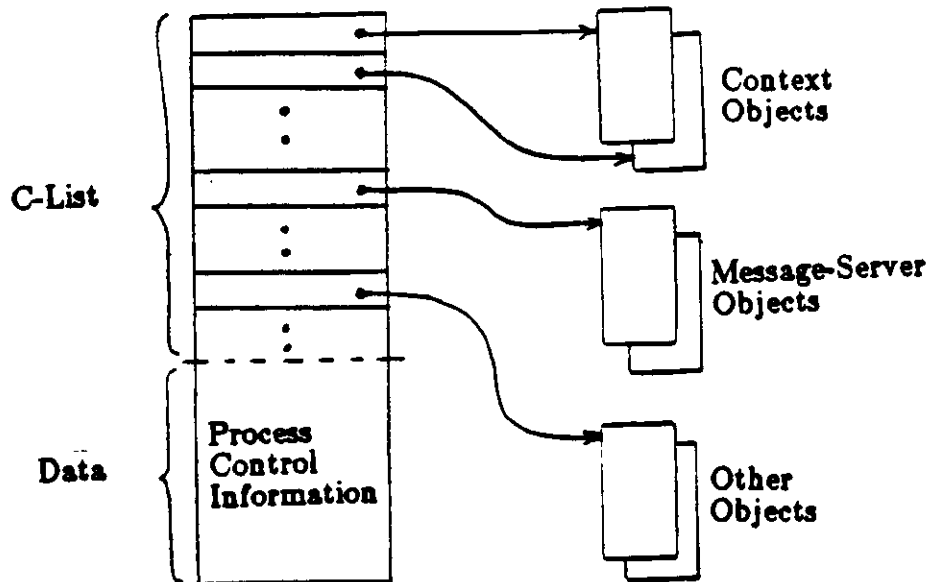


Figure 3.4: Structure of a Process Object

instruction of the main routine in the user's program. It then schedules the process for execution using the mechanisms described in the section dealing with the PMS. In the next paragraph we will describe how execution takes place.

SMS executes each procedure of the user's program in a unique and independent execution domain. This is done using the data abstraction facilities of the kernel. Each user procedure is invoked using the primitive `D_call()` (or `D_call_type()`). This execution domain is defined by another special kernel object which we call a *context* object. Each time a `D_call()` is invoked from the user's program, a new context object is created and a capability for it is stored in the C-list of the process object in a LIFO manner; that is, the Capabilities for each context object are stacked. When a `D_return()` terminates the

procedure, the context stack is popped and the current context object is deleted; that is, the environment is destroyed. Notice that each invocation of a procedure results in a new context (or environment). Furthermore, it is impossible for one context to access objects in another context, unless a capability for the object has been passed as a parameter during the procedure's invocation. This data abstraction mechanism is the root of all protection in SMS; it is the same mechanism, for example, which permits isolation between users in the system - each user executes in a different context, even if they are executing the same primitives. Fig. 3.5 illustrates a process object with several contexts in effect.

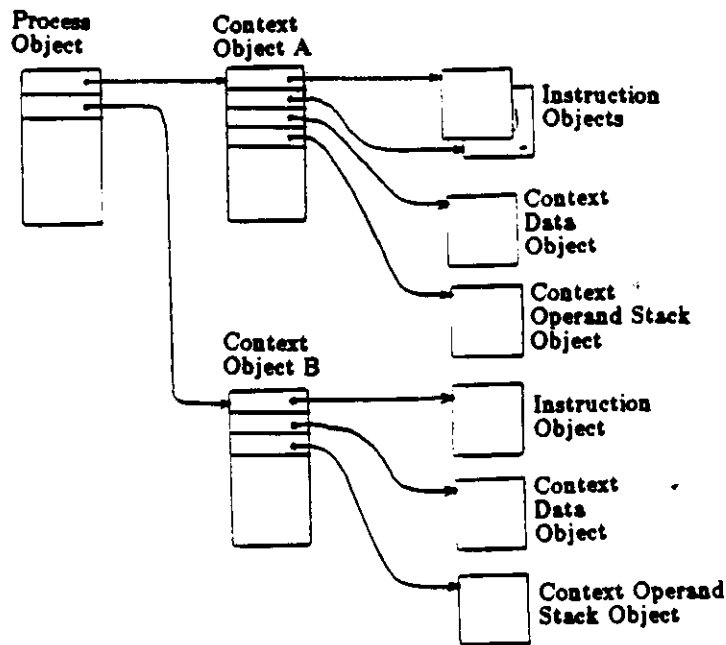


Figure 3.5: Process Object with Several Contexts

Each time a context object is created, its capability list is updated with all those capabilities in the C-list of the instruction object of the invoking procedure, those passed in as parameters during the call, and capabilities to any

temporary objects that the context might need. Now that we have introduced context objects, we can revisit `P_create()`. This call can also be used to create a new process from a currently running one; if the call is given from a currently executing process, a new process is created with the currently executing context as its initial context - the process is not started until the `P_start()` call is used. The calling process continues to run. This behavior is similar to the "fork" system call in UNIX.

Whenever the context stack empties or a `P_kill()` is executed, the process object is destroyed.

3.5.2 Abstract Data Types

A user will want to create an abstract data type whenever it is necessary to exert some control over how a particular equivalence class of objects is manipulated. An example of this is an object of type *directory* which we will discuss below. Each abstract data type is represented by an object which defines the type - the Type Definition Object (TDO). Initially, the kernel provides a TDO from which all other TDO's are created. Thus, the set of objects in SMS forms a 3-level tree as shown in Fig. 3.6. To create an object of a particular type, the user issues an `O_create()` call specifying the type of the object. A capability which permits the creation of new types must be submitted. Such a capability can be obtained by users who are permitted to create new types (this could be controlled by a system administrator, for example). As the distinguished representative of a class of objects, the type object contains capabilities for instruction objects which define the operations permissible on objects in the class. Such operations are invoked by using the `D_call_type()` primitive with

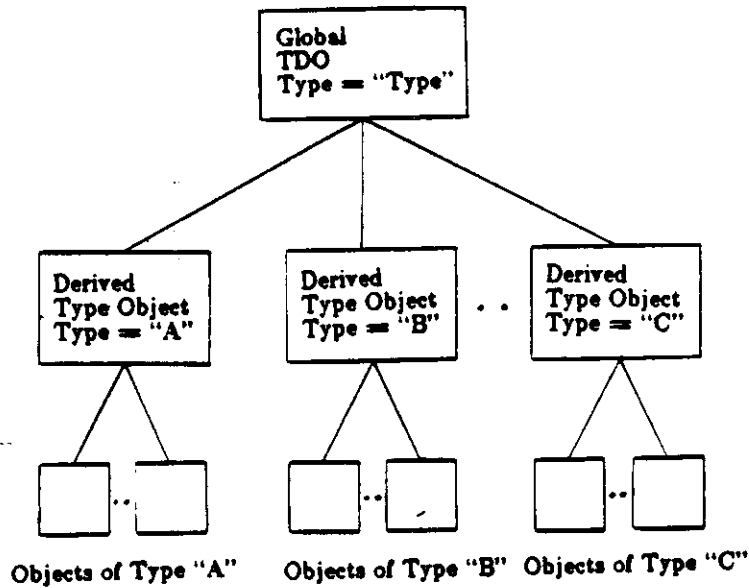


Figure 3.8: Tree of Object Types

a capability for the TDO. In such a case, the instruction object representing the operation to be performed is loaded into the executing processes context and execution progresses. If an illegal operation is specified, an exception is signalled.

It is with this mechanism that users are able to provide their own environments by defining objects of a particular type and enforcing the operations that can be performed on them. For example, a user with a UNIX operating system can declare types called "unix_files" and enforce the operations that can be performed on them.

3.6 Building an Operating System Interface

Let us now discuss how one would build an operating system interface on top of SMS and what types of services it should provide. First, it is necessary to understand that an operating system interface is merely a program. It is a program which never stops (unless killed) which has the ability to invoke other programs; invoked programs run concurrently with it. The operating system interface, therefore, is implemented as a process which is started at some time during the FTSS "boot" procedure. Once it starts, other users can use it to invoke their own programs. Traditionally, operating systems have provided services relating to file, process, I/O, directory, and job management. FTSS is not meant to be a general-purpose operating system, although there is no reason why it cannot act as one. However, we will concentrate on those areas of a traditional operating system that are of particular importance to us. These are object, directory, and job management.

There are several ways in which a user can interact with FTSS. The first is as a remote server where messages are sent to store and/or retrieve information as files, objects, etc. In this case, the user has minimal interaction, merely uploading and downloading whole objects or parts of objects. The second is the case where the user uses FTSS transparently. That is, operations appear to be local to the user's operating system; in this mode SMS is acting as a guest layer on the user's operating system. The third is the case where a user uses FTSS interactively to upload and download files without requiring any particular features tailored to his operating environment. This case is distinguishable from the first in the sense that in the latter the user wants numerous facilities but does not wish to build them. For each of the cases above, however, each

user requires an "environment" in FTSS. We will next describe how these environments can be built and how they are used.

Initially, all users share a common environment, that which we refer to as the operating system interface. This interface is written by the system builders and performs the functions mentioned above. For each new user of the system, a default environment must be established. Furthermore, a logon and authentication procedure is required, as well as a logoff procedure. Let us deal with new users first. A new user is added to the system by a system administrator (or someone suitably authorized) using an operating system program that has been designed specifically for that purpose. There is a global directory of all users managed by the directory management subsystem; each entry contains information which identifies the user, specifies an authentication procedure (it may be a password or a key based on a public-key encryption system), and points to the default environment of the user. Initially, the user's default environment is an *instance* of the operating system interface program. That is, each user gets an individual copy of the interface. Once in this default environment, the user may create other environments, as subsystems, using the type-extension facilities of the kernel, and may install them in the global directory as future default environments.

The logon/authentication procedure always uses the default environment. A user logs on by sending a message to FTSS requesting a connection; the message identifies the user. The global user directory is consulted to verify the eligibility of the user and, if completed satisfactorily, the authentication procedure is invoked. Authenticating a user requires care, especially when the user is remote. The problem is to prevent users outside the protection domain

of FTSS from masquerading as a user or tampering with a user's messages. We would propose an encryption-based scheme based on public and private keys; each user is given a key and another key is kept in the global user directory. Once a user is authenticated by providing his (encrypted) key, a virtual circuit is established over which all communication takes place until he logs off. This implies that all messages over the virtual circuit are encrypted. The other alternative is simple, but less secure: give the user an unencrypted password and assume that the external communication system is secure. Capabilities are not permitted outside the protection domain of FTSS!

Logoff is conceptually simple. When a user logs off, all the objects that belong to the user must be preserved and the user's environment must be destroyed. Destroying the environment is simple - we just kill the process. However, preserving the user's state so that the next time he logs on he can observe the same state as when he logged off requires some form of object state preservation. The way this is done, if it is done at all, depends on the semantics of the user's environment - a user who creates his own environment is free to make arbitrary decisions. The default environment writes all modified objects back to the object store (in case the logoff is due to communication failure), updates the user's local directories, and updates any information necessary in a state object associated with the user (all users are assumed to have one stored in some standard place).

Recall that a user, once logged into the system, may change his default environment by updating the entry in the global user directory which points to the default environment. Once in the default environment, a user may create and/or switch to new environments (user-supplied or publicly accessible)

dynamically. The method of doing this uses two key kernel primitives: the data abstraction facility and the type extension mechanism. The data abstraction facility allows the nesting of environments by treating each as a context of the original environment. However, the context is more general in that the new environment runs as a separate process. The type extension facility permits a new environment to be created by defining the environment as an abstract data type.

Now that we understand what a user environment is and how a user establishes contact with it, let us go back to the main functions of the operating system which provides services for object, directory, and job management. Object management is concerned with the management of objects in the object store; it permits users to access objects that may be either public or private and to perform a limited amount of management on them. Directory management permits users to create and maintain directories of names referring to objects or other directories. Job management is concerned with the management of a *job* or a family of processes. If one were to express these three as a hierarchy, then job management is needed for both object and directory management, and directory management is needed for object management. Directories are implemented as an abstract data type (a user-level subsystem) since most users will want to use directories, even if they design their own environments. Job and object management are implemented as programs since they are generic. We will not discuss the actual functions implemented by each of these management tasks; they may be assumed to be similar to those found in any modern operating system. Instead what we will do is to illustrate the object state of the operating system as each of these tasks execute from logon

to logoff. In doing so, we will follow steps that would typically be taken by a user. These steps are:

1. User logs on
2. User creates an object, makes additions to it and stores it in the object store.
3. User reads an object already stored.
4. User creates a new environment and installs it as his new default.
5. User logs off

We now present each step in more detail.

Logon

We assume that the system is currently executing a login process which is always running, waiting for users to log in. This process must communicate with the user and authenticate his access to the system; we will assume that the user has used the system before and has established an authentication mechanism. We now take a snapshot of a portion of the object state of the system showing which objects are currently in use, and how they are connected to form a capability graph during the authentication procedure. This is shown in Fig. 3.7. If the user is authenticated, then the login process will invoke the operating system interface and attach the user to it. The operating system interface is invoked as a process independent of all other processes in the system; with respect to the user, Fig. 3.8 shows the object state just after the operating system is invoked. A command processor context is invoked and the user is attached via a communication port; the command processor is waiting for commands at this point. Logon is now complete.

Object creation, modification, and storage

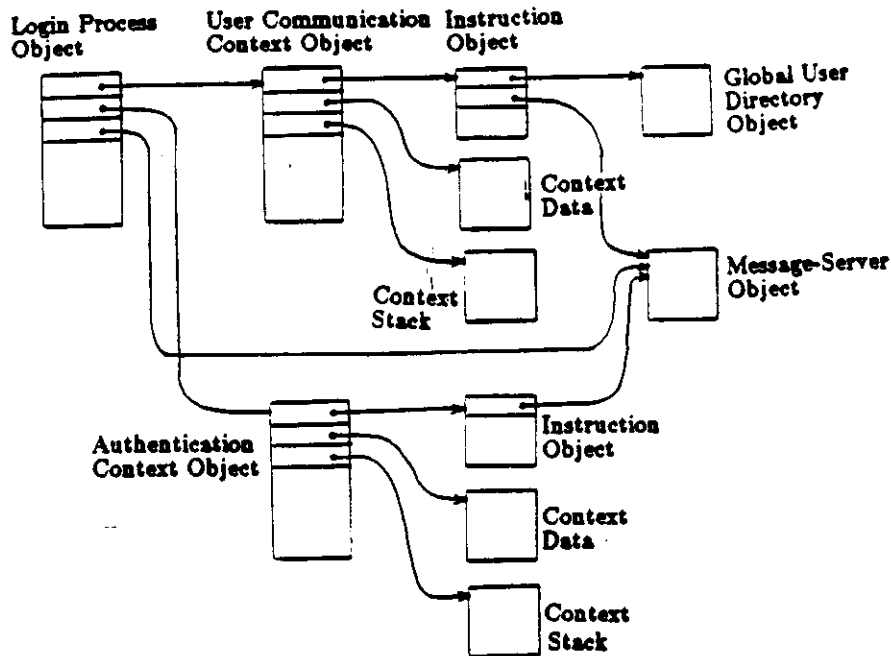


Figure 3.7: Snapshot of Login Authentication

We assume that the operating system provides commands to execute each of these functions. Creating an object (let us assume it is a text object) implies that the user's current directory will be updated with the name of the object once it is created. Objects are created by making an appropriate call to the kernel; adding the name to the local directory implies calling the directory subsystem. Since object creation involves no new contexts, we will defer showing an object state diagram until the next step which is to add some text to the object. There are two ways in which this can be done: the user can either send the text as a message, or do it interactively using, say, a text editor. As far as the object state is concerned, it makes no difference which method is used (only the context would be different). We therefore assume that the user is uploading information from his local processor. We also assume that the

operating system interface supplies a command called "upload" which will allow the user to add data to the designated object. Fig. 3.9 now shows the object state. The order of the context objects from top to bottom indicates the calling hierarchy.

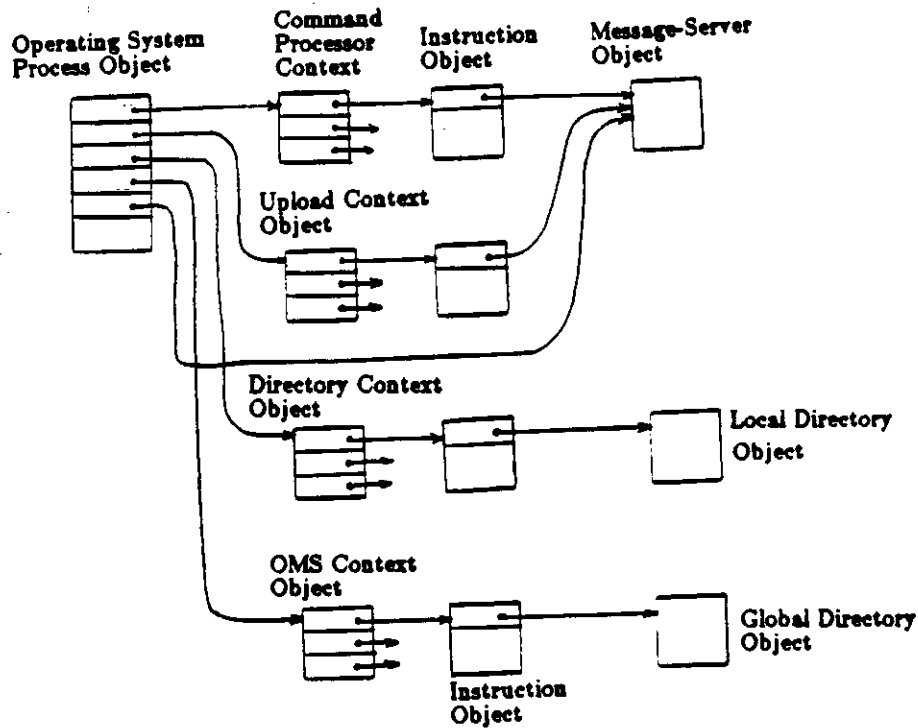


Figure 3.8: Object Update

We should point out that the structure shown is very simplistic; it does not indicate all possible objects, just the ones of importance - the most important thing is to note how various contexts are established, and their relationship to one another. When a context terminates (via a `D_return()`), control returns to the previous context as they are stacked in the process object. When the object is to be stored, the user invokes an appropriate "store" command and the

object is written to the object store. The details of writing an object to the object store is handled by the operating system interface via the `O_put_object()` kernel call - only a capability for the object is required.

Reading an Object

To read an object, we assume that the operating system interface command processor provides a "read" command. The semantics of this read command is simple: it takes the name of the object from the user, looks it up in the directory specified, retrieves a capability for the object and invokes a kernel primitive to get the data (this may involve more than one use of the kernel primitive depending on how much of the file the user wants to see). The data which is read is transmitted to the user. The object state we would show is exactly the same as in Fig 3.8 except that the upload context is replaced with a "read" context.

Creating a new environment

A new environment is tantamount to a new copy of the operating system which replaces the old. The data abstraction (subsystem) facility can be used for this although, as we saw with the directory subsystem, it is not necessary that the subsystem be used in this way. The new subsystem can be invoked either on demand by a holder of a capability for it, or by default when the user logs in - we shall assume the latter. Building an environmental subsystem is exactly the same process as building any other subsystem - the user creates a new abstract type and installs instruction objects that define operations on the type. It is normal for users to create a new environment when the default environment does not meet their needs. To facilitate a new subsystem,

the default operating system provides the tools that the user needs - a text editor for writing source code for the subsystem operations, a compiler for generating machine code, commands to put the machine code in instruction objects, and a possibly interactive program for putting the subsystem together. The snapshot of Fig. 3.10 shows the state of affairs just as the subsystem builder is installing capabilities for instruction objects into the new type object.

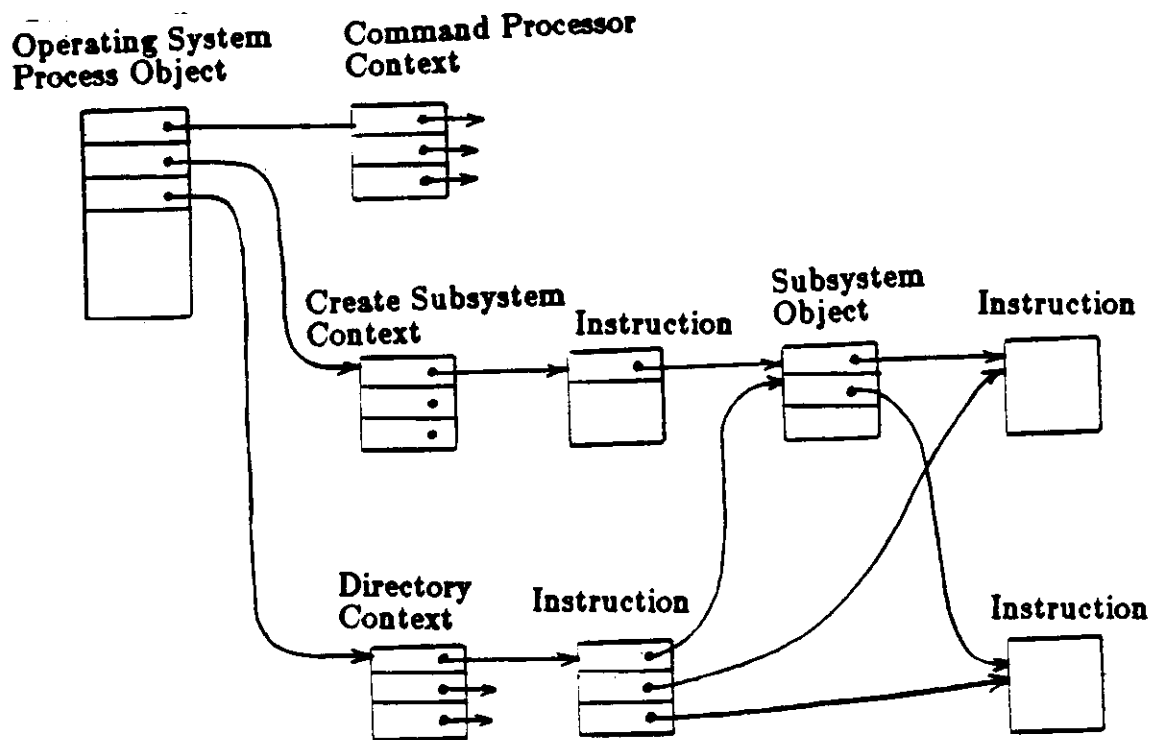


Figure 3.9: Snapshot During Subsystem Creation

Once the subsystem is built and the user has the capability for it, it can be installed in the global directory object by using an operating system installation program. The installation program simply installs the capability for the

subsystem object in the global directory object replacing the one which pointed to the previous default environment.

Logoff

If we assume that the user is in his default environment, then logging off causes the current environment process (operating system or user subsystem) to reinvoke the login process; it then kills itself. For example the operating system process would invoke the login program as an independent process and terminate the command processor context - this will cause the operating system process to die since there are no outstanding contexts.

Much of what we have discussed in this section is what we would implement; however, the mechanisms are sufficiently flexible that virtually no constraints are placed on what can be built. In appendix E we have provided skeletons of operating system functions that further illustrate how the basic primitives can be used to implement useful interfaces.

3.7 Summary

In this chapter we have reviewed architectural issues for the design of a storage management system for a centralized storage system. The most important issues are related to extensibility, security, and availability. We have developed a software architecture for an SMS which is based on the object model, and which uses capabilities both as a means of enforcing protection, and as a naming mechanism. In chapter 8 we shall propose a hardware architecture upon which this software architecture can be mapped -- an environment which provides support for high availability and security while, at the same time,

enhancing the performance of the system.

CHAPTER 4

SECURITY ISSUES

Information security in online storage systems is a means of preventing the unauthorized disclosure of information. Several novel schemes have been proposed to deal with this problem. However, their primary limitation is the amount of flexibility they impose in the implementation of arbitrary security policies. In this chapter, we describe a mechanism for implementing arbitrary security policies based on access controls. The mechanism uses the concept of capability sealing in which a security policy is conceptually sealed within a capability; there is very little restriction on what sort of information might be sealed within a capability. The approach differs from other capability sealing approaches in that it is applicable only to information storage objects, and in the mechanism by which information is sealed in and extracted from a capability.

4.1 Introduction

The issues of *security* in programmed systems are well known. Security regulates activity both internal and external to the programmed system. A secure computer system depends upon:

1. controlled access to objects within the system,
2. reliable hardware components,

3. prevention of threats perpetrated outside the system,
4. controlled disclosure of information within the system, and
5. controlled disclosure of information obtained by inference within the system.

In FTSS, we are interested in providing security mechanisms that permit a wide range of policies to be implemented by the user on objects stored in the system.

Several techniques have been proposed for meeting the security requirements of operating systems, of which the capability mechanism appears to be the most promising [Ekan79, Rede74, Glig79]. Similarly, several mechanisms have been proposed for meeting the security requirements of files in information storage systems (as, for example, the use of access matrices, access control lists, and so on) [Jone79]. The capability mechanism has also been used to a limited extent in some extant systems (proposed or implemented) [Orga72]. Applying security mechanisms via capabilities across the system (although desirable) leads to complications since it is necessary to implement these mechanisms at kernel level, resulting in a kernel which is difficult to verify. Access matrices and access control lists, while useful for implementing access control policies on files (and therefore information storage objects) do not have desirable semantics when applied to revocation; neither do they permit fine-grained control over capability copies in a capability system. (See [Glig79] for a discussion of capability copy and access review).

Because of the diverse security requirements of operating and storage systems, several different security mechanisms have been developed to satisfy the requirements of each part of the system. This has resulted in difficult interfaces between subsystems, often at the compromise of security. For example, in Multics [Orga72] capabilities are used internally whereas access control lists are used in files; thus two different access control mechanisms are used and need to be verified. What is needed is a homogeneous integrated mechanism which can be used in all parts of the system. The goal of this chapter is to provide the basis of such an integrated mechanism that can be used to satisfy arbitrary security policies.

4.1.1 Limiting the Scope

In this discussion, we are mainly interested in *information security*.¹ Its implementation implies the implementation of various *controls*. It has been argued that *access controls*, *information flow controls*, *cryptographic controls*, and *inference controls* form a sufficient set of controls whereby a wide range of useful security policies can be implemented [Denn82]. Our goal is to provide a flexible enough mechanism that will permit the implementation of all (or a sufficient subset) of these controls.

One aspect of information security that has received considerable attention is *protection*. Protection can be loosely defined as the enforcement of a rule, or set of rules, that maintains order during the mapping from input to output values of the set of algorithms that make up the programmed system [Jone73]. The algorithms may be implemented in either hardware or software. While the goals of protection vary markedly from one system to another, the

¹Other authors has referred to this as *data security*).

following are applicable to most systems:

1. concurrent users should be able to cooperate without unnecessarily *interfering* with each other,
2. users should be able to store information permanently or semi-permanently so that it is inaccessible to others without authorization, and
3. users should be protected from themselves: resources and powers a user employs while performing one task should not be available (for possible misuse) when performing another task.

Because of the wide range of so called *protection policies* that can generally result due to diverse systems or applications, it is best to provide the user with *protection mechanisms* from which arbitrary (but useful) protection policies can be developed. Protection mechanisms based on access controls have been demonstrated to be quite useful as the basis of a protection system, especially when coupled with the capability mechanism. We therefore provide for protection of information storage objects via the access control mechanisms applicable to capabilities.

In the following discussion, all of our proposed security mechanisms are based on some form of access control mechanism.

4.2 The Basic Mechanism

We propose a mechanism whereby security may be implemented to any degree of effectiveness needed in the system. While the proposed mechanism can be used in a very general sense on all capabilities in all parts of the system,

our intent is to use it solely for the application of security policies to objects stored semi-permanently in the system. In particular, we do not propose that the mechanism be used on capabilities for objects that are dynamically created and manipulated by the kernel (for example, process, device, and other kernel-defined objects). By making this assumption, it is possible to remove much of the mechanism from kernel space to user space; this is an important consideration if the security of the kernel is to be verified. Also, by taking this approach, the security of the mechanism need not be verified since its failure cannot subvert the basic protection mechanism provided by the kernel (at worst a user can be restricted from accessing an object, but never given access to one for which the appropriate rights are not present). The mechanism is built on top of the basic capability mechanisms of the kernel, and makes use of the concept of *capability sealing*. Generalized capability sealing was first proposed by Redell [Rede74], where it was used to provide type extension and revocation in a capability-based system. Here we use the methodology, extending it slightly in a different direction, to provide the mechanisms needed to implement security policies.

4.2.1 Capability Sealing

The idea of sealing a capability is analogous to putting a capability in a *box*. Conceptually, the box has no independent existence, acting more or less as the "skin" of the capability. There may be several types of boxes, and there is no restriction upon the number of boxes in which a capability might be encased. Boxes are created from *templates* that are either predefined or defined by the sealer of the capability. When a capability is sealed, the box acts as a *protection domain* to which access can be gained only through the proper

authorization. By providing access control policies to these boxes, we will show that it is possible to create a mechanism with desirable properties.

A capability is sealed by providing a suitably authorized template for a box. This is normally done by providing a capability for a type object which defines the template. The following operation is illustrative of the concept:

$$C_S \leftarrow \text{seal}(C, C_T)$$

Executing *seal* creates capability C_S by sealing capability C in a box specified by the template contained in type T , as authorized by the privilege of sealing in capability C_T . The following *unseal* operation reverses the process:

$$C \leftarrow \text{unseal}(C_S, C_T)$$

These operations require further explanation. However, before we do so, we outline the goals of the mechanism. First of all, the mechanism must provide the basis for arbitrary security policies. Second, it must provide for the selective and unconstrained revocation of rights. Third, it must provide for the selective and unconstrained restriction¹ of rights. Fourth, it must not in any way restrict the operations that can be performed on *base* (unsealed) capabilities, save for some penalty in the execution performance of those operations.

It should come as no surprise that the action of sealing a capability in different boxes can be used to implement arbitrary security policies. In effect, what we are proposing is two levels of access control policies, both of which are enforced by the system. The first is a policy for accessing the representation of

¹Restriction and revocation have slightly different semantics. Revocation is the *removal* of a right, whereas restriction is some *constraint* upon the use of a right.

capabilities, the second, a policy for accessing objects. It is appropriate to treat them separately since capabilities are not objects. Let us now look at the policy for accessing the representation of capabilities.

We begin with a base capability. A base capability typically consists of a set of *rights* and a *unique identifier* or UID. A right is a privilege which permits the possessor of the capability to perform some operation on an object to which the capability points. To restrict an operation on an object, it is sufficient to restrict the rights to that operation in the capability which is used to access the object. In order to establish a generalized mechanism for rights manipulation in a capability-based system, it is important that the use and interpretation of (at least some of) the rights be user-definable. This approach has been used in a number of systems in which there are a set of predefined (or *kernel*) rights and a set of user-definable (or *auxiliary*) rights [Wulf81]. A limitation of these schemes, however, is that the set of user-definable rights is limited (usually based on the physical size of the capability, in bits). A more general mechanism should put little restriction upon what sort of "rights" can be defined by the user. Of course, this leads to the problem that the size of a capability is now variable, a fact which must be accounted for in the design of kernel mechanisms which manipulate capabilities.

The kernel rights are interpreted directly by the kernel and form the basis of protection in the system. This is inviolate and vigorously enforced by the kernel on all object accesses. The interpretation of these rights is system-dependent. We will discuss some of their possible interpretation as the need arises. All kernel operations on objects are done in terms of base capabilities. If the kernel is presented with a sealed capability, an attempt will be made to

unseal it to the base capability before it is used. As mentioned before, the operation of sealing merely encapsulates (and insulates) the base capability in a very special way.

It is convenient to think of a capability as being a *record*, in the programming language sense. When a capability is sealed, this is analogous to adding another record to the record which represents the base capability. Note that we use the notion of a record merely as a descriptive device and not as suggestion for a possible implementation. Each time a capability is sealed, a new record is added to the existing set of records. Conversely, the unsealing of a capability is analogous to the removal of a record from the set of records. Clearly, it is not feasible to unseal a capability which is not sealed (i.e., there is only a single record in the set). Since the order of sealing may not have a one to one correspondence with the order of unsealing, it should be possible to randomly access any record in the set. This can be done using any of several standard methods available to the implementor.¹ Using a simplistic programming system, a sealed capability might be structured as follows:

```
base_capability = record
    UID: integer pointer
    system_rights: bit vector
    sealed_info: integer pointer
end

sealed_info = record
    type_capability: capability
    user_info: string
    auxiliary_rights: bit vector
    sealed_info: integer pointer
end
```

Thus, a sealed capability can be viewed as a linked list consisting of a base

¹One possibility is as a doubly linked list of records.

capability at its head followed by none, one, or several information records that describe the sealing operations.

It is perhaps easiest to view a sealed capability as an ordered set of boxes, which we term *prioritized sealing*. This is, in fact, a two-dimensional representation of the more general view that a sealed capability is a set of nested boxes in which the outermost box represents the most recent sealing operation. In terms of an ordered set, a sealed capability S is a one-dimensional array of elements (boxes) s_i , $0 < i \leq n$ such that s_1 represents the base capability, and s_n represents the most recent sealing operation. For simplicity, we assume that the format of each box is identical, with the proviso that some of the fields of the boxes are not used (null fields) when they are not needed.

It is the base capability that is always used by the storage management kernel. The rest of the capability (the records indicating various sealing operations) is merely used to modify the rights in the base capability. Initially, a base capability has all rights *ON*. From this point on, rights may be explicitly restricted (via the *restrict* operation provided by the kernel), or implicitly via various sealing operations. It should be noted that a right, once removed, can never be reinstated in the base capability (except possibly by *amplification*¹).

If we take the view that a sealed capability is an ordered set of boxes (or an ordered list), then when the capability is unsealed, it is necessary to investigate the information sealed in the capability for all boxes (list items) with a higher priority than the one being unsealed. As the list of records is being scanned, an attempt is made to match the capability of the type object

¹Amplification is the addition of a right to a capability without an explicit request from the holder.

presented as part of the unseal operation with that sealed in the capability itself. If no match can be made, an error status is returned. If a match is made, then that record is unsealed using the type object, and immediately following this, all other records of higher priority are automatically unsealed by the system (without requiring an explicit capability for the defining type object). Thus, if the i th box s_i , $1 < i \leq n$ is to be unsealed, then a total of $i - 1$ unsealing operations are required to obtain the base capability.

The purpose of the type definition object is to define those operations that are permissible on the object to which the base capability points. Fundamentally, there are no restrictions upon the type of operation that might be imposed on the capability by the type object. In fact, the `user_info` field of the capability permits arbitrary information to be stored in the capability itself. The idea of recursively unsealing the capability is to allow each higher priority sealant the opportunity to observe how the object being pointed to is about to be used. This is accomplished through an interpretation of the `user_info` and `auxiliary_rights` fields of the sealed capability.¹

4.3 The Mechanism in Use

Consider the following scenario which illustrates how the basic mechanism works: Fig. 4.1 shows a base capability that has been sealed twice. It has first been sealed by user *A* and then by user *B*. We will assume that *B* has defined an operation on a base object whose task is to make a copy of the object, producing a new object and a base capability for it. In order for the

¹In principle, the auxiliary rights field is not required since it can be imagined to be a subfield of the `user_info` field; however, we distinguish between the two in order to maintain semantic difference between what a type object requires and what other information the user chooses to seal in the capability.

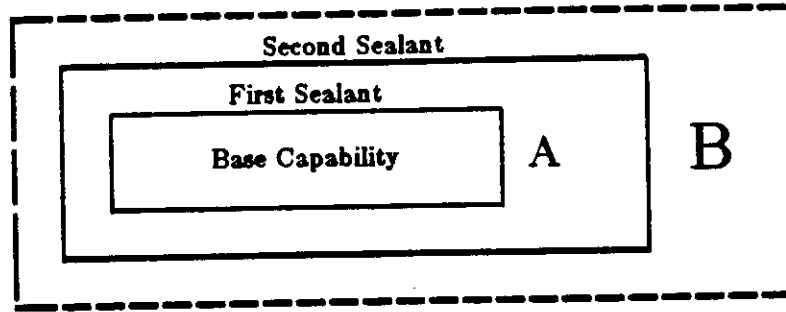


Figure 4.1: Sealed Capability

operation to be successful, the rights to read and write the object are required (we do not assume the existence of an explicit *copy* right, although this is such a common operation that one might indeed be desirable). In order for the copy operation to proceed, *B* must (indirectly) provide the base capability to the system. However, before this is done, the capability is recursively unsealed based on the reverse order of sealing. Thus, *A* is effectively “consulted” to see if the operation requested by *B* is allowed to proceed. Note that, independent of *A*, if the base capability has either read or write right missing, the operation cannot proceed. Therefore, a simple optimization is to check the base capability to see if the required rights are present before recursively unsealing the capability. If both read and write rights are present in the base capability, then it is possible that *A* might have placed some restriction on their use. For example, *A* may have imposed some limit on the number of read and/or write operations that can be performed on the base object. Or, *A* may demand to know the identity of the holder of the capability in order to allow the operation to proceed. It is important to realize that the ability of one user to place restrictions upon the use of a (sealed) capability is handled on a capability-by-capability basis. This permits very selective revocation or restriction of rights

and permits the finest granularity of control in the system.

Given the mechanism just described, it is now possible to describe how it can be used as a means to implement several security-related policies in the system.

4.3.1 Protection

The use of capabilities and access rights as the basis of a protection mechanism is well known [Jone73, Denn66, Lamp69]. The mechanism of capability sealing that we have proposed here in no way affects the use of capabilities and the rights mechanism as the basis of a protection system. In fact, the only reason we seal a capability is to provide an additional level of protection by ensuring that the base capability cannot be used in arbitrary ways. We therefore continue to assume that each capability contains a set of rights which are enforced as part of the protection system, but which can be modified through the actions of sealing and unsealing. The rights of the capability do indeed become the basis of all security policies in the system.

4.3.2 Revocation

Revocation is the act of removing a right from a base capability that is not in the possession of the "owner". There are several flavors to revocation, depending upon the semantics of the application and its environment. However, it is generally agreed that revocation should, at least, have the following properties:

1. It should take effect immediately.

2. It should be possible to revoke the various privileges in a capability independently.
3. Revocation should require no global knowledge of capability propagation.
4. Any distributor of a capability should be able to revoke its privileges (i.e., not just the "owner" of the object).
5. The users of capabilities should not need to distinguish between revocable and non-revocable capabilities.
6. The cost of revocability should not be excessive.

There are two approaches to implementing revocation: By using independent or dependent capabilities. In an independent capability scheme, a special *revoker* capability is used to break the mapping between a capability and the object to which it points. In the dependent capability approach, revocation of a privilege from a capability implies the removal of that privilege from all capabilities that are, in a sense, dependent on that capability. The dependency relations between capabilities depends on how capabilities are passed around in the system. The advantage of the independent capability approach is that revocability, since it is enforced with a capability, is itself revocable. It is not clear which approach is dominant since they both lead to workable solutions. Redell [Rede74] describes a revocation method based on dependent capabilities. Independent capabilities were used in the design of a secure operating system by Neumann [Neum77]. Our method of prioritized capability sealing can be used to implement revocation schemes based on either

independent or dependent capabilities. We illustrate next how the mechanism can be used to implement a very general form of revocation based on dependent capabilities while removing some of the limitations of previous implementations.

In our implementation, revocation is achieved through prioritized capability sealing. There are two novelties of the implementation. First, there is no explicit right of revocation; that is, any sealer of a capability has the right to revoke privileges from that capability. If we assume that all users have the right of sealing, then it can be concluded that all users have the ability to perform the action of revocation. Second, revocation can take place selectively on a capability-by-capability basis, as well as globally for all capabilities. The mechanism works in the following way: If a user wishes to revoke privileges in a capability, he simply seals that capability in a box defined by a template contained in a user-defined type object. Since the type object is user-defined, arbitrary policies can be dictated by the user to determine when a right might be revoked. We saw one such policy in a previous example based on the operation of copying an existing object. A concrete example is useful here to illustrate the flexibility of the mechanisms.

Consider a user *A* who wishes to restrict the number of copies of an object that may be made. *A* seals the capability in such a way that it maintains a count of each time a write operation is performed with the capability, and passes it to user *B*. User *B* in turn makes a copy of the sealed capability (without resealing it) and passes it on to a user *C*. If either *B* or *C* attempts to use the capability, it will be unsealed to check for any restrictions that *A* might have placed on the use of the capability. After *A* determines that an

appropriate number of write operations have been performed, he then removes write rights from the capability (forever). Note that the removal of write rights from this capability does not affect similar rights in other capabilities sealed by *A*. This is therefore illustrative of *selective revocation*. Note also that any capability which was derived from the one *A* passed to *B* will also effectively have write rights removed since, when they are used (and therefore unsealed by *A*) write rights will be immediately removed. Thus whereas rights are not removed immediately from all outstanding copies of capabilities, they are effectively removed since any attempt to use them will result in immediate rights revocation.

It has been stated that it should be possible to revoke rights from all outstanding (dependent) capabilities and that this revocation should have immediate effect. It should be noted that any capability from which rights can be revoked must be sealed. If a user possesses a base capability, then ostensibly that user has unrestricted use of the rights of that capability, and they are not subject to revocability. While this may appear to be a limitation, this is not necessarily so. It merely means that a user must plan for revocability (by sealing) before passing on a copy of a capability.

In some implementations, a copy of each dependent capability is kept by the owner of the capability. When the capability is to be used, this copy is used to replace the original. Global revocation can then be accomplished simply by removing the rights in the copies kept by the owner. In our scheme, it is not necessary to maintain a copy of a capability since the rights of access are automatically checked on each use of the capability. One may argue that recursive unsealing on each access is inefficient. However, there are two things that

should be considered. First, it is possible to perform optimizations at implementation time in the sense that once a capability is unsealed, the base capability can be kept in some readily accessible location so that it can be used directly without the need for unsealing the original capability. Second, in light of the possible utilization pattern for permanent and semi-permanent objects, it does not appear that sealing and unsealing will occur frequently enough to have a significant impact on system performance [Smit81].

4.4 Implementation Issues

As explained in section 4.2, capability sealing is implemented as a subsystem in FTSS which presents a visible interface to users. In this section we present more details of the implementation.

4.4.1 Kernel Support

The kernel provides direct support since the representation of a capability must be modified by sealing and unsealing operations. Although the actual base capability is not modified in sealing/unsealing, for security reasons, only the (verified) kernel must be permitted to access the locations of an object which contain capabilities; viz. the capability list. If user code could directly access these locations for sealing/unsealing (even by a "trusted" process), security could be subverted either by a malicious process or one that exhibited unaccounted-for failure modes. In our implementation, the kernel manipulates sealed capabilities by stripping sealed information from a capability and passing it to the rest of the security subsystem in user space via a predefined kernel object. For example, if a (sealed) capability C is to be unsealed, the kernel strips all sealed information from C and passes it to the security subsystem in

user space along with a copy of the base capability C_b . The subsystem may now freely manipulate this information, making the necessary calls to the kernel to restrict rights if necessary. A similar situation takes place when C is to be sealed; the subsystem collects all the information to be sealed with C_b and passes it to the kernel along with C_b . The actual sealing is done by the kernel. There are now a number of further details that must be explained.

First, the kernel must be able to distinguish between capabilities that are sealed and those that are not. This is easily accomplished by using a special flag in the capability which is set when sealed, and reset otherwise. Second, if a sealed capability being used by the kernel is to be copied or in any way transferred out of the kernel, its sealed version must be used. This implies that the kernel must keep track of which of the capabilities currently being used in kernel operations are sealed. One way to do this is to put another flag in the capability. Another is to keep a table in the kernel which can be searched each time a capability is to be moved out of the kernel; this table simply maintains the UID (or address) of active sealed capabilities. In view of the fact that sealing is applicable only to capabilities pointing to objects that represent information stored online in the storage part of the system, it is unlikely that there will be a large number of such objects currently "open" at any one time. Therefore a table search should not be prohibitively expensive. Of course, this must be weighed against the cost of the extra bit in the base capability. There is another case for the kernel table approach. If a copy of a capability which has already been unsealed is presented to the kernel for sealing, an optimization is immediately realized by searching the table of active capabilities. Thus, repeated unsealing of the same capability is prevented. Of course, much of the

same arguments apply to the case when capabilities are to be sealed.

The kernel assumes that sealed capabilities have the following structure:

- A base capability (with the sealed capability bit set)
- An ordered list of capabilities for type objects which define the operations to be performed on the information sealed with the capability, and
- An ordered list (in one-to-one correspondence with the list of capabilities for type objects) of information sealed with the capability.

Each time the capability is (re)sealed, a capability for a type object and information to be interpreted by the type manager for the type object is added, each to the corresponding list. Since the size of a sealed capability is unbounded, it would be inefficient for the kernel to manipulate sealed capabilities directly. Therefore, a sealed capability exists in one of two forms: the *long form* as represented by the structure discussed above, and the *short form* which consists of just the base capability (the first element of the structure). When the kernel is first presented with the long form of a sealed capability, it preserves the ordered lists of type object capabilities and sealed information in a table indexed by the UID of the base capability. From this point on, the kernel always uses the short form. In order to keep things simple, we make the decision that the only time a short form capability is converted to its long form is when an object containing a short form capability is moved out of the server and back into the storage system. This determination could be made by the kernel. However, in order to minimize kernel code, it is moved into the OMS; the OMS scans the capability list of any object to be written back to storage and requests the kernel to convert any short form capability found back into long form.

A sealed capability is always presented to the kernel as a pair:

<[long,short] form capability, type object capability>

If only the long or short form capability is presented, an error is signalled. The type object capability is needed by the SS to determine where in the ordered list of sealed information to begin recursive unsealing, and also how to do it (via the abstract type).

In summary, the kernel provides operations for sealing and unsealing capabilities, and for converting long form into short form capabilities and vice versa. The conversion mechanisms should present no problems for the reader, based on the discussion above. Below, we outline the algorithms for sealing and unsealing capabilities.

Algorithm 4.1: Sealing

```
/* The seal operation is invoked by an external subsystem.
The following parameters are expected: A (possibly
sealed) capability  $C$ , a capability for a type definition
object  $C_T$ , and the information to be sealed in the
capability  $INFO[]$ . The operation returns the sealed
capability  $C_S$ . NOTE: Since external subsystems may
not manipulate capabilities, specifying a capability  $C$  is
equivalent to specifying a capability for an object and an
index into the capability list of that object in which  $C$ 
will be found; the returned capability  $C_S$  will be
stored in the same place. */
```

step 1: Check the sensibility of the parameters. If any parameter not sensible, signal ERROR and stop.

[Assumption] Any errors in the following steps will lead to a signal ERROR and the algorithm will be terminated.

step 2: If C is sealed and in short form, index into the table of sealed capabilities using the UID (see step

2 of Algorithm 4.2 below) and retrieve the C_T and $INFO[]$ lists.

step 3: Append C_T to the (possibly empty) list of capabilities for type definition objects:
 $LIST[C_T, LIST[C_T]]$.

step 4: Append $INFO[]$ to the (possibly empty) list of information for each sealer of the capability:
 $LIST[INFO[], LIST[INFO[]]]$.

step 5: If C was originally in short form, update the kernel table and reconvert C to short form.

step 6: Return C_S and stop.

Algorithm 4.2: Unsealing

```
/* The unseal operation is performed automatically by the
kernel on any sealed capability to be used in an operation
on an object. The unseal operation is not performed when the
kernel performs an operation on a capability (such as "copy
a capability"). Unseal expects a sealed capability  $C_S$ 
as a parameter. */
```

[Assumption: If there are any errors in each of the following steps, signal ERROR and stop.]

step 1: If the capability is in short form, go to step 6.

step 2: [Assumption: C_S is in long form, in which case this is the first time it is being seen by the kernel. We also assume that the kernel table for holding long form capabilities has been properly initialized.] Use the UID of the base capability in C_S as an index into the table.

step 3: Insert the list of type definition capabilities $LIST[C_T]$ into the table as a LIFO list (stack) such that the most recent sealer is at the head of the list (on top of the stack).

step 4: Repeat step 3 for the list of sealed information: $LIST[INFO[]]$.

step 5: Convert C_S to short form.

step 6: Create SYS _SEAL object (for communication with external subsystem).

step 7: Put C_B , C_T , and LIST[C_T] in the capability-list part of SYS _SEAL, where C_B is the base capability. Thus, the capability list of SYS _SEAL is ordered as: LIST[C_B , C_T , LIST[C_T]].

step 8: Put OP _CODE and LIST[INFO[]] in the data part of SYS _SEAL as: LIST[OP _CODE, LIST[INFO[]]]. OP _CODE is an operation code for the operation currently being performed by the kernel.

step 9: Signal external subsystem and pass it the capability for SYS _SEAL. Wait for response, timing out if necessary.

The kernel may also provide other additional "utility" calls in support of capability sealing. We will not cover them in detail; however, they might be of importance in a real implementation. Such calls are: a call to remove information sealed in a capability when it is no longer needed, thus reducing the size of sealed capabilities; and a call to review and change the information sealed in the capability. One can imagine these additional calls being used, for example, to seal temporal information in the capability which can be removed or changed when the time period no longer applies. In the next section, we discuss how the user-level subsystem interfaces with the kernel for the important aspect of capability unsealing.

4.4.2 User Level Support

The user space part of the Storage Subsystem (SS), which we shall henceforth call simply the SS, performs all operations on capabilities to be sealed or unsealed except for those performed by the kernel. Specifically, SS performs all review and manipulation of information sealed in capabilities. SS permits a

user to seal (or reseal) any capability for an information storage object, or to review information which has been sealed in a capability. Sealed information may also be removed or changed via direct kernel calls. In this section, however, we will be concerned only with the unsealing mechanism, since it is the most crucial to an understanding of the basic security mechanism.

The following program fragment is meant to illustrate how the SS *might* be encoded to interface with the equivalent kernel mechanism. To recap briefly, when a capability is to be unsealed, the kernel gathers all information sealed in the capability, puts them in a kernel-defined object, and signals the user-level subsystem that an unsealing operation is required. The user-level subsystem must now retrieve this information, perform the necessary unsealing (recursively, if necessary) and return a capability to the kernel, with access rights adjusted based on the unsealing operation, to be used in the operation in which the kernel encountered the sealed capability. Here now is this program fragment; it is written in a psuedo-programming language -- the reader should find no difficulty in following the logic, perhaps helped by the copious comments. In order for things to work, it is assumed that the SYS_SEAL object passed from the kernel is organized as shown in Fig. 4.2.

Program 4.1: SS_Unseal

```
/* This program is invoked via an interrupt from the kernel
to a server process running on behalf of the SS. */

/* Get number of items in sealed capability list. This is 2
less than the number of capabilities in the capability list
of SYS_SEAL passed from the kernel. */

no_of_items = get_clist_length(sys_seal) - 2
```

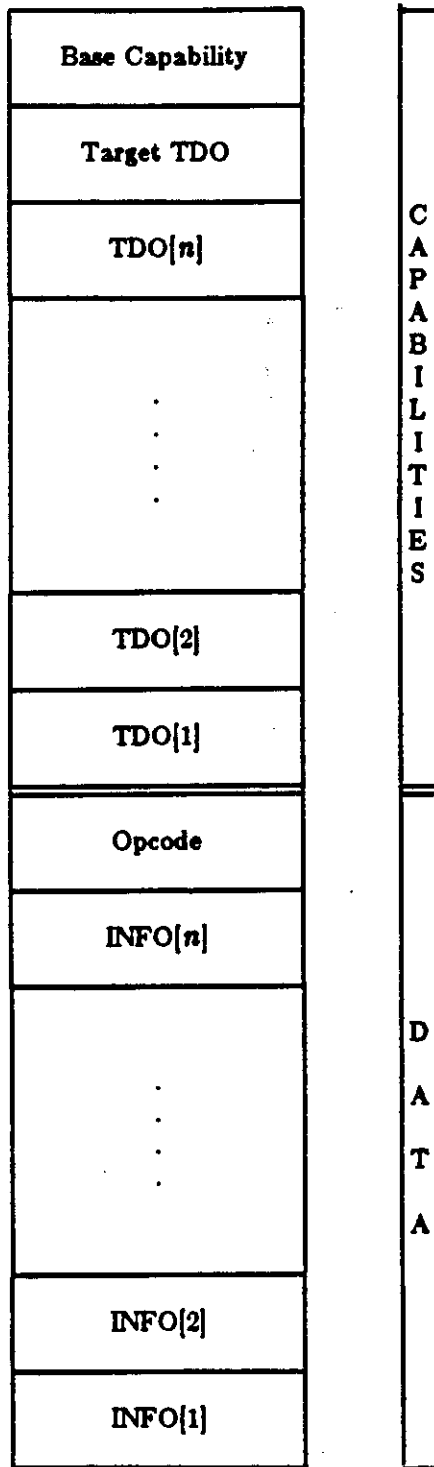


Figure 4.2: Structure of the SYS_SEAL Object

```

/* Get the capability for the type object from slot 2 */
tdo_capa = get_capa(sys_seal,2)

/* Compare with each TDO capability in LIST[C_T] */

slot_index = 0
for i=1 to no_of_items do
begin
  if(compare_capa(tdo_capa, get_capa(sys_seal, i+2)))
  then begin
    slot_index = i+1
    break
  end
end
if (slot_index == 0) return(s_abort(sys_seal)) /* tell the
kernel to deny the request. No matching TDO. */

/* If we get here, we have a matching TDO capability */

/* Start recursive unsealing */

for i = slot_index to no_of_items do
begin
  info[] = get_data(sys_seal,slot_index) /* get data
indexed by slot number */
  tdo_capa = get_capa(sys_seal,i) /* get the TDO
capability which sealed this information */
  op_code = get_data(sys_seal,1) /* get the operation
code from the first slot in the data part */
  new_info[] = check_info(tdo_capa,info[],op_code,status)

/* check_info is a type call to the abstract data type used
to seal the information in the capability. It returns new
information, if any, in new_info[], and status in the
"status" variable. This status indicates to SS how the
rights in the capability should be modified */

  check_status(status) /* checks the status returned
from check_info */
  update() /* updates the base capability */
end

/* return the modified base capability in slot 1 of the
sys_seal object. The kernel will destroy sys_seal */

return(put_capa(sys_seal,base_capa, 1))

/* End of program */

```

`get_clist_length()`, `get_capa()`, `put_capa()`, `get_data()`, and `compare()` are kernel calls that manipulate capabilities, as discussed in section 4.4.1.

4.5 Summary

In this chapter we have presented a mechanism whereby users can implement their own security policies based on the concept of Capability Sealing. The fact that the type of information that can be sealed in a capability is unconstrained, gives great flexibility to the mechanism; this flexibility is often very useful in the context of information storage. The mechanism permits a wide range of revocation strategies to be implemented, perhaps the most important of which is the possibility of *selective revocation* whereby a sealer of a capability can selectively revoke rights based on the usage of the capability. Although not explicitly discussed, the mechanism also permits capability review since it is possible for a user to store information about who has used, or is about to use, a capability, or even who has outstanding capabilities for an owned object -- this is made possible by the abstract type mechanisms of the object model.

CHAPTER 5

AVAILABILITY ISSUES

The availability of information in a distributed processing system is impacted by the system's architecture, as well as the reliability of its components. In this chapter, we study the effect of system architecture and replication strategies on availability for a general class of distributed systems. We start by studying a distributed system in our own operating environment consisting of a number of VAX minicomputers connected in a local network, and running the distributed operating system LOCUS. We then extend this study to bus networks, and finally to general multi-connected topologies. Our primary goal is to get a feeling for the level of replication required in achieving any given level of availability. We apply the results of this study to the implementation of FTSS in a later chapter.

5.1 Conceptual Framework

Availability is the long-term probability that a system is successful in providing a specified level of service (completing an event under a constraint), given that it was operating correctly at some prior "reference" time. Information availability is specifically concerned with the probability that a specified piece of information is available at some future time.

In distributed storage systems, the tendency is to replicate information at different storage sites so that a site failure will not render it immediately inaccessible. This procedure often leads to difficulty in keeping the replicates mutually consistent when the network becomes partitioned [Park82].

In this discussion, we view a distributed processing system to be a collection of computational nodes (or sites) interconnected by a relatively high-speed communication subsystem facilitating information transfer among the nodes. A computational node consists of a processing unit (or several such units operating either independently or collectively), and to which is attached a storage subsystem consisting of one or more storage devices. In our environment, the processing unit is a VAX, and the storage devices are one or more high-capacity winchester SMD disk drives. We refer to a computational node as a *storage site*. Such a system is diagrammed in Fig. 1.1, which we repeat here in Fig. 5.1 for completeness. This type of system architecture characterizes a *Local Computer Network* (LCN) [Clar78].

From Fig. 5.1 it may be deduced that a storage site is a series (in the probabilistic sense) connection of processors and storage devices, each being viewed as an independent subsystem. This observation leads to an important result. It is clear that the availability of any information at the storage site is directly proportional to the series reliability of processors and storage devices (assuming a simplex configuration). Put another way, information becomes unavailable if either the processor subsystem fails (for any reason), or the component of the storage system in which the information is stored fails. We assume for the moment that repair of the affected component is not instantaneous or immediate.

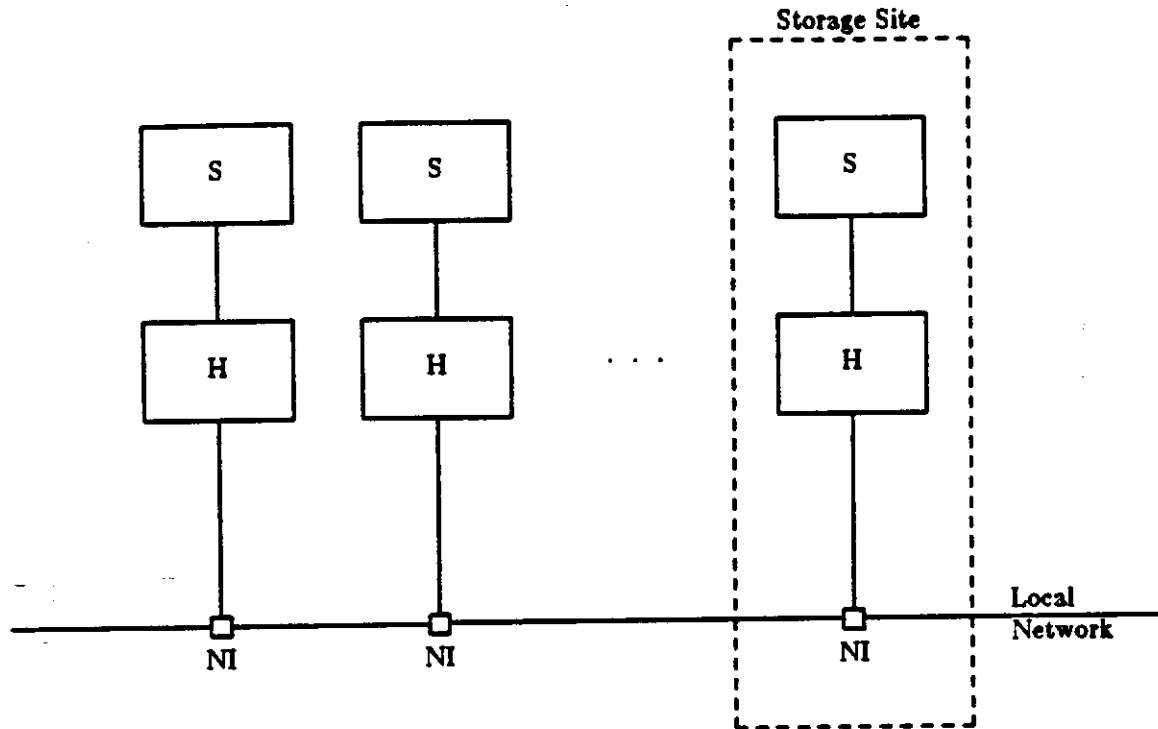


Figure 5.1: Distributed System of Storage Sites

5.2 Hardware Reliability versus Information Availability

We make the optimistic assumption that in a LCN, information is considered available as long as it is possible to access any single copy of the (possibly replicated) information, regardless of the storage site at which the information is stored. In view of our distributed system model, this is true if at least one processor in the *Storage Set*¹ is operational and the storage device on which the information is stored at that storage site is also operational.

¹We define a *Storage Set* to be the set of storage sites at which information is stored in a distributed processing system.

5.3 Approaches to Achieving High Availability

Several approaches have been used for achieving high information availability in distributed processing systems. The first approach is what might generally be called the "high reliability" approach. The objective is to make the system hardware and software as reliable as possible either by adding some form of redundancy to the storage site (fault-tolerance), or by choosing components which are extremely reliable (fault-avoidance). Information is not replicated, and the availability is the same as the reliability of the storage site. The fault-avoidance scenario requires that the hardware and software components of the storage site be non-redundant and ultra-reliable; in general, this is a costly approach (cost being defined in terms of the manufacturing and development costs) since it is expensive to create non-redundant highly reliable hardware and software. Furthermore, the system fails when any component fails, directly leading to unavailability. The fault-tolerance scenario replicates key hardware and software components so that failures can either be masked (static fault-tolerance) or detected and corrective action taken (dynamic fault-tolerance).

The second approach attempts to overcome some of the limitations of the first approach by replicating information at a single storage site. Thus, there is a higher probability that in the face of hardware and/or software faults, at least one copy of the required information will be available. In this scenario the hardware and software of the storage site may satisfy the requirements of the first approach, or they may be simplex, i.e., non-redundant. In either case, information is replicated either on multiple storage devices or on the same device, usually in such a way that each copy has an independent

probability of failure barring catastrophic (non-recoverable) failures. We note that in the serial arrangement of Fig. 5.1 the availability of information is still heavily dependent upon the reliability of the processing subsystem, since if this subsystem should fail, information stored in the storage subsystem would become unavailable despite the fact that it is replicated. Therefore, the advantage of this approach is that less reliable components can be used in implementing the storage subsystem (equating with lower costs) while achieving a given level of availability. The disadvantage is that the processing subsystem must still be highly reliable in order to attain availability goals. Another disadvantage is the need to maintain mutual consistency between the multiple copies of the information.

The third approach replicates information, but instead of storing multiple copies at a single storage site, each replicate is stored at a different storage site. The idea is to increase availability by making sure that no single (or perhaps multiple) site failure will result in unavailability, as long as it is physically possible to access the needed information from the site at which it is stored. The primary advantage of this approach is that less reliable and therefore less costly components of processor and storage subsystems can be used in attaining a given level of availability. The primary disadvantage is the need to maintain mutual consistency among the copies, particularly in the event of partitions in the network. We point out here that in some distributed systems, it is not possible to access copies of information at other sites, especially when the sites are geographically far apart, or when there is some other physical barrier that prevents access. In these situations, the only advantage of replication is to permit "local" updates which must later be reconciled with the other copies for

consistency when the network merges or, alternatively, to increase performance by having a local copy of the information on hand. We will not assume the latter need for replication; instead we assume that replication is a means to increased availability.

5.4 An Information Availability Model

In this section we develop a model which permits us to derive quantitative bounds on availability. The data for this model was obtained from a study of failure logs of our local distributed system over a 6-month time period.

Costes et al. [Cost78], developed a stochastic availability model for maintained systems featuring hardware failures and design faults. They were able to show both the transient and long-term effects on availability as a function of hardware and software reliability. They did not consider periodic maintenance, but assumed that repair would be performed on-demand until the system was repaired. They were not considering a distributed processing environment although they did take hardware redundancy into account. This was an extension of the work by [Lapr75] who considered the reliability and availability of repairable structures.

Makam and Avizienis [Maka81] considered the reliability modeling of repairable structures with and without periodic maintenance. They were able to derive computational procedures for characterizing not only reliability, but other life-cycle measures including availability. The results are applicable to a wide-range of fault-tolerant systems.

Raghavendra and Makam [Maka83] investigated the dynamic reliability modeling of computer networks using a boolean algebraic approach. They show that it is possible to obtain useful life-cycle measures in a computationally tractable way without resorting to Markovian analysis as is normally done for these cases.

All of the above techniques are applicable in our model, and we make use of them where possible. Our goal is to obtain a deeper understanding of availability in distributed processing systems, rather than inventing new computational procedures.

5.4.1 Model Definition

We strive to make our model accurately reflect the operational behavior of information in a distributed processing system. We view the system as operating in two sets of states: operational states and failure states or, more precisely, available states and unavailable states (with respect to information). The available states have a one-to-one correspondence with the operational sites of the network that belong to the storage set for a given piece of information. The unavailable states would then correspond with those sites in the storage set from which information is unavailable. Thus, the set of available and unavailable states represent but a subset of the states (or sites) of the network; viz. those sites at which the information is stored. We assume that there are no dependencies between sites that belong to a storage set and those that do not. This allows us to reduce the state set of our model significantly while still maintaining reasonable realism.

We make the following assumptions in structuring our model: First, information in a storage subsystem is semi-permanent and is never lost. The characteristics of contemporary storage devices such as disks and tapes makes this a reasonable assumption; the use of error-checking and correcting codes, off-line backup, and so on makes this so. Therefore, information may be unavailable over some time period, but it will become available again at some later time. Second, the hardware and software failure rates are Poisson processes with constant failure rates. Third, the ratio of failure time to repair time is very large; that is, the mean repair rate is much less than the mean failure rate. Fourth, the network is repaired on both an on-demand and a periodic basis. These assumptions allow us to model the system as a semi-markov renewal process with complete regeneration at each renewal epoch. The mathematical principles involved are well known (see, for example, [Heym82]).

Fig. 5.2 shows the behavior graph or the markov state diagram for the system. The operational states of the model are indexed by a parameter (a positive integer) representing the number of operational sites in the storage set from the viewpoint of, say, an application program. For example, the state indexed with the parameter $(n-2)$ indicates that of n storage sites containing a copy of the information, $n-2$ are operational (that is, the information at two of the n sites is unavailable due to an operating system crash, hardware failure, or both).

We choose to lump all the unavailable states into a single *fail* state no matter what the ultimate cause of the unavailability. The transition between the states characterize failure and repair operations in the following way:

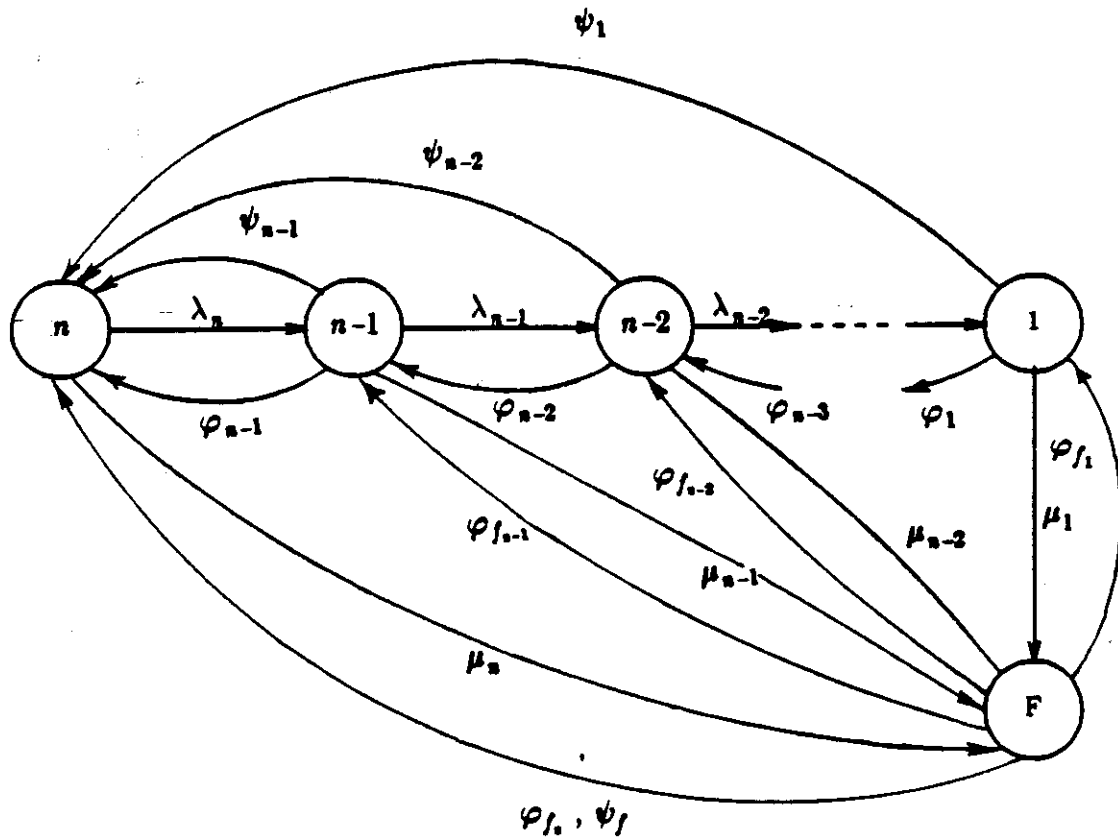


Figure 5.2: Markov State Diagram of Information Availability

- $\lambda_i =$ Mean failure rate of a storage site, and is due to both hardware and software failures.
- $\mu_i =$ Mean failure rate resulting in total loss of availability from site i . This is a function of whether or not the application has been or has to be aborted due to the inability to continue the computation because the information at this site is unavailable.
- $\varphi_i =$ Mean on-demand repair rate at site i . This would characterize, for example, the mean time to reboot the operating system at site i after a crash.
- $\psi_i =$ Mean periodic repair rate from site i resulting in full regeneration of the number of sites in the storage set for the application program.

We emphasize once again that we are considering *two* types of repair policies: one due to on-demand maintenance, and the other due to periodic maintenance. We feel that the inclusion of both policies in our model results in a closer approximation to the operational characteristics of contemporary distributed processing systems.

5.4.1.1 Network System Operation

We perceive the LCN operating in a manner illustrated by the time diagram in Fig. 5.3. We break up the time axis into periodic *service intervals*. The system begins operation at t_0 in a full-configuration state (all storage sites in the storage set available), and operates up to time t_1 where the first service interval begins. During this time $(t_1 - t_0)$, on-demand maintenance on

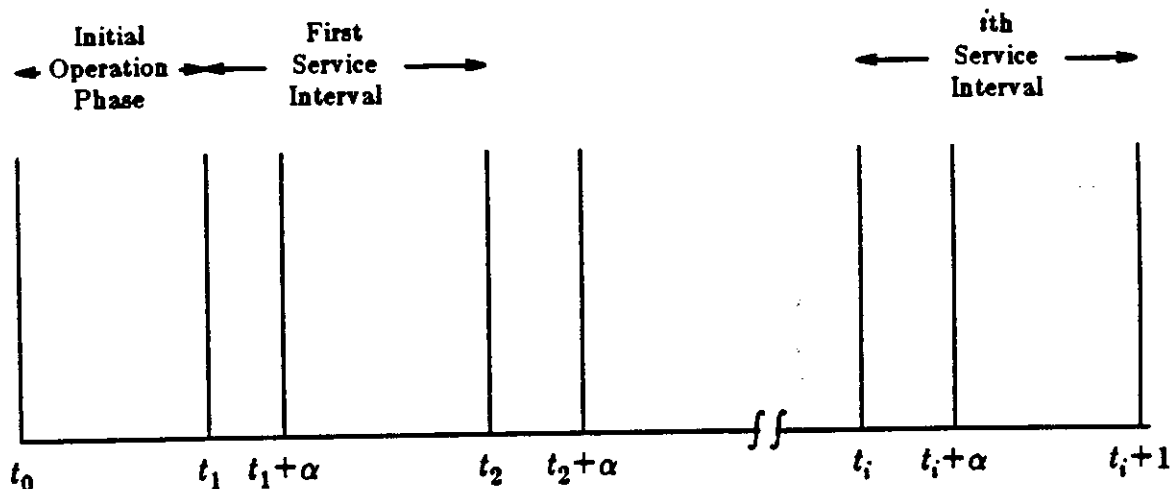


Figure 5.3: Time Diagram Showing Operation of the LCN

hardware and/or software is permitted. We assume that on-demand maintenance will not always be successful and some types of repairs will have to be left for periodic maintenance which occurs at the start of the next service interval. During this service interval the system may move about the states of the markov model, except along the renewal transitions identified by ψ_i .

At the start of each service interval, the system undergoes total regeneration (which we assume is always successful). The time to accomplish this is identified by the random variable α . We assume that any failures that occur during this *renewal interval* (indicated by the shaded area at the start of each service interval) will be taken care of as a part of the system regeneration (if a failure occurs while the service technician is on hand then it will be fixed immediately).

5.4.1.2 Hardware and Software Considerations

Hardware and software are characterized by stochastic processes representing failures and repairs. We will assume that, as far as system reliability is concerned, hardware and software failure mechanisms are independent and can therefore be modeled as a series decomposition of failure probabilities [Cost78]. We further assume that software is not fault-tolerant and any failures due to software must be fixed either by on-demand maintenance (reboot or reprogram) or periodic maintenance.

We model on-demand hardware and software maintenance in a similar way to closed fault-tolerant systems with repair capability [Maka82a]. In such systems, failed modules undergo repair in some repair facility and are brought back into service when the repairs are completed; it is therefore possible to compute life-cycle measures such as availability since the system can recover from module failures. If the hardware or software fails and on-demand maintenance is ineffective, then the system becomes unavailable until the next renewal epoch.

5.4.2 Model Development

We use the computational procedures developed by [Maka82b] for the analysis of fault-tolerant systems. In particular, we will use those procedures developed for the reliability and life-cycle analysis of *Periodically-Renewed Closed Fault-Tolerant Systems*, or PRC Systems for short, and repairable systems. The details of the mathematical analysis can be found in [Maka81] and [Maka82b].

In order to use the PRC methodology, we take a two-phased modeling approach to the solution of the availability $A(t)$ for a distributed system. In the first phase, we model the hardware and software at each storage site as a repairable system. In the second phase, we model information availability directly for the distributed processing system as a periodically-renewed closed fault-tolerant system with appropriate model parameters obtained from the first phase. In the case where coverage is perfect [Bour69], this decomposition adds no errors to the model. When coverage is imperfect, a small error is introduced depending upon the extent of coverage.

5.4.3 Reliability Estimation for Storage Sites

In the first phase the ARIES [Maka82b] reliability modeling tool is used to establish a relationship between subsystem failure rates and subsystem availability. It permits us to estimate reliability, availability, and other life-cycle measures for the processor and storage subsystems at a storage site.

5.4.3.1 The Processor Subsystem

We model the processor subsystem at a storage site as a homogeneous continuous-time finite-state markov process. The basic state diagram is shown in Fig. 5.4. We choose not to differentiate between hardware and software failure and repair processes in our model; that is, we are not concerned about which event leads to processor failure since our primary goal is information availability at the storage site. In the model, the state labeled A represents the "normal" operating state of the processor subsystem; in this state we assume that the system is operating in its prespecified manner and there are no hardware or software failures that will lead to information unavailability. The

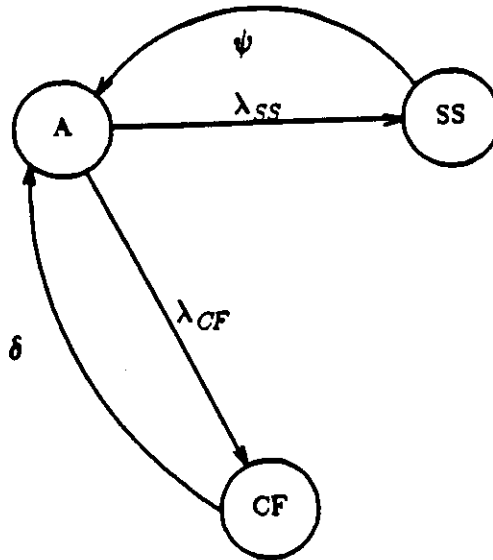


Figure 5.4: Markov State Diagram for the Processor Subsystem

states labeled *SS* and *CF* represent the safe-shutdown and crash-failure states respectively. The *SS* state represents the case where the processor subsystem is safely shutdown by system administrators for either hardware or software maintenance on an on-demand basis; transitions to this state are governed by a failure process characterized by the transition rate λ_{SS} , while transitions out of the state is governed by a repair process with constant transition rate ψ . The *CF* state represents information unavailability due to operating system crashes or hardware failures for which automatic system recovery/restart is provided; λ_{CF} represents the constant failure rate from the normal operating state, while δ represents the mean restart rate after a system crash. This simple model is sufficient to gain some insight into the importance of the processor subsystem at a storage site.

Fig. 5.5 shows a plot of reliability and availability for a processor subsystem. λ_{SS} and λ_{CF} represent the number of failures per time unit respectively, ψ the number of repairs that can be accomplished per time unit, δ the average

Reliability and Availability for the Processor Subsystem

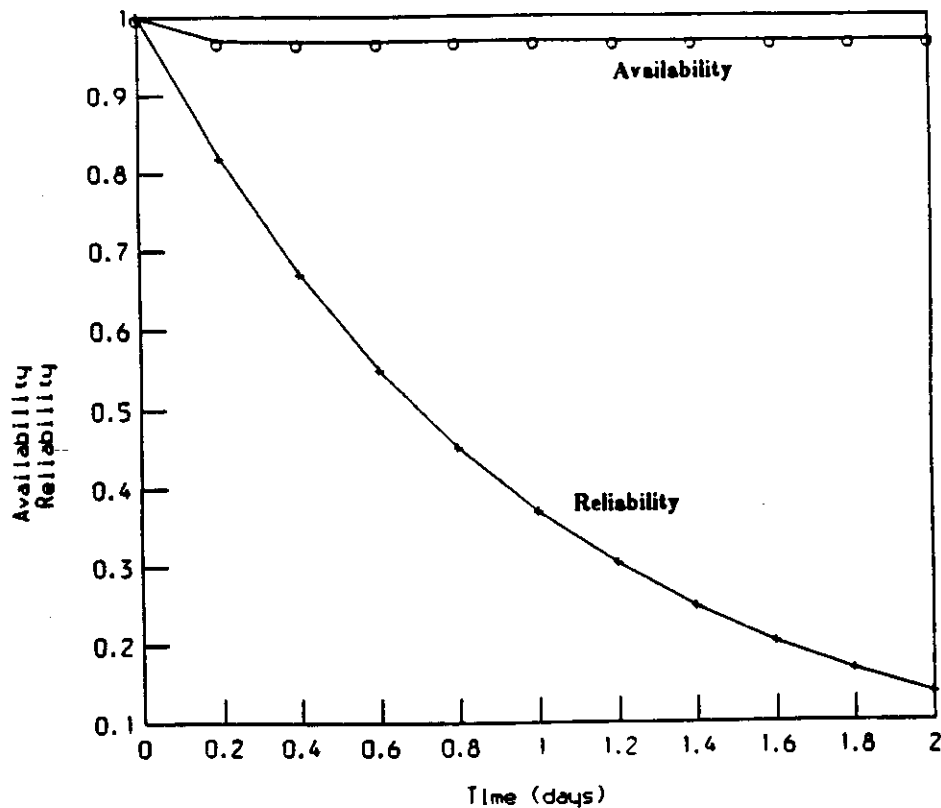


Figure 5.5: Plot of Reliability and Availability for the Processor Subsystem

number of restarts that can be performed per time unit, and m the number of “repairmen” that are available to perform repairs when necessary. The time unit is an arbitrary value that can be interpreted as hours, days, weeks, millions of hours, etc., as a function of the processing environment. In our studies, our time unit was based on a “day” (24 hours). Table 5.1 shows the ARIES parameters used in the model. In Fig. 5.6 we show the effect of crash coverage¹ (varying $CY[1]$) on information availability.

¹*Crash Coverage* is the percentage of failures that lead to a safe system shutdown.

Notation	Description	Value
Type	Subsystem Type	5
D	Number of Degradations	0
S	Initial Number of Spare Modules	0
CS	Coverage for recovery from a spare module failure	1
λ	Failure rate of one active module	1
μ	Failure rate of one spare module	0
$\underline{Y} \equiv (Y[0], \dots, Y[D+1])$	Active Resource Vector	$Y[0]=1, Y[1]=0$
$\underline{CY} \equiv (CY[0], \dots, CY[D+1])$	Coverage Vector for Active Modules	$CY[0]=1, CY[1]=0.2$
M	Number of repair facilities	1
ψ	Repair rate per module	12
δ	Restart rate from the CF state	48
ν	Failure rate of one good module in the SS state	.001

Table 5.1: ARIES Parameters for the Processor Subsystem

Effect of Crash Coverage on Availability

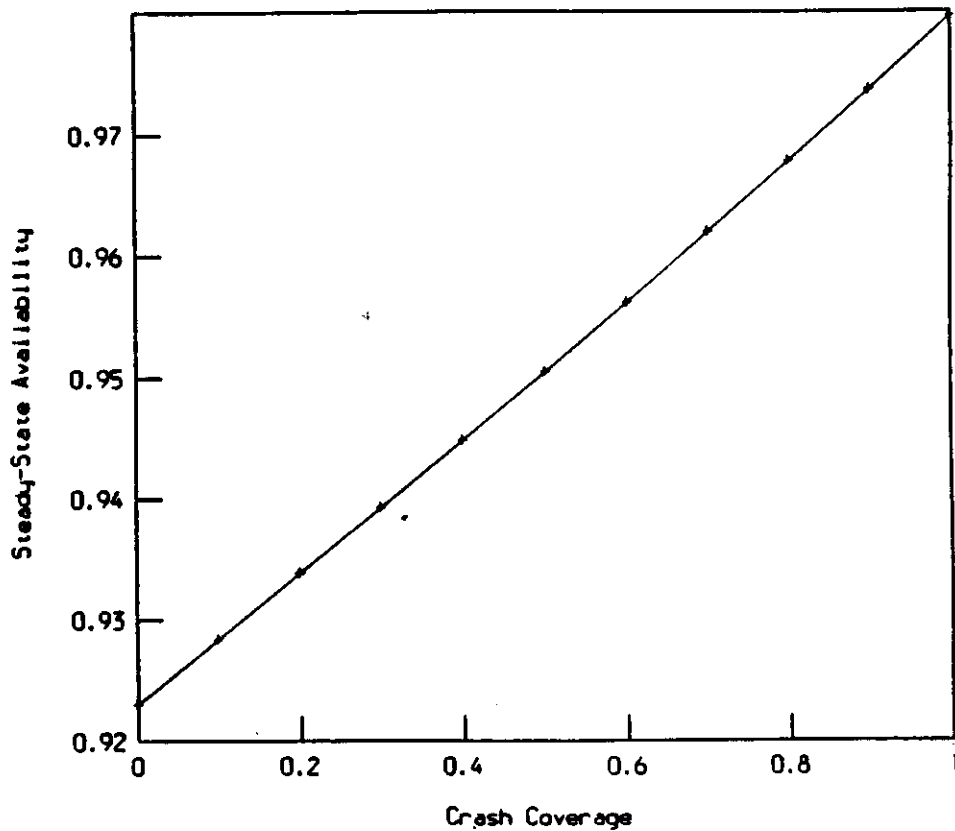


Figure 5.6: Effect of Crash Coverage on Availability

5.4.3.2 The Storage Subsystem

We model the storage subsystem in an analogous way to the processor subsystem, in the sense that the markov state diagram remains the same. The interpretation of the parameters is the same. A transition to the *SS* state for the storage subsystem occurs when a failure event transpires that results in safe shutdown of the subsystem; we assume that in this situation data stored in the subsystem is preserved and is brought back on-line subject to the repair rate ψ (which we assume to be constant). A transition to the *CF* state occurs when the subsystem fails in such a way that data is lost and must presumably be restored from off-line devices such as tapes; the transition out of the state is characterized by the parameter δ representing an average restart rate (this rate will, in general, be slower than ψ if we pessimistically assume that a service technician must be consulted to perform the necessary repairs).

In Fig. 5.7 we show the reliability and availability for a typical storage subsystem using parameters we have obtained from studying our local distributed processing system. The interpretation of the parameters is similar to that for the processor subsystem presented in the previous section. The parameters were obtained by assuming that all information is stored on a single storage device whose characteristics approximate those typically associated with high-capacity rotating magnetic disk assemblies. For example, the failure rate λ used in the model was obtained directly from published Mean Time Between Failure (MTBF) data in the manufacturer's data sheet for the disk subsystem used in our distributed processing environment. As with the other subsystems, we have assumed a basic time unit of one day. The ARIES parameters of interest are shown in Table 5.2.

Reliability and Availability for the Storage Subsystem

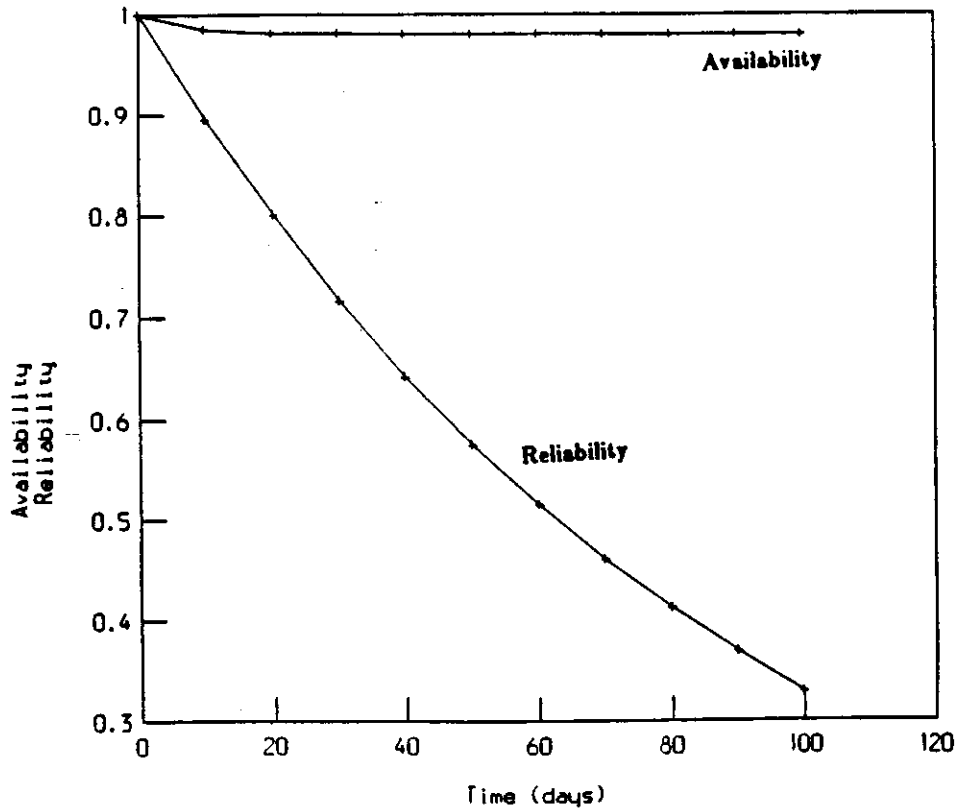


Figure 5.7: Plot of Reliability and Availability for the Storage Subsystem

Notation	Description	Value
Type	Subsystem Type	5
D	Number of Degradations	0
S	Initial Number of Spare Modules	0
CS	Coverage for recovery from a spare module failure	1
λ	Failure rate of one active module	0.011111
μ	Failure rate of one spare module	0
$\underline{Y} \equiv (Y[0], \dots, Y[D+1])$	Active Resource Vector	$Y[0]=1, Y[1]=0$
$\underline{CY} \equiv (CY[0], \dots, CY[D+1])$	Coverage Vector for Active Modules	$CY[0]=1, CY[1]=0.25$
M	Number of repair facilities	1
ψ	Repair rate per module	24
δ	Restart rate from the CF state	0.0143
ν	Failure rate of one good module in the SS state	.001

Table 5.2: ARIES Parameters for the Storage Subsystem

5.4.3.3 The Storage Site Subsystem

We now combine the models above (in series) to form a single subsystem and investigate the reliability and availability under the various failure and repair rate assumptions. Once again we use the ARIES modeling tool to obtain results of interest.

Fig. 5.8 shows the time-dependent reliability and availability for a storage site using the "baseline" processor and storage subsystems presented previously...

5.4.4 Network Availability

There are several ways in which one can address the subject of Network Availability. However, before we discuss them, we wish to reiterate that we are considering availability only from the point of view of being able to access information which has previously been stored in a secondary storage subsystem. We are also careful to point out that unavailability should not be confused with unrecoverability, although the latter certainly has some probability of occurring (no matter how small); we have assumed in our prior analysis that once information is stored in such a subsystem, it will not be lost. Finally, when we speak of network availability, we only do so with respect to a given application program; from this viewpoint we are therefore not considering all possible storage sites in the network, rather, only those that participate in the storage set of the application program.

Reliability and Availability for a Storage Site

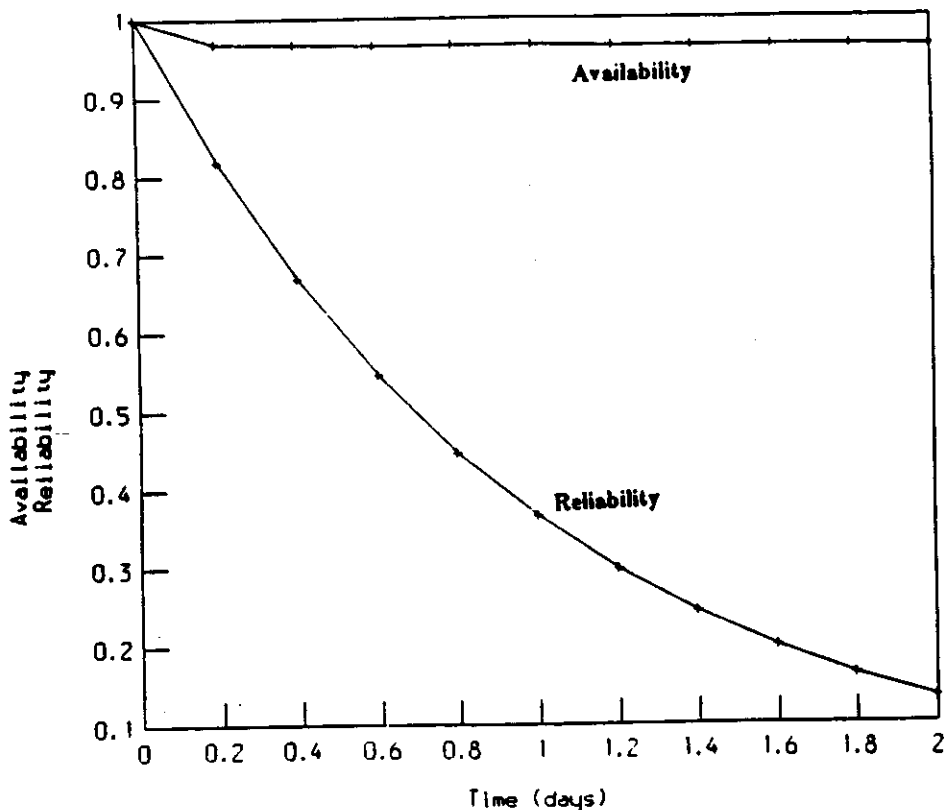


Figure 5.8: Plot of Reliability and Availability for a Storage Site

5.4.4.1 Simple Topologies

The simplest way to approach network availability is to assume a very simple interconnection structure for the network so that topological issues do not have to be taken into account, then assume that all storage sites in the storage set have identical configurations of hardware and software as well as the same failure and repair characteristics. The type of assumption we are making here is characterized by a class of network architectures often referred to as *Bus Topologies* [Cham80]. We assume, for simplicity, that the bus in

these topologies is perfectly reliable (it is easy to account for the reliability/availability of the bus since a failure here generally leads to an n -fold partition of the system, where n is the number of operational sites in the storage set). Under these assumptions, it is possible to represent the availability of the information as a simple polynomial in $A(t)$, where $A(t)$ is the time-dependent availability of a storage site. For example, given a collection of storage sites, the *Network Availability* $A_{n,m}(t)$ can be expressed as:

$$A_{n,m}(t) = \sum_{i=m}^n \binom{n}{i-m} (1-A(t))^{i-m} A(t)^{n-i+m}, \quad i \geq m \quad (5.1)$$

where n is the cardinality of the storage set and m is the minimum number of sites that must be available in order for the network to be considered available. If we assume that the arrival of failures is governed by a Poisson failure process and the repair times are exponentially distributed, then the time-dependent availability can be expressed as:

$$A(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t} \quad (5.2)$$

where λ and μ are failure and repair rates, respectively. Table 5.3 shows the time-dependent availabilities for a network application program whose storage set has a cardinality of 4 and using the computed availabilities for a baseline storage site from section 5.4.3.3.

It is interesting to note from the figure the decrease in the availability as the dependencies¹ among the sites increase. Availability is highest (as we would suspect) when there are no site dependencies. It is also interesting to note the

¹Here, dependency occurs if it is required that more than one copy of the information exist at different storage sites.

Time	Baseline Availability $A(t)$	$A_{4,1}(t)$	$A_{4,2}(t)$	$A_{4,3}(t)$	$A_{4,4}(t)$
0	1	1	1	1	1
0.1	0.971926	0.999999379	0.99991336	0.99544625	0.892345
0.2	0.968247	0.999998983	0.99987499	0.99420355	0.87891
0.3	0.967021	0.999998817	0.99986	0.993758	0.874467
0.4	0.966481	0.999998737	0.9998531	0.9935563	0.872516
0.5	0.966145	0.999998686	0.9998487	0.9934295	0.8713
0.6	0.965872	0.999998643	0.999845	0.9933256	0.8703187
0.7	0.96562	0.9999986	0.9998416	0.993229	0.86941
0.8	0.965376	0.99999856	0.99983827	0.9931348	0.86853
0.9	0.965138	0.999998523	0.99983495	0.9930424	0.8676761
1.0	0.964903	0.999998483	0.99983162	0.9929505	0.8668314

Table 5.3: Time-Dependent Availability for a Bus Network

relatively high levels of availability that can be obtained through replication at independent storage sites, even when the storage site by itself is not particularly available. It is not a panacea however, and we must be very careful about *over-replication* because of the attendant problems of consistency among the many replicates; furthermore, too much replication can actually lead to higher levels of *unavailability*. While the latter is easy to show analytically using the expressions above, we can do so through a very intuitive argument; for instance, the more copies of information that we have, the higher the probability that one copy will become unavailable - in such a situation, it may be necessary to avoid updates (but not read access) to all but a single of the remaining copies so that consistency can be restored later (the need to avoid updates is, effectively, unavailability).

Equation 5.1 is quite general and can be used, for example, to determine what level of site availability and replication factor is necessary in order to meet a given network availability goal. It should be kept in mind, however, that this is merely an approximation to the real state of affairs in contemporary networks. Our point is to provide some quantitative basis on which to discuss

forthcoming events, and not to model the "real world" exactly.

5.4.4.2 Periodically Renewed Networks

We may also discuss network availability in terms of the availability of storage sites as well as the fact that the network receives periodic maintenance. We alluded in section 5.4.1 to this fact when we mentioned that in "real" networks it is common to have two types of repair policies -- one due to on-demand and the other to periodic maintenance. So far we have modeled our storage sites using strictly on-demand maintenance; we now extend the repair policy at the network level by considering that a service technician arrives periodically and will restore the hardware and software of the entire network to a working state (we are aware that in cases where storage sites are geographically distant this may not be possible in a single service interval; we simply assume then that there is a repairman at each site which works in $1/n$ th the time of a single repairman for an n -site network).

Before we are able to apply the PRC methodology, we are forced to revise our model of section 5.4.1 to be compatible with the ARIES modeling tool. This revised model is shown in Fig. 5.9. We have renamed the "fail" state as a "safe shutdown" state since this more closely reflects the *modus operandi* of contemporary networks (recall that we are dealing with a storage set which is but a subset of the sites of the network; unavailability in the storage set does not necessarily imply that the network has failed). We have also removed the transitions indicated by the parameter φ ; under the assumption of periodic renewal (once a site becomes unavailable, it remains so until the next renewal epoch); however we compensate for this later in model execution by adjusting

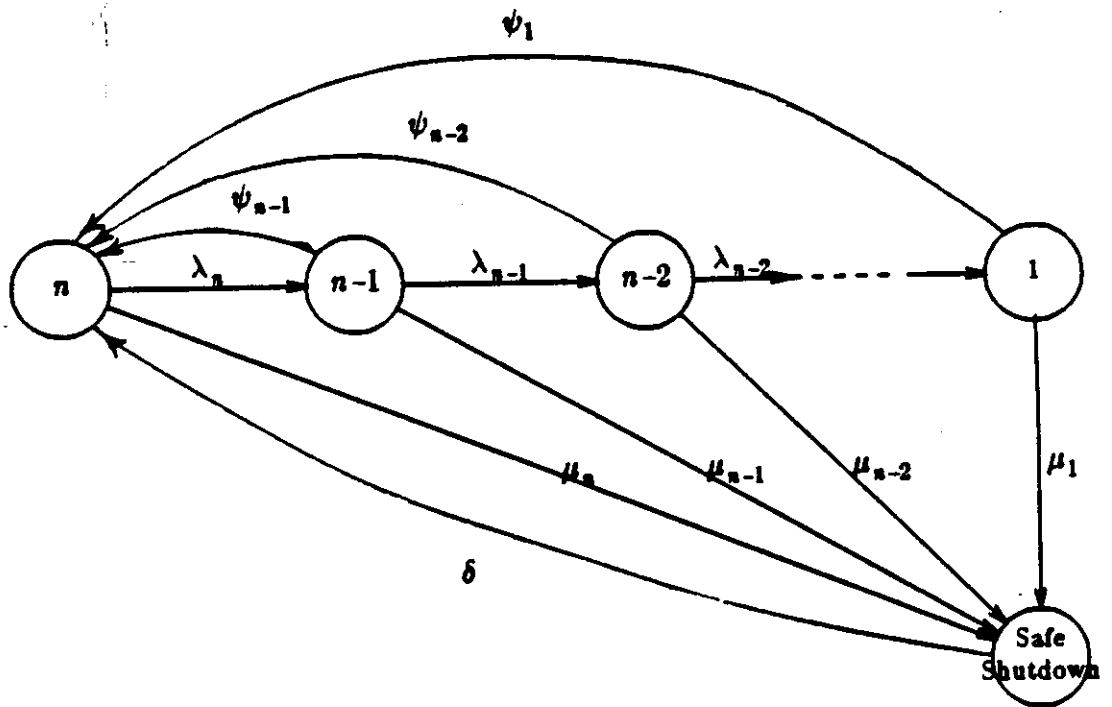


Figure 5.9: Revised PRC Model

the site failure rates so that the probability of a site failing during a *service interval* is about the same as if it could be repaired and put back into service as the parameter φ_i was meant to indicate.

Before we discuss our model results, we would like to say something about the model parameters we have used. First, we are modeling what we consider to be our *baseline* system; this is simply a network representing a storage set of baseline storage sites. We have assumed a service interval length of 60 time units and a mean system checkout time β of .0417 time units (if the time unit is days, this represents about 1 hour). There is a single repairman (or service technician) who can service 12 storage sites per time unit. The system operates in various configurations which we will detail later, but always with a failure rate of 0.00087283 failures per time unit.¹ This allows us to find the availability of the network in terms of the availability of the storage sites. While this decomposition does not produce exact results for availability, we find that they are acceptable (and realistic) in this discourse.

In Table 5.4 we show the effect of information replication on the network availability. The table shows the steady-state availability that is achievable depending upon the number of sites in the storage set. Once again we see that very high levels of availability can be attained through replication. However, this is tempered somewhat both by the difficulties of maintaining consistency as well as the storage costs for the $n-1$ replicates. We see from the figure that unless one is seeking 100% availability, it is perhaps not necessary

¹This figure was obtained by treating the steady-state availability as the reliability of the system and then computing λ from the expression $R=e^{-\lambda t}$ when $t = 60$ time units (representing the reliability at the end of 1 service interval).

No. of Sites	Steady-State Availability
1	0.786904
2	0.940952
3	0.981965
4	0.994157
5	0.998032
6	0.999319
7	0.999759
8	0.999914
9	0.999969
10	0.999988

Table 5.4: Steady-State Availability versus Replication Factor

to have more than two or three replicates at different sites (under our assumed parametric constraints).

We have modeled the effect of site dependencies/failures on the availability of information using the PRC methodology by considering a 5-site storage set. That is, information is replicated at five sites. We investigate the effects of different failure partitions on overall steady-state availability under the assumption that if a site is operational, the information stored at that site is available; otherwise, it is unavailable. The results are shown in Fig. 5.10 where each curve indicates an N/M partition, N being the number of failed sites and M the number of operational sites. The ARIES parameters of interest used in the model is shown in Table 5.5. Table 5.6 shows some additional results of interest regarding the steady-state availabilities of the four cases and the amount of time lost due to safe shutdowns (we assume that all shutdowns are "safe" in the sense that a shutdown of one site does not affect the availability of the other sites). Again we find that relatively high levels of availability can be achieved even when only a few replicates are available; in the case of only two copies available in a 5-site storage set, over 99% availability is realized.

Effect of Site Dependencies on Network Availability

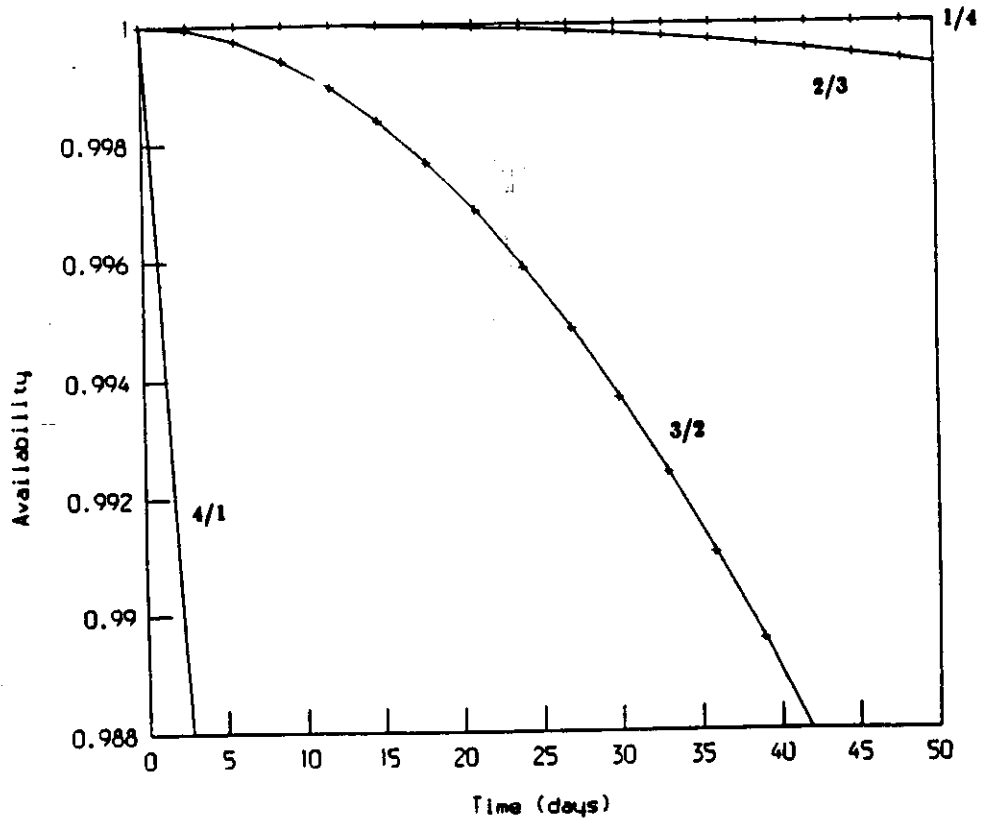


Figure 5.10: Effect of Site Dependencies on Network Availability

5.4.4.3 Complex Topologies

In the simple bus topologies we have assumed in the previous sections, it was assumed that the communication system (or bus) was perfectly reliable. In such networks, the notion of *network partitioning* has less significance than in a network in which the storage sites are arbitrarily connected. In fact, if the bus fails in a bus-topology network the network becomes m -partitioned, where m is the number of operational sites in the storage set (that is, network information availability is identical to information availability at a storage site). On the

Notation	Description	Value
Type	Subsystem Type	6
D	Number of Degradations	3
S	Initial Number of Spare Modules	0
CS	Coverage for recovery from a spare module failure	1
λ	Failure rate of one active module	0.00087283
μ	Failure rate of one spare module	0
$Y \equiv (Y[0], \dots, Y[D+1])$	Active Resource Vector	$Y[0]=5, Y[1]=4$ $Y[2]=3, Y[3]=2$
$CY \equiv (CY[0], \dots, CY[D+1])$	Coverage Vector for Active Modules	$CY[0]=1, CY[1]=1$ $CY[2]=1, CY[3]=1$
$Z \equiv (Z[0], \dots, Z[D+1])$	Computing Capacity Vector	$Z[0]=1, Z[1]=1$ $Z[2]=1, Z[3]=1$
M	Number of repair facilities	1
ψ	Repair rate per module	12
δ	Restart rate from the CF state	12
ν	Failure rate of one good module in the SS state	.0001
β	Periodic service interval	60
ω	Mean system checkout duration	0.0417

Table 5.5: ARIES Parameters for Site Dependencies

Partition	Steady-State Availability	Lost time due to safe shutdowns
1/4	0.999993	0.00040128
2/3	0.999682	0.0190772
3/2	0.991845	0.489312
4/1	0.879645	7.22131

Table 5.6: Steady-State Availabilities for Various PRC Partitions

other hand, in *multi-connected topologies* the network (from which the storage set is a subnetwork) may become partitioned in arbitrary ways. Therefore, in discussing information availability in multi-connected topologies, it is necessary to take the topology of the network into account, as well as the availability of the communication paths that interconnect the storage sites.

Consider Fig. 5.11, for example, which shows a typical multi-connected computer network. The information at storage site 5 is available to the proces-

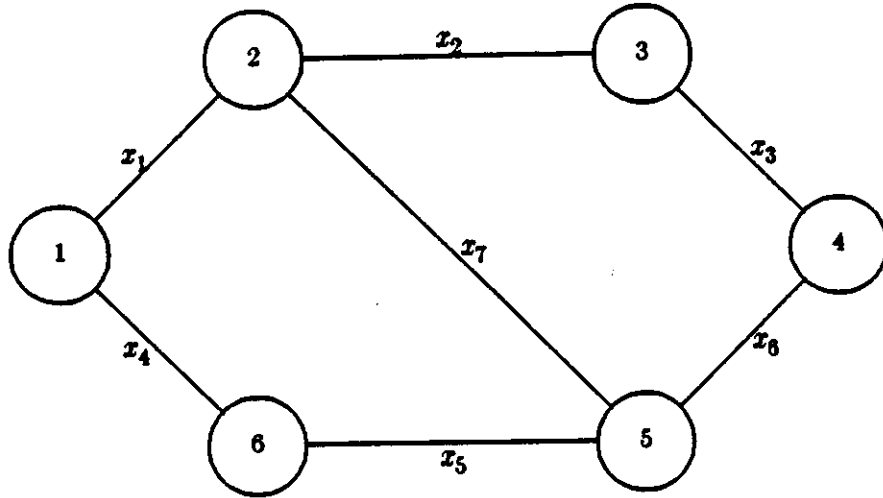


Figure 5.11: Multi-Connected Computer Network

sor at site 1 via any one of three possible (not necessarily independent) paths, namely, x_4x_5 , x_1x_7 , or $x_1x_2x_3x_6$. Therefore, the information availability with respect to all possible site pairs in the storage set must take the network topology into account.

A general procedure for computing network availability in terms of link availabilities is discussed in [Maka83]. We make use of this procedure but extend it slightly to include the availabilities of the storage sites. By treating the connected subset of sites that make up the storage set as a subnetwork, we are able to compute the information availability. Consider, for example, the simple bridge subnetwork of Fig. 5.12 which we consider to represent the storage set of an application program; in other words, there is one "master" copy of information which is replicated at three other sites interconnected as shown in the figure. We would like to get some feeling for the availability of this information in terms of link and node (storage site) availabilities. We make the pessimistic assumption that the availability reduces to the availability of a

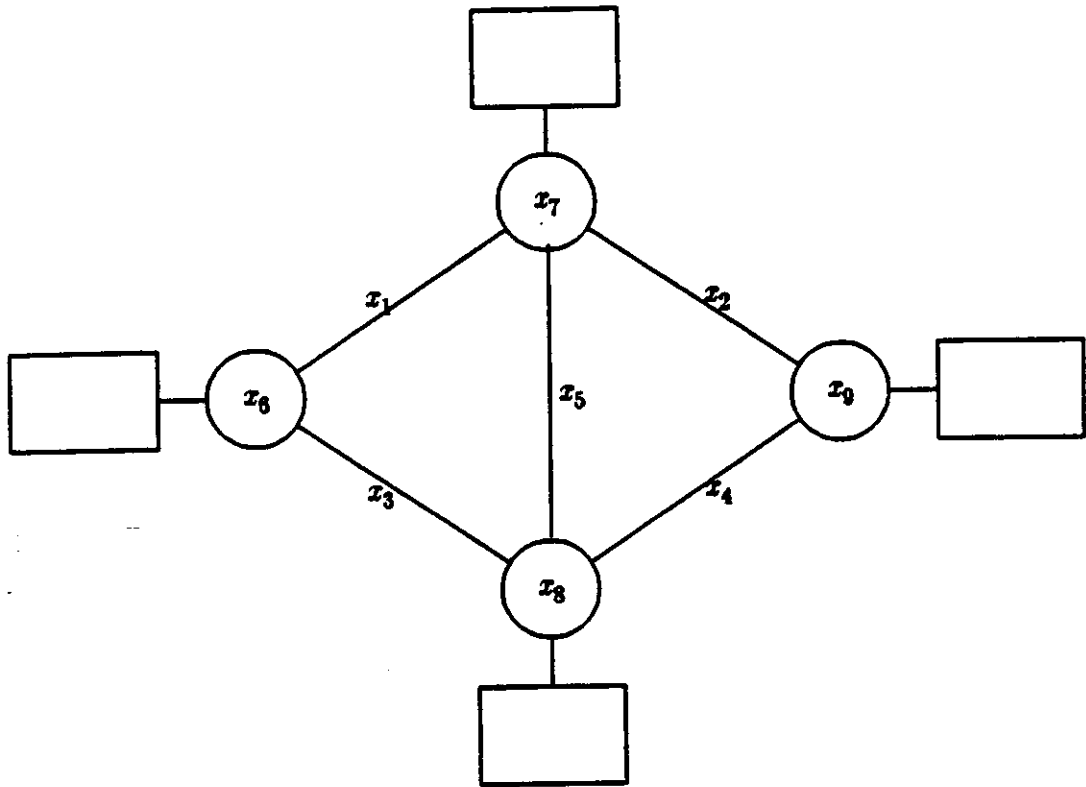


Figure 5.12: Bridge Subnetwork of an Application Program

single storage site if the subnetwork becomes partitioned.¹

To compute the subnetwork availability, we use the efficient algorithm proposed by [Grna81] as well as the procedure discussed in [Maka83] to obtain a symbolic expression for the multiterminal availability of the network. We do this by first computing the terminal-pair connectivity of every distinct node-pair in the network (there are 6 such pairs), then we take the logical AND of the resulting expressions (simplifying terms where possible) to obtain an expression for the multiterminal connectivity. Finally, we use the resulting

¹This assumption is justified if the nature of the information is such that it is impossible to maintain consistency if copies in separate partitions are independently updated. We therefore restrict updates to a single copy in *some* partition.

expression to obtain the multiterminal availability. The algebra involved is rather laborious, but equation 5.3 shows the symbolic expression for availability in terms of the link and node probabilities p_i where

$$p_i = Pr[\text{ith link or node is operational}]$$

$$\begin{aligned}
MC = & p_6 p_7 p_8 p_9 [p_1 p_2 p_3 (1-p_4)(1-p_5) + p_1 p_2 p_3 p_4 p_5 + p_1 p_2 p_3 p_4 (1-p_5) \\
& + p_1 p_2 p_5 (1-p_3)(1-p_4) + p_1 p_2 p_4 p_5 (1-p_3) + p_1 p_2 p_4 (1-p_3)(1-p_5) \\
& + p_1 p_3 p_4 p_5 (1-p_2) + p_1 p_3 p_4 (1-p_2)(1-p_5) + p_1 p_2 p_3 p_5 (1-p_4) \\
& + p_1 p_4 p_5 (1-p_2)(1-p_3) + p_1 p_3 p_4 p_5 (1-p_1) + p_3 p_4 p_5 (1-p_2)(1-p_1) \\
& + p_2 p_3 p_5 (1-p_1)(1-p_4) + p_2 p_3 p_4 (1-p_1)(1-p_5)] \tag{5.3}
\end{aligned}$$

where MC is the multiterminal connectivity. If we assume, for simplicity, that all links have the same failure and repair rates and likewise for nodes (the rates are different for links and nodes however), the time-dependent availability can be expressed in the following simplified form:

$$A(t) = b^4(4a^5 - 11a^4 + 8a^3) \tag{5.4}$$

where

$$a = \frac{\mu_1}{\lambda_1 + \mu_1} + \frac{\lambda_1}{\lambda_1 + \mu_1} e^{-(\lambda_1 + \mu_1)t} \tag{5.5}$$

and

$$b = \frac{\mu_2}{\lambda_2 + \mu_2} + \frac{\lambda_2}{\lambda_2 + \mu_2} e^{-(\lambda_2 + \mu_2)t} \tag{5.6}$$

where λ_1 and λ_2 are the failure rates of links and nodes respectively, and μ_1 and μ_2 are the repair rates of links and nodes respectively.

In Table 5.7 we show the relationship between network steady-state availability and the steady state availabilities for links and nodes.

Link Availability	Node Availability	Network Availability
1.0	0.99	0.960596
1.0	0.98	0.922368
1.0	0.97	0.885293
1.0	0.96	0.849347
1.0	0.95	0.841506
1.0	0.94	0.780749
1.0	0.93	0.748052
1.0	0.92	0.716393
1.0	0.91	0.68575
1.0	0.90	0.6561
0.99	1.0	0.999796
0.98	1.0	0.999169
0.97	1.0	0.998099
0.96	1.0	0.996567
0.95	1.0	0.994555
0.94	1.0	0.99205
0.93	1.0	0.989037
0.92	1.0	0.98551
0.91	1.0	0.98145
0.90	1.0	0.97686

Table 5.7: Availability for Multi-Connected Topology

It is interesting to note that the network availability is more sensitive to node availabilities than to link availabilities because of the redundancy of paths between node pairs.

5.5 Discussion of Results

There are a number of significant results from this analysis. First, the bundling of processor and storage, as is typical in most distributed systems, does not contribute to high availability, primarily due to the unreliability of the components. If processors and storage devices could be separated, higher information availability could be achieved. Second, high levels of availability can be achieved through replication, even with relatively unreliable com-

ponents. The analysis shows that it is not really necessary to have more than about two copies of any piece of information in order to get better than 98% availability. One must be careful, however, since we reiterate that this analysis makes assumptions which may not apply in all cases. Third, a periodically renewed system with information replication in sites with independent failure probabilities gives very high levels of availability under both periodic and on-demand maintenance. If the repair interval is reasonably short (depending on the probability of failure of sites), replication is not required at more than two sites. Finally, we observed that high availability can be achieved in a multi-connected topology if the reliability of the nodes can be kept high with respect to the link reliabilities; by storing information at multiple sites, there is a high probability that information will be available for access because of the redundant links.

5.6 Summary

In this chapter we investigated availability issues in distributed processing systems by analyzing various models representing a broad cross section of information storage strategies. The analysis was based, in part, on modeling tools developed in our own environment, and on data collected from our local distributed processing system over a six-month period. The significant conclusion of this study is that a periodically renewed multi-connected system with redundant interconnections between nodes and with on-demand maintenance, when necessary, provides very high levels of availability in the face of unreliable components by replicating information sparsely at various nodes. We will use these results in a later chapter when we propose a hardware architecture (FTSS) to support highly available information-storage systems.

CHAPTER 6

FAULT-TOLERANT STORAGE SYSTEM ARCHITECTURES

Contemporary distributed processing systems are requiring an increasing amount of on-line storage in order to meet the processing needs of several classes of users with varying storage requirements. In chapter 1 we developed the need for specialized back-end storage subsystems to offload storage management from the hosts, as well as providing an environment in which storage and storage management can be optimized. In this chapter, we present a class of fault-tolerant extensible architectures suitable for use in the design of back-end storage subsystems for distributed processing systems. The architectures are based on a modified symmetrical hierarchical tree system to which topological additions have been made to provide fault-tolerance. We show that such structures provide arbitrary extensibility, improved reliability, and employ relatively simple routing algorithms. We concentrate on a representative of the class and show how it can be applied to the design of Back-End Storage Networks (BSN). This representative serves as the basis for the implementation of FTSS.

6.1 Introduction

As computer systems become more decentralized, there is a greater need to share resources that are either too specialized or too costly to replicate. This need has led to the development of specialized *Back-End* processing systems

which provide functions that are in a sense *optimized* and that may be shared by several users. Characteristic of such systems are Database Machines [DeWi79, Bana79, Su79], Back-End Storage Networks [Thor80, Svob81, Ewin82], and multiprocessing systems such as X-Tree [Sequ79], Hypertree [Good81], and MPP [Batc80]. When one examines these back-end processing systems, it is surprising to note the lack of emphasis that has been placed on system reliability as a part of the design methodology, particularly so in view of the usually high cost and specialized nature of these systems. The application of fault-tolerance, a proven design methodology for achieving high reliability [Avi78], is noticeably lacking. In this chapter, we examine one approach to the application of fault-tolerance to the design of BSN's. We show that by making modest assumptions about the frequency of failures and the mean time to repair those failures, it is possible to design a reliable fault-tolerant tree architecture for a BSN.

The BSN is a shared storage system that permits high-speed information transfer between a set of storage devices and host computers. Early BSN approaches took advantage of the cost-sharing and optimizations that could be achieved by pooling storage resources; the BSN simply performed high-speed file transfers and did very little information processing. A recent trend in BSN architectures is the off-loading of storage management functions from the host computers to the BSN; in fact, the BSN represents an excellent opportunity to bring processing to information (i.e. by processing information *in situ*) rather than bringing information to the host processor, processing it, and returning it to the storage system.

BSN's provide several advantages over the alternative, which is to distribute storage among participant sites in a shared network environment or individually among users. First, they provide a means of off-loading information processing and storage management from host processors allowing some reduction in the code size of host operating systems. Second, by considering the BSN as a homogeneous system, it is possible to optimize information and resource management in the system in ways which are either very difficult or not possible in distributed storage environments. Third, by taking advantage of economies of scale, it is possible to reduce the overall cost of storage; this is an important consideration since studies have shown that the increasing operating costs far outweigh the reducing cost per bit of storage [Ewin82]. By pooling and sharing resources, operating costs can be reduced. Fourth, the BSN provides an environment in which information can be automatically migrated and archived as a function of frequency of use. Fifth, information can be incrementally backed up without involving host computers. Sixth, the homogeneity of the BSN environment permits system designers to share data between inhomogeneous hosts by having the BSN perform data transformations invisibly to hosts. (This introduces the larger issue of *data compatibility* in information processing systems. By performing transformations on data being passed between dissimilar hosts, the BSN provides at least one solution to this issue.) Finally, the BSN provides a means whereby storage and data management functions can be provided for hosts that do not inherently possess such capabilities; this is particularly important in view of the current trend towards diskless workstations.

The remainder of the chapter is structured as follows: In section 2, we discuss various extensible architectures and their suitability for use in BSN design. In section 3, we present a fault-tolerant architecture based on hierarchical tree systems which appear promising for use in BSN's, and discuss various properties of the architecture. In section 4, we perform a simple reliability analysis of the resulting structure. Section 5 discusses the use of hierarchical trees as BSN architectures by presenting one of several interpretations of the use of links and nodes.

6.2 Architectural Considerations

There are two important factors that warrant considerable attention in the design of BSN's: 1) the architecture should be very reliable at a reasonable cost, in order to achieve high information availability and, 2) the architecture should be extensible so that the system can take advantage of the cost benefits associated with incremental growth. In this section, we discuss two promising extensible architectures and their suitability for applications in BSN's.

There are several extensible topologies that warrant consideration for applicability in BSN architectures. These include various forms of *rings*, *buses*, *stars*, *arrays*, *lattices*, *networks*, and *trees* [Desp78]. However, of these, buses and trees appear to be the most promising with respect to ease of extensibility and application of fault-tolerance.

6.2.1 Extensible Topologies

We define an *extensible* topology as one in which new components, in the case of Local Computer Networks (LCN) -- nodes, can be added to the topology without destroying its fundamental properties. Two of the important

requirements of any BSN are the abilities to incrementally increase the storage capacity of the system and the capability of building new services out of existing ones [Wats80].

6.2.1.1 Tree Topologies

The definition of *trees* are well known (see, for example [Knut73]). They are perhaps the best example of incrementally extensible structures since it is always possible to extend the tree through the addition of a single node or link, unless it is necessary to maintain the "balance" of the tree. It is also relatively straightforward to increase the reliability of the tree through the addition of extra links and nodes [Haye76, Ragh83]. In the case of binary trees, it has been shown that simple routing algorithms can be developed to pass messages between any pair of nodes in the tree [Desp78]. One potential difficulty with the application of trees in the design of BSN's is that all communication between the host and the BSN, or between the BSN and another subnetwork, must take place via the root node, implying that this node could prove a "bottleneck" to the performance of the system. It is interesting to point out that simple analysis shows that in a full binary tree with every node communicating to every other node, the root will not have the most traffic [Desp78]. However, this potential performance bottleneck certainly cannot obviate the useful properties of tree structures with respect to incremental extensibility, fault-tolerance, and simplified message routing. They therefore warrant serious consideration for use as BSN architectures.

6.2.1.2 Bus Topologies

Bus topologies are perhaps the most widely used of all extensible topologies. A bus is characterized as a high-bandwidth communication medium, of at most a few hundred feet in length, to which devices that must communicate are attached. As such, several other extensible topologies (most notably the *star* and the *ring*) can be classified as special cases of bus topologies, depending upon their physical configuration. Buses are arbitrarily extensible, limited only by the available bandwidth. It is also relatively easy to increase its reliability through redundancy, and the cost of most bus communication medium is inherently low. The primary objection to our use of buses in BSN's is the fact that all communication must occur over the bus; this can, at times, be detrimental especially when communication is preemptive. Despite this, they have been particularly successful in BSN's.

6.3 A Highly Reliable Architecture for BSN's

We have chosen a tree topology as the starting-point of our investigation of reliable extensible topologies. In this section we investigate the properties of a class of hierarchical tree topologies which exhibit high reliability through the use of redundancy via the addition of extra communication links to a basic binary tree. We show that these tree topologies exhibit properties which make them especially suitable for use in fault-tolerant BSN architectures.

6.3.1 Hierarchical Tree Systems

The theoretical framework for Hierarchical Trees has been laid out by a number of researchers [Haye76, Kwan81]. The model we use in this paper is similar in many respects to the model developed by Hayes, and we borrow liberally some of the definitions and concepts presented there. However, whereas Hayes' model is very general, we are attempting to solve a very specific problem related to the design of BSN's. Therefore, where differences exist in the modeling approaches, we point them out carefully.

Definition: A *facility* is any hardware or software component of a system that can fail independently of the remaining facilities.

Facilities in our system are comprised of hardware components such as processors, device controllers, storage devices, storage media, and software components such as executive and application programs. Each facility has access to certain other facilities in the system which permits the representation of the system by a graph.

Definition: A *facility graph* is an *undirected* graph G whose nodes $\{x_i\}$ represent system facilities and whose edges represent communication links between the facilities. G is a *labeled* facility graph if a number t_i is associated with each node x_i of G ; t_i is interpreted as the *facility type* of x_i .

Definition: A *hierarchical tree* $T_{D,p}$ is a rooted tree such that all nodes are grouped according to levels, the *level* of a node being

the distance of this node from the root node $x_{0,1}$ (the only node at level 0). p denotes the number of levels in $T_{D,p}$. For any node $x_{i,j}$ in $T_{D,p}$, i is the level of $x_{i,j}$ and j is the index ($2^i \leq j \leq 2^{i+1} - 1$) of the node in level i . A node \bar{x} is called a *child* of x if x is in level i , \bar{x} is in level $i+1$, and x is adjacent to \bar{x} . D denotes the set $\{D_0, D_1, \dots, D_{p-2}\}$ where $D_i = \{d_{i,2^i}, d_{i,2^i+1}, \dots, d_{i,2^{i+1}-1}\}$. Here $d_{i,j}$ denotes the *degree* of $x_{i,j}$, that is, the number of children of $x_{i,j}$.

It is normal to associate a facility type t_i with each level i of $T_{D,p}$ such that all nodes in level i are of type t_i and different levels may have different facility types. In view of the trends in current mass storage technology, this is not an unreasonable assumption.

Definition: A *symmetric hierarchical tree* is a hierarchical tree $T_{D,p}$ in which every nonterminal node has degree d , that is, if we write $D = \{d_0, d_1, \dots, d_{p-2}\}$ for the case when $d_{i,2^i} = d_{i,2^i+1} = \dots = d_{i,2^{i+1}-1} = d_i$ for each i , then $d_0 = d_1 = \dots = d_{p-2} = d$. In other words, $T_{D,p}$ is a d -ary tree.

Hayes, and later Kwan and Toida, have studied the optimal design of fault-tolerant symmetrical hierarchical trees. In the next section we show how fault-tolerance can be applied to hierarchical tree systems, not in an optimal sense, but in a way in which the desirable properties of the tree structure with respect to extensibility and routing are preserved.

6.3.2 Fault-Tolerant Hierarchical Tree Systems

The use of hierarchical tree systems and, in particular, symmetrical hierarchical trees in the design of computing systems has been restricted to the design of multiprocessor systems in which each node of the tree represents a processing site or a site from which input or output (I/O) might take place. Examples of these are the X-Tree multiprocessing system [Desp78, McCr80, Sequ79], and the Hypertree multiprocessing system [Good81]. Tree structures have also been used, to a somewhat limited extent, in the design of computer communication systems. As a result of this trend towards multiprocessing, much of the work that has been done in the application of fault-tolerance to tree architectures has been constrained by the need to maintain the rigid tree structure, even in the presence of failures. The optimal fault-tolerant symmetrical hierarchical trees proposed by Hayes, for example, provide mechanisms by which the tree structure is maintained in the presence of faults by switching in "spare" nodes and links to replace faulty facilities. The system is considered unreliable or "failed" when no more spare nodes and links are available and the tree structure can therefore no longer be maintained. Later work by Raghavendra, *et al.*, show non-optimal schemes which improve reliability in the presence of faulty nodes and links, but again under the constraint that the tree structure must be maintained at all times [Ragh83].

The following definitions are useful in allowing us to say things about our topology later:

Definition: A k -fault F in a system S is the removal of any k nodes $\{x_1, x_2, \dots, x_k\}$ from S . All edges connected to these nodes

are also removed. The resultant graph will be denoted by S^k .

Definition: A *simplex system* S^0 is a non-redundant system that cannot tolerate any faults. The corresponding graph is called the *simplex graph*.

Definition: S is *k-fault tolerant* with respect to S^0 if, for every k -fault F in S , there is a vertex connected subgraph of F . S is called a *k-FT realization* of S^0 .

Notice that our definition of k -fault tolerance is much more liberal than that in Hayes. Whereas in Hayes the resulting subgraph after a k -fault must be isomorphic to S^0 , we require only that there be vertex (or node) connectivity in the resulting subgraph. The result of this generalization is that we can provide k -fault tolerance solely through the addition of extra links rather than extra links and nodes (as we will illustrate in a later section). In a practical sense, what we are saying is that it is not necessary to always maintain isomorphism so long as the mean time to failure of a facility is much longer than the mean time to repair of that facility. In terms of fault-tolerance jargon, we treat our system as an *open* or *repairable* system -- thus, it is possible to allow some system degradation during periods of outages (node failures). A temporary degradation is acceptable as long as the availability of the complete tree remains sufficiently high.

Our basic approach to maintaining high availability is to provide node connectivity in the presence of failed nodes. This we do by providing additional links to the tree in such a way that the system can degrade gracefully.

We make the assumption throughout this paper that links are bidirectional, allowing information to move freely between nodes in any direction. We will show that by using this approach, it is possible to design systems with as high (or higher) reliability than when isomorphism is maintained. We should point out that a similar approach is taken in X-Tree and Hypertree. In the next section we present the basic topology of a fault-tolerant hierarchical tree system suitable for use in BSN architectures.

6.3.3 The Basic Topology

The *basic* topology we are proposing is based on a 2-ary symmetric hierarchical tree system in which additional bidirectional "cross" links have been placed to make the resulting topology 1-FT. This topology is shown in Fig. 6.1. For this example, the basic topology is a $T_{2,4}$ hierarchical tree system. The nodes have been labeled in accord with our definition of hierarchical tree systems; at this time, we afford no special significance to the types of nodes (i.e., facilities) that exist at each level. It can be readily seen that the tree is essentially a complete (or full) binary tree to which extra links have been added to improve reliability over the simplex graph.

The topology we have presented above can be generalized for any $T_{2,p}$ system in the following way:

A pair of nodes $(x_{i_1 j_1}, x_{i_2 j_2})$ are connected if they satisfy any of the following relationships when $j_2 > j_1$:

$$\left. \begin{array}{l} j_2 = 2j_1 \\ j_2 = 2j_1 + 1 \end{array} \right\} i_2 = i_1 + 1$$

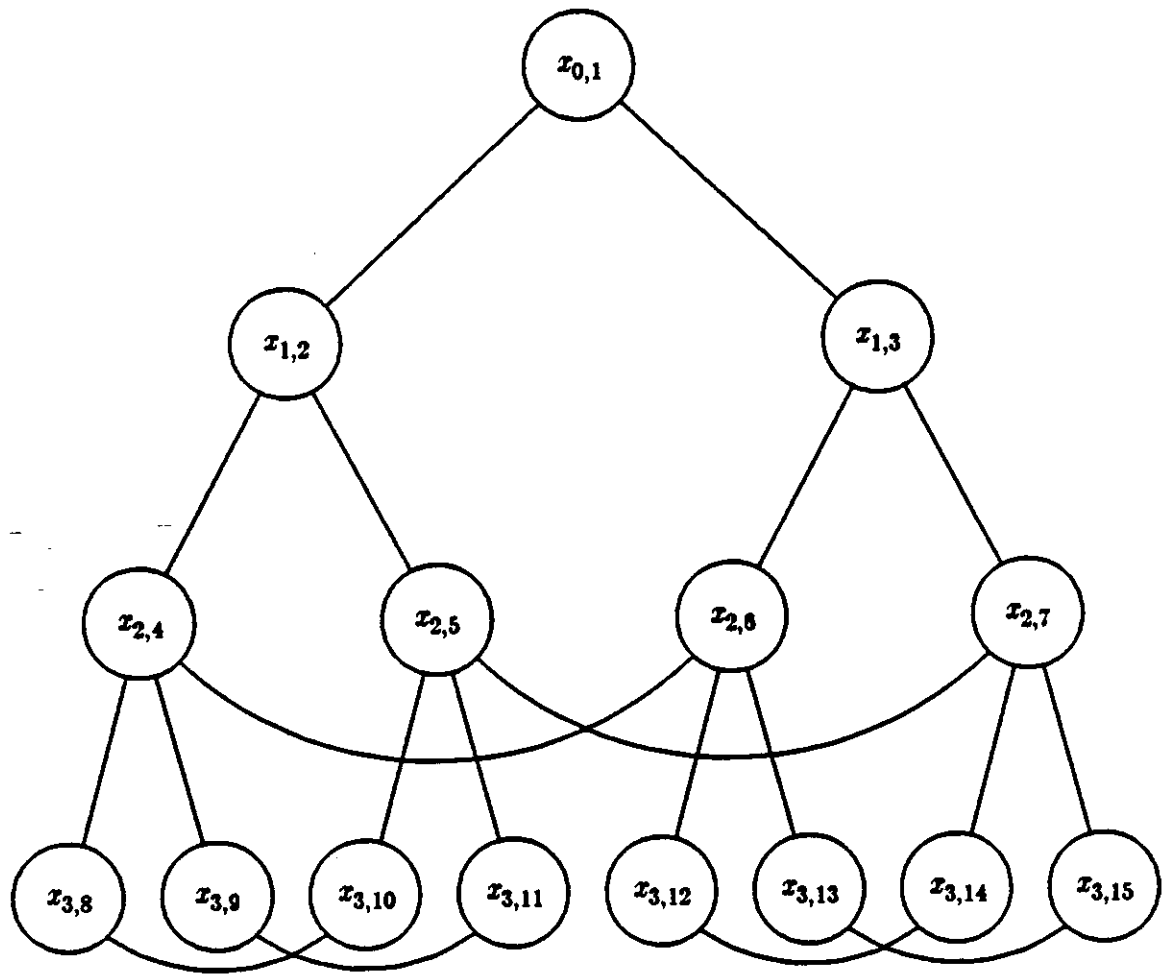


Figure 6.1: A $T_{2,4,1}$ 1-FT Symmetrical Hierarchical Tree

(6.1)

$$j_2 = j_1 + 2 \left. \vphantom{j_2} \right\} \left\lfloor \frac{j_2}{2} \right\rfloor \text{ even, and } i_1 = i_2 \quad (6.2)$$

The 1-fault tolerance of the resulting topology is guaranteed by observing that every node is *reachable* in the presence of any single node failure, except the root node (the problem with the root node is dealt with later).

6.3.4 Properties of the Topology

We will now investigate several interesting properties of these 1-FT $T_{2,p}$ topologies. We will denote topologies of the type we have defined above as $T_{2,p,1}$ topologies where the added term of 1 in the subscript indicates the k-fault tolerance of the system.

Theorem: The number of links in a $T_{2,p,1}$ tree is bounded from above by

$$2^{p+1} + \sum_{i=2}^{p-1} 2^{i-1} - 2 \quad (6.3)$$

Proof: We know that a tree with n vertices has exactly $n-1$ links (edges). Assume that the tree is full. Let N denote the number of nodes in the tree. In general, $N = 2^{p+1} - 1$. Thus, there are exactly $2^{p+1} - 2$ links in the tree, not counting the additional cross links. We only need to find the number of additional links in the tree to complete our proof. It is not hard to see that there are exactly 2^{i-1} additional links in each level of the tree for $i > 1$ (there are no cross links in the first two levels of the tree)

since every node in each level is connected to exactly one other node in the same level via a cross link; thus the number of cross links is exactly one half the number of nodes in each level, and there are exactly 2^i nodes in each level. Therefore, the total number of additional cross links is

$$\sum_{i=2}^{p-1} 2^{i-1}$$

Q.E.D.

If the reliabilities of links are treated differently from the reliability of nodes, that is, link and node failures are independent (although the failure of a node implies the failure of all links connected to that node), then the number of links in the topology becomes significant. A topology with a large number of links is likely to be less reliable than one with a smaller number of links, all else being equal.

Theorem: The maximum degree of any node in a $T_{2,p,1}$ tree is 3.

Proof: The proof begins by observing that the maximum degree of any node in a binary (2-ary) tree is 2. The basic substructure of a $T_{2,p,1}$ tree is a binary tree. It is therefore necessary and sufficient to show that in a $T_{2,p,1}$ tree no node has more than one additional cross link associated with it. We note that nodes connected by cross links are related by the expression

$$j_2 = j_1 + 2 \left. \vphantom{j_2} \right\} \left\lfloor \frac{j_2}{2} \right\rfloor \text{ even, and } i_1 = i_2 \quad (6.4)$$

when $j_2 > j_1$. It is clear that additional links are generated only

between pairs of nodes in which the parent of the lower numbered node j_1 is represented by an even integer, and the nodes are in the same level. This is the same as saying that cross links are generated on the descendants of even-numbered nodes (for the value of j_k), and terminate on the descendants of odd-numbered nodes. Thus for every node pair that satisfies the relation above, there is at most one link.

Q.E.D.

From the point of view of node complexity, in terms of information transfer and message routing, it is important to keep the number of connections per node as small as possible. This topology succeeds in this respect by guaranteeing that no node will require more than four connections to communication channels.

Corollary: All terminal (leaf) nodes in a $T_{2,p,1}$ tree have degree 1.

Proof: Non-terminal nodes have no children (by definition) and are connected to exactly one other node in the same level via a cross link.

Q.E.D.

Theorem: The diameter D_T (defined as the largest distance between any node pair in terms of the number of links separating them) of any $T_{2,p,1}$ tree is

$$D_T = \begin{cases} i, & i \leq 5 \\ 2i - 5, & i > 5 \end{cases} \quad (6.5)$$

Proof: The proof appears in Appendix A.

The diameter of any interconnection structure is an important metric in determining the maximum distance a message or other body of information must travel within the structure. While it may be very rare that this maximum distance is utilized, it is nevertheless a useful bound. It is interesting to note that the diameter of a full binary tree is $2i$, where i is the number of levels in the tree. Thus the trees we are proposing here represent an improvement over a simplex tree. Clearly, it is advantageous to minimize the number of levels in order to minimize the diameter.

6.3.5 Routing

The regularity of the $T_{2,p,1}$ topologies suggest the existence of relatively simple routing algorithms. While we are not able to present in-depth routing algorithms for these topologies in this thesis, it is nevertheless interesting to explore routing somewhat intuitively. As far as the basic binary tree is concerned, routing is relatively straightforward [Knut73]. The only additional complication concerns the use of *cross links*. Intuitively, a cross link should be taken whenever it will reduce the number of hops between a source and destination node. The general rule we adopt is that a cross link is traversed whenever any two nodes along a path from source to destination belong to the same $T_{2,3,1}$ substructure *AND* such a traversal will result in a smaller number of hops than if it is not utilized. The algorithm can be made robust in the presence of node failures by traversing a cross link whenever a path from a source to a destination node is blocked by a failed facility. We have been able to develop a simple distributed routing algorithm based on these intuitive notions which, while not optimal, will always choose a path between source and

destination which minimizes the number of links traversed; this algorithm is sketched in Appendix B.

6.3.6 Extensibility

Although tree structures are, in general, arbitrarily extensible, there are only a finite number of ways in which new nodes can be added to $T_{2,p,1}$ trees. We discuss them here.

The *Criterion for Extensibility* of $T_{2,p,1}$ trees is:

All non-terminal nodes of the tree must have degree at least two and all terminal nodes must have degree one.

In meeting this criterion, none of the other properties of the tree should be violated (for example, the connection property that determines when two nodes are connected by a link). Fig. 6.2 shows some of the ways in which a $T_{2,p,1}$ tree can be extended. Note that the criterion above requires that a *minimum* expansion consist of two nodes that are connected by a cross link. It should also be noted that the criterion permits the tree to be extended in an *unbalanced* way; that is, some of the levels do not have their full complement of nodes (incomplete levels). The desirability (or undesirability) of extending the tree in this way is a function of the resulting complexity of the routing algorithms used to move information and messages between the nodes. Simple routing algorithms will probably require that only the highest numbered level of the tree be incomplete.

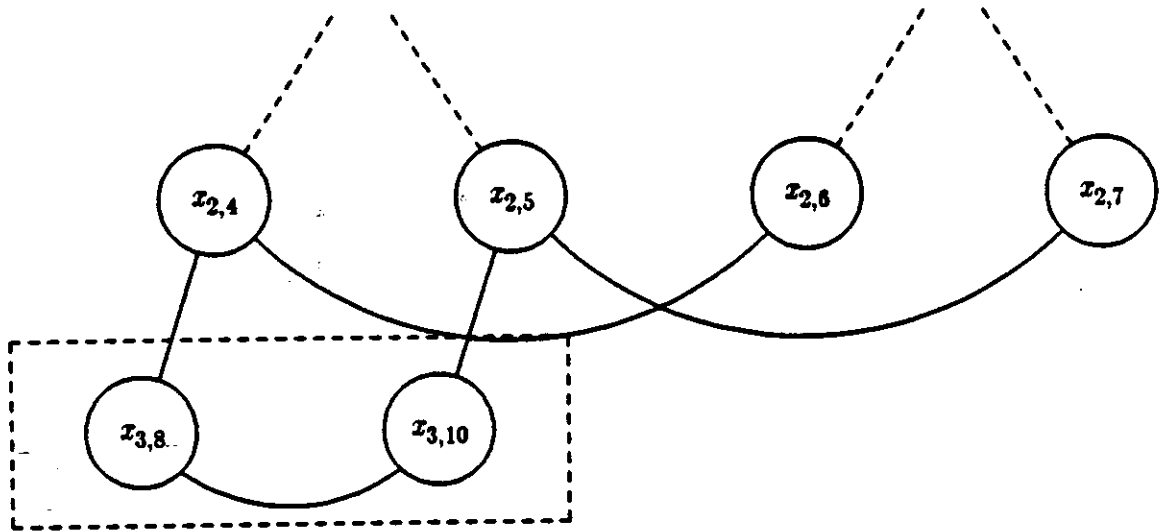


Figure 6.2a: Two-Node Extension

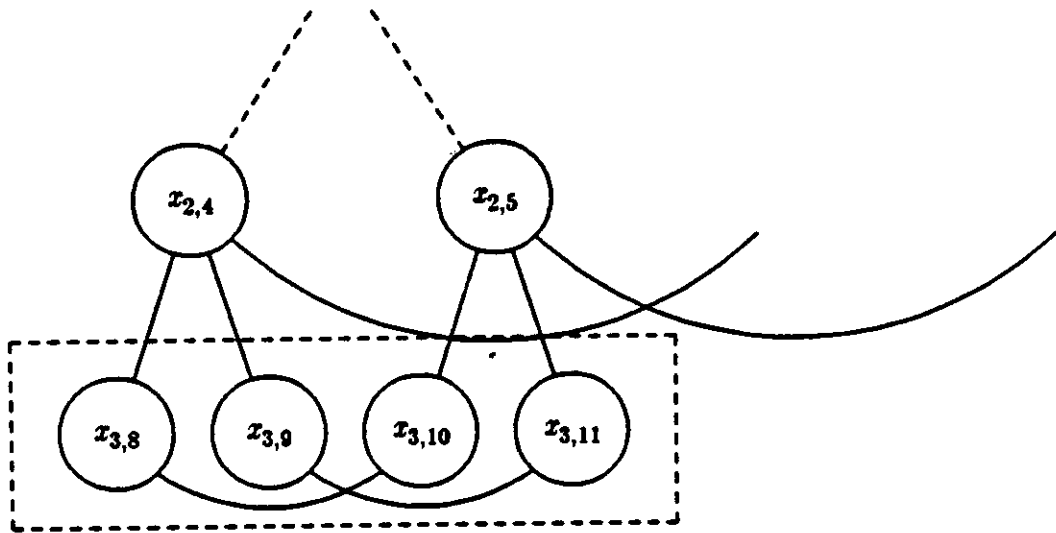


Figure 6.2b: Four-Node Extension

6.4 Fault-Tolerance and Reliability

In contemporary information processing systems, the use of fault-tolerance to improve system reliability at lower costs than other alternatives is well established [Avi78]. In this section we examine, in a quantitative way, the reliability improvements that are accrued as a result of our topology. As pointed out previously, the topology was designed from the outset to be 1-FT. However, it is possible that multiple failures can occur in different parts of the tree without violating the reachability condition defined for k -fault tolerance. We assume throughout the rest of this section that the failure rate of links is negligible compared with nodes, thus only node failures are considered. Admittedly, this is an optimistic assumption, but it is supported by the observation that the links are bidirectional and link failures do not imply that a node is unreachable; relatively simple reliability analysis of the topology is made possible by this assumption.

6.4.1 Reliability Analysis

In performing reliability analysis of $T_{2,p,1}$ topologies, we consider the topology as a fault-tolerant structure consisting of a series of homogeneous subsystems $S_{i,k}$ ($i = 0, 1, \dots, p-1$; $k = 1, 2, \dots, L$) where i represents the level of the tree (the i -th level) and k represents the number of subsystems in the i -th level. With a failure rate λ , the reliability of a single node is

$$R = e^{-\lambda t} \tag{6.6}$$

In a simplex (non-redundant) binary tree with p levels, there are $2^p - 1$ nodes, all of which must be operational for the system to be considered operational (i.e., reliable). Therefore, the reliability of the non-redundant binary tree is

$$R_T = R^{2^p - 1} \quad (6.7)$$

In a $T_{2,p,1}$ tree, all levels except the first can tolerate node failures. The allowable number of failures per level, however, is determined by i (the level number) and the way in which the nodes are interconnected in that level. For example, level 1 (the second level) can tolerate a single node failure while all other higher-numbered levels can tolerate several node failures. To illustrate the reliability of the topology, we consider several cases:

Case 1: Single node failures

Under this assumption, any node except the root can fail. It can be shown that the reliability of the $T_{2,p,1}$ tree is given by:

$$R_{T_{2,p,1}} = R^{2^p - 2} + (2^p - 2)R^{2^p - 3}(1 - R) \quad (6.8)$$

since there are $2^p - 2$ nodes, each of whose failure can be tolerated.

Case 2: Double node failures

Under this assumption, we wish to characterize the reliability of the topology for all double-node failures. If we do not consider the root node, there are $\binom{2^p - 2}{2}$ ways in which double-node failures can occur. However, not all such failures will leave the non-failed nodes connected. If node failures occur in different levels of the tree, clearly the structure remains connected, the only exception being the case when a failure occurs in each of the two highest levels of the tree. In this latter case, it can be shown that there are 2^{p-1} ways in which such failures can lead to unreliability. There is one other case in which double-node failures can lead to unreliability, and that is the case where two failed nodes are in the same level AND they have the same parent node (the

leaf nodes being an exception). Out of the $\binom{2^i}{2}$ possible double-node failures in each level, exactly 2^i are safe, the others leading to unreliability. Based on this analysis, it is possible to characterize the double-node reliability as:

$$R_{T_{2^p,1}}^D = R^{2^p - 2} + (2^p - 2)R^{2^p - 3}(1 - R) + kR^{2^p - 4}(1 - R)^2 \quad (6.9)$$

where

$$k = \frac{\binom{2^p - 2}{2} - \sum_{i=2}^{p-1} 2^i - 1}{\binom{2^p - 2}{2}} \quad i \geq 2$$

Case 3: Multiple node failures

We are able to go one step further in our reliability analysis by considering the more general case of multi-node failures. In order to do so, we break up the system into a number of homogeneous subsystems.

We notice from Fig. 6.2 that in any level of the tree for $i > 1$, node pairs are connected with cross links so that every four nodes starting from the leftmost node in that level represents the lowest level of a $T_{2,3,1}$ tree. Each of these node groups forms an independent subsystem for the purpose of reliability analysis. We will refer to these node groups generically as a *Group Subsystem* or GS. The number of GS's in each level of the tree is 2^{i-2} for $i > 2$; we assume that the GS's are in series for reliability analysis since they must all survive for level i to survive. The reliability of any GS except those in the highest level of the tree (the leaf nodes) is:

$$R_{GS} = R^4 + 4R^3(1 - R) = R^3(4 - 3R) \quad (6.10)$$

Notice that only single-node failures are allowed; multiple-node failures make the structure unreliable. There are cases in which a failure in a leaf node and a node in the immediately preceding level can make a surviving leaf node unreachable (this case occurs when both the parent and the connected sibling of a leaf node fail). Since this is expected to occur in only a small number of cases (especially if the number of levels is small), we choose to investigate the reliability under the single-failure assumption for leaf nodes in the same GS (otherwise, we must assume that leaf nodes do not fail.) From a practical point of view, it is possible to tolerate these failures by replicating information in leaf nodes belonging to the same GS that are not siblings (i.e., connected by a cross link), or by adding additional links between leaf nodes. Under these assumptions, $R_{GS_{leaf}} = R_{GS}$. The reliability of level i is therefore

$$R_i = \begin{cases} \prod_{k=1}^{2^{i-2}} R_{GS_k} & ; \text{if } 1 < i < p - 1 \\ 2^{p-3} R_{GS_{leaf}} & ; \text{if } i = p - 1 \end{cases} \quad (6.11)$$

Since all levels from $i = 0, 1, \dots, p - 1$ must be operational for the tree system to be operational, the system reliability is given by the expression:

$$R_{sys} = \begin{cases} R & ; \text{if } i = 0 \\ R(R^2 + 2R(1 - R)) & ; \text{if } i = 1 \\ R(R^2 + 2R(1 - R)) \prod_{i=2}^{p-1} R_i & ; \text{if } i \geq 2 \end{cases} \quad (6.12)$$

We can compare the reliability of the $T_{2,p,1}$ system with the non-redundant binary tree by evaluating the *Reliability Improvement Factor* (RIF) as:

$$RIF = \frac{1 - R_T}{1 - R_{sys}} \quad (6.13)$$

If we assume that the fault coverage c (defined as the conditional probability of successful detection of a fault, given that a fault has occurred) [Bour69] is not perfect, then the system reliability is given by

$$R'_{sys} = \begin{cases} R & ; \text{if } i = 0 \\ R(R^2 + 2cR(1 - R)) & ; \text{if } i = 1 \\ R(R^2 + 2cR(1 - R)) \prod_{i=2}^{p-1} R_i & ; \text{if } i \geq 2 \end{cases} \quad (6.14)$$

where R_{GS} is now expressed as:

$$R'_{GS} = R^4 + 4cR^3(1 - R) \quad (6.15)$$

As an example, we will perform a reliability analysis for a $T_{2,4,1}$ tree and compare the results with the corresponding 4-level binary tree. We begin with a study of the effect of node reliability on overall system reliability under the assumptions that all nodes have the same failure rate and that coverage is perfect. In Fig. 6.3 we show the reliability of the system for the three cases discussed above, under the assumptions of perfect coverage and a failure rate $\lambda = 0.1$ failures per unit time. The time unit may be assumed to be any value convenient to the reader. A typical interpretation of time in this context would be thousands of hours. In the figure we have also included the reliability of the non-redundant 4-level binary tree as a means to compare the reliability improvement. Table 6.1 shows the reliability of the $T_{2,4,1}$ tree under the assumptions above by levels in the tree -- this is useful since it helps one to get a feeling for the contribution of each level to reliability (or unreliability). We also show in Table 6.2 the improvement in reliability of the $T_{2,4,1}$ tree over the

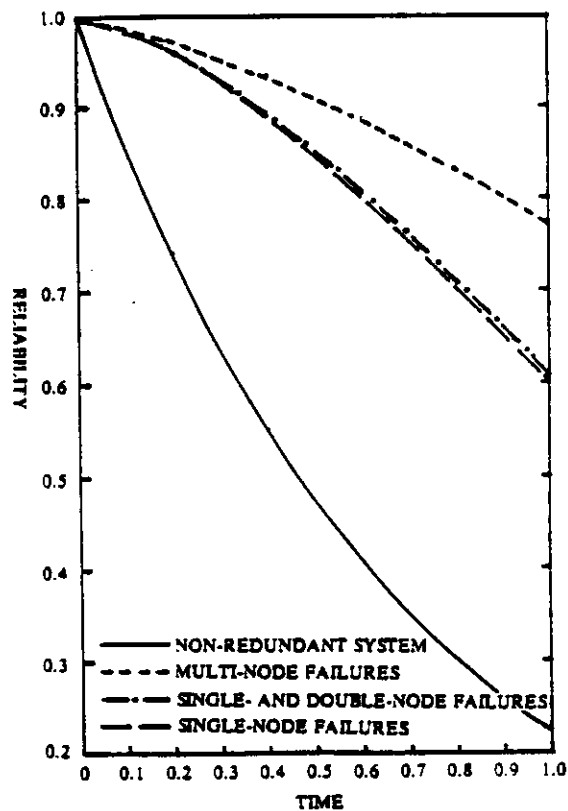


Figure 6.3: System reliability under different failure assumptions
 $\lambda = 0.1$ coverage = 1

t	R_1	R_2	R_3	R_4	R_{sys}
0.0	1	1	1	1	1
0.2	0.9801987	0.9996079	0.9970909	0.9954234	0.9730958
0.4	0.9607894	0.9984625	0.9912504	0.9825773	0.9343511
0.6	0.9417645	0.9966086	0.9811973	0.9627481	0.8866168
0.8	0.9231163	0.9940889	0.9680643	0.9371486	0.8325193
1.0	0.9048374	0.9909440	0.9523127	0.9068996	0.7743878

Table 6.1: Reliability of a $T_{2,4,1}$ Tree

non-redundant binary tree in terms of the RIF. An important metric in determining the relative merits of one system over another in reliability analysis is the *Mission Time Improvement Factor* or MTIF. This is the improvement in mission time of one system over another for a given level of reliability. This we show in Table 6.3.

$\lambda = 0.1$		<i>coverage = 1</i>	
<i>t</i>	R_{sys}	R_T	<i>RIF</i>
0.0	1	1	0
0.2	0.9730958	0.7408182	9.63
0.4	0.9343511	0.5488116	6.87
0.6	0.8866168	0.4065697	5.23
0.8	0.8325193	0.3011942	4.17
1.0	0.7743878	0.2231302	3.44

Table 6.2: Reliability Improvement of $T_{2,4,1}$ Tree

$\lambda = 0.1$		<i>coverage = 1</i>	
Rel. Level	Mission Time	MTIF	
0.9731	0.2	10.99	
0.9344	0.4	8.791	
0.8866	0.6	7.481	
0.8325	0.8	6.547	
0.7744	1.0	5.869	

Table 6.3: Mission Time Improvement of $T_{2,4,1}$ over Non-Redundant Tree

Although it is desirable in some applications (most notably, VLSI systems) to have identical nodes in the tree structure, it is hardly likely that this would be the case in the context of a BSN. We would probably expect, for example, the root node to be well protected (low probability of failure) because of its criticality in network operations. Furthermore, we would expect that failure rates would decrease as the level in the tree increases based upon the characteristics of contemporary mass storage devices, that is, we think of the tree architecture as representing a hierarchy of storage devices with capacity increasing with the level of the tree and speed of information handling decreasing with increasing level. We have modeled just such a case for a $T_{2,4,1}$ tree and the results are shown in Table 6.4.

Level No.	Coverage	λ
1	1.0	.0001
2	1.0	.01
3	0.98	.001
4	0.95	.0001

Table 6.4a: Fault Coverage and Failure Rate at each Level

t	R_1	R_2	R_3	R_4	R_{sys}	RIF
0.0	1	1	1	1	1	0
0.2	0.99998	0.999996	0.9999838	0.9999919	0.9999518	103.01
0.4	0.99996	0.999984	0.9999671	0.9999839	0.9998951	94.51
0.6	0.99994	0.999964	0.9999499	0.9999759	0.9998301	87.297
0.8	0.99992	0.9999365	0.9999323	0.9999679	0.9997568	81.1
1.0	0.9999	0.999901	0.9999143	0.9999599	0.9996752	75.72

Table 6.4b: $T_{2,4,1}$ Tree with Different Failure Rates and Coverages at each Level

6.4.2 Discussion of Results

Two things are immediately clear from our simple reliability analysis: 1) The $T_{2,4,1}$ tree is more reliable than the corresponding 4-level binary tree for all cases of interest, and 2) The redundant reliability is very sensitive to the fault coverage present in each subsystem (or level of the tree in this case).

The relatively high levels of RIF that we have obtained attest to the fact that fault-tolerance can also be applied to *repairable* tree architectures without resorting to the dynamic methods of *sparing* that are necessitated in *closed* architectures [Haye76, Ragh83].

It is also clear that in order to achieve high reliability for general $T_{2,p,1}$ topologies, it is necessary to *balance* the reliabilities in each level of the tree. This suggests that, in particular, the nodes in the top two levels of the tree should have a higher reliability than the other nodes in the tree since the effects of failures there tend to be more pronounced. Such techniques as using

dynamic redundancy on the root node of the tree are indeed viable, and should be employed in practical applications.

6.5 Applying the Topology in BSN's

In applying $T_{2,p,1}$ trees to BSN's, we treat each node of the tree as a *Storage Site* (SS); i.e., each SS is a facility. The SS's are interconnected with high-speed bidirectional communication paths represented by the links of the tree. Each SS has a *capacity* which is the maximum amount of information that can be stored there (independent of the number or storage characteristics of the physical devices at the SS), and a *bandwidth* which is the maximum rate at which information can be accessed at the site. Each SS might have a unique capacity and a unique bandwidth (this we leave as an implementation issue.) Communication links also have a bandwidth which is the maximum rate at which information may be sent via the link. Each link can have a different bandwidth. *One* convenient way to view the architecture (as discussed in Section 6.4), is as a hierarchy of SS's such that all SS's in the same level have similar storage characteristics (typically, the same storage bandwidth), but which may be different from storage characteristics in other levels. For example, we might consider the root node as a global controller for the entire BSN which is heavily fault-protected through the application of dynamic redundancy, the next level as representing high-speed buffers which are bandwidth-matched to the network with which the BSN communicates, and so on, eventually extending down to the lowest level of the tree which would represent bulk-storage (large capacity) sites for the dynamic archiving and de-archiving of information.

The salient feature of $T_{2,p,1}$ trees is the ability to tolerate single (and some classes of multiple) node and link failures while maintaining node connectivity. This is a particularly attractive feature in BSN's because it means that a node or link failure will not cause information stored at a SS to become inaccessible. Clearly, in order to utilize this feature effectively, it is necessary to replicate information in the BSN so that valid information exists in at least two SS's with independent failure probabilities. This is required so that if a site goes down, valid information will still be accessible from another site. The replication factor is arbitrary; however, as shown in chapter 5, given contemporary storage devices, replication at two sites with independent failure probabilities (and assuming links have a significantly lower probability of failure than storage sites) will result in acceptably high availability.

One appealing aspect of the topology in the context of a BSN is the fact that if each SS is considered an independent entity (i.e., each SS consists of a controlling processor with attached storage devices) which communicates with other SS's via the interconnecting links, then it is possible to perform *local* operations on information as well as inter- and intra-site information transfers; thus, the global controller does not have to be involved (except for initialization of a request and possible coordination of activities in the system.) The significance of this is that operations such as information migration, automatic archiving, and information copying (involving simple site transfers) can occur totally independently of activities in other parts of the architecture, and certainly without intervention of host processors; since no host transfers are needed, the root node (and buffer nodes) are not involved except for possible coordination by the root node.

What we have shown in this section is one way in which the nodes and links of $T_{2,p,1}$ trees can be used in a BSN; there are, of course, many other possible interpretations subject to the availability and type of storage devices, implementation constraints, further application of fault-tolerance to links and nodes, and so on. In chapter 7 we use a $T_{2,4,1}$ architecture as the basis of the hardware design of FTSS.

6.6 Summary

We have presented a general class of fault-tolerant architectures based on symmetrical hierarchical trees that are suitable for a wide range of applications which require reliability and incremental extensibility. We have shown how a particular member of this family of trees ($T_{2,p,1}$) might be applied in the design of BSN's to satisfy given capacity, availability, and performance constraints; we also performed simple reliability analysis of the topology to give an indication of the reliability that a designer might expect from such a system.

The ability to arrange capacities and bandwidths as a function of need, along with the ability to perform operations such as information migration, automatic archiving, and inter-device transfers -- independently of host machines and without using network bandwidth -- makes these fault-tolerant topologies extremely attractive for use in BSN's. Additionally, since fault-tolerance is achieved in a totally static way (through the addition of extra links alone), the implementation of such a system is much simpler than if dynamic fault-tolerance was used throughout.

CHAPTER 7

THE OBJECT STORE: FAULT-TOLERANCE AND OTHER ISSUES

In the previous chapter we presented a class of fault-tolerant architectures which possess features which make them very suitable for application in back-end networks. In this chapter we adopt this architecture as the framework on which FTSS is built and present several mechanisms which take advantage of its unique features. In particular, we extend those mechanisms discussed in chapter 3 which deal with the location, representation, and management of objects in the object store.

7.1 Mapping FTSS onto a Fault-Tolerant Tree Architecture

The hardware is organized as a tree of *Storage Sites* (SS) as shown by the $T_{2,4,1}$ architecture in Fig. 7.1. Each SS is structured as a *Site Processor* (SP), and one or more *Storage Devices* (SD). Each SS has a *Capacity* C which defines the amount of information which may be stored at that site. The constraints on C is a function of the availability of SD's, and on the required performance of the system. The SS's are interconnected by bidirectional communication paths (or *links*) along which information may flow from site to site in a controlled and efficient way. One may view this architecture as a *network* of storage sites arranged as a tree, and possessing a distinguished site which is known as the *root* site. For convenience, the SS's may be identified by the *level* in which they appear in the tree.

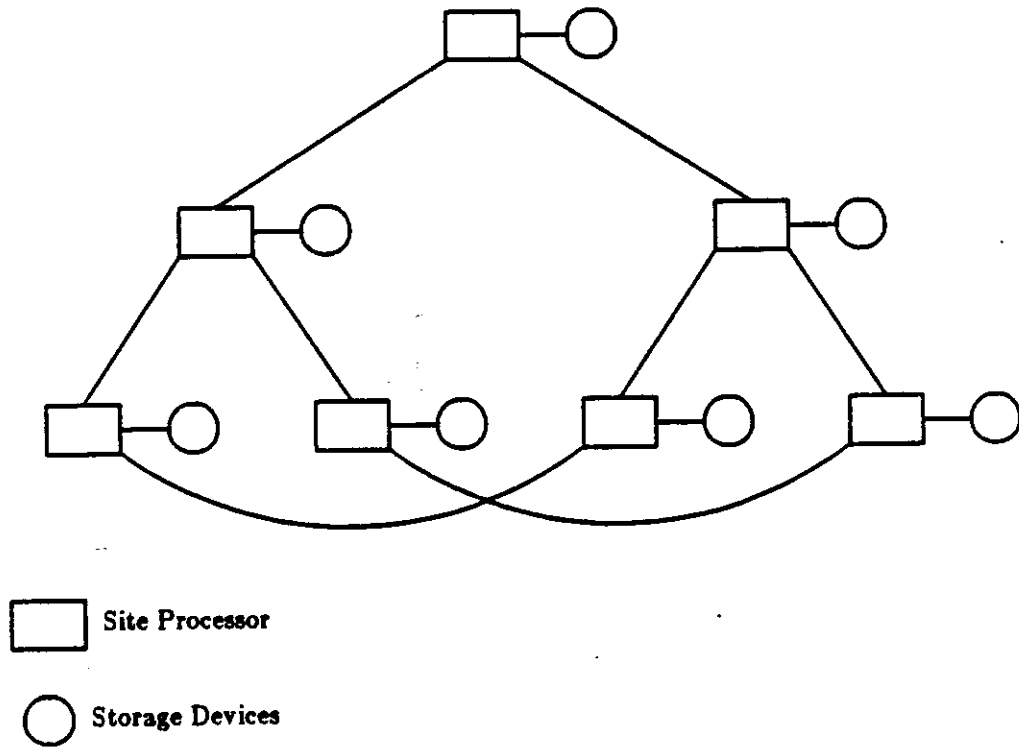


Figure 7.1: Conceptual Hardware Architecture of FTSS

Although the $T_{2,p,1}$ architecture suggests a natural hierarchy, one does not always have to view it as such. In fact, if the communication links between sites is short (tens of feet) and the bandwidth high (millions of bits per second), it can be viewed as a special case of a distributed bus architecture -- this is especially true when all sites provide the same performance. We view the FTSS architecture as being composed of three functional levels that constitute a hierarchy.¹

1. The *Top Level* (TL) provides high speed random access cache memories

¹The reader should note that a hierarchy, in the classical sense of memory hierarchies, is useful only if there is at least an order of magnitude difference in performance and capacity between successive levels.

for buffering object blocks. A processor is provided at this level to provide global management of the system, interfaces with host processors, and support of user-defined subsystems. This is the highest bandwidth portion of the system since all communication with the local network of users is handled at this level.

2. The *Intermediate Level (IL)* consists of high performance secondary storage devices. Storage objects in use by the host processors on the LAN are migrated to this fast storage. Since several hundred processors may share the services of FTSS, very fast response is needed at this level.
3. The *Bottom Level (BL)* is the bulk storage level. Here information will reside which is not actively in use. A very large store with a relatively slow response time is anticipated.

The combination of the IL and BL will be referred to henceforth as the *Internetwork*.

7.2 Object Representation

Every object in the Internetwork is represented as a triple:

<header, capability-list, data>

This is the same representation used in the kernel of the SMS except that the header is different; the header maintains information on the state of the object and other information useful in its maintenance. When the object is being accessed by the kernel, the header is the information kept in the incore directory (ICDI); when it is in the Internetwork, it keeps enough of this information so that the next time the object is activated, the incore header can simply be

updated (there are certain invariants such as the UID and the type name of the object which must be preserved from one activation to another). In addition, the Internetwork header contains some information which relates to the state of the object there. Logically, every object in the Internetwork is represented as a linear one-dimensional array of blocks; a header, capability list, and data. However, its actual representation on physical storage may not correspond to this logical organization - much depends on the access characteristics of the devices. For example, if moving head disks are used, blocks might be arranged to minimize rotational latency and head seek times; if fixed head disks are used, we might want the blocks to be stored consecutively so that access time can be minimized (the same would be true for other serial access devices such as tapes).

7.2.1 Object Representation in the IL

The IL is the set of storage sites which contain the representation of objects either currently being accessed, or having recently been accessed. These sites are designated *a priori*. Since the frequency of access is expected to be relatively high and the devices relatively unstable, additional precautions need to be taken to provide the high availability expected of the system. Based upon our analysis of chapter 5, we maintain at least two copies of each object at all times, each stored on a device with an independent failure probability. The cost of replication is not expected to be excessive since only a small portion of the object population is expected to be in the IL at any given time.

7.2.2 Object Representation in the BL

The BL is the set of storage sites which provide high capacity and relatively low performance compared with other sites. Here objects which are not in use will reside. Like the IL, these sites are determined *a priori*. Here we do not expect the access rate to be high in view of the hierarchical nature of the system. Our strategy for maintaining high availability at this level varies as a function of the cost/bit of storage. If the cost/bit is low, replication is viable; for example, if the storage medium is tape, the incremental cost of storing an object of moderate size twice is relatively low. On the other hand, if disks are being used, one may adopt the strategy of using a single disk to maintain checksum sectors over corresponding sectors of a set of disks to achieve full fault protection at a fractional increase in cost. For example, groups of four disks might be backed up by a checksum disk providing a worst case slowdown by a factor of four if one disk fails completely, but the capability to reconstruct data and recover at an additional cost of only 25%.

7.2.3 Consistency

Whenever objects are replicated, there is an issue of *consistency* that must be addressed. The idea of *consistency* has its roots in transactional database systems where it was clearly realized that data validity in the presence of software and/or hardware failures was crucial to the financial well-being of online transaction processing systems. This requirement led to the foundation of a theory of consistency which gave rise to the formal definitions of *atomicity* and *transactions* [Gray78]. While these are useful paradigms, they are all based on the premise that hardware and software are unreliable; that is, the

instants at which they may fail are unpredictable. While this is true for all implementable systems (assuming a finite amount of resources), given the current state of the art, recent trends in the application of fault-tolerance to the design of computing systems make the prediction of failures much less uncertain. It has also been demonstrated that fault-tolerance can be applied in cost-effective ways, while still producing relatively high performance.

The objective is that the components of a replicated object must satisfy some constraints about the consistency of the information which is stored in each of them. Consistency theory has been applied to distributed databases [Bern80], and to the design of transactional file systems [Dion80, Frid81]. There are two forms of consistency: one dealing with the external behavior of a system, and the other with the internal behavior. The first is known as *external* consistency, and the latter as *internal* consistency. The strongest and most popular form of external consistency is the requirement that all externally visible operations be atomic. An operation is atomic if it satisfies the properties of:

- *Failure atomicity.* The operation is executed exactly once or not at all.
- *Serializability.* Its execution is serializable with respect to other atomic actions.
- *Correctness-preservation.* Given a correct initial state, any serializable execution of atomic actions results in a correct state.

Mutual consistency is the most popular form of internal consistency. It requires that all operational components of the system occupy the same state. A more common terminology is *interactive consistency* which requires that operational components be mutually consistent, but that failed components may be

inconsistent. *Total consistency*, on the other hand, requires that even failed components reflect a serializable ordering of operations. Whether a system should exhibit interactive or total consistency is a function of the cost of implementing each; in general, it is much more costly to implement systems that are totally consistent, although they are much easier to recover from when total failures occur. The advantage of designing a fault-tolerant system that guarantees certain failure properties with high probability, is that there are tradeoffs that can be made between external and internal consistency. We take the viewpoint that given reliable hardware and software mechanisms, there is no need for state-based mutual consistency mechanisms for the internal state of a replicated object. Instead, one can satisfy consistency requirements solely from constraints on the operations and responses of the system. To put this another way, we can afford to put a certain amount of trust in the internal operations of replicated objects, with the knowledge that the system will perform correctly within its defined reliability goals. In the discussions that follow, the internal consistency of an operation is implied if a favorable response to the operation is received; otherwise, it is assumed that a failure has occurred and an appropriate recovery action is invoked. Once again, we stress that this viewpoint is reasonable only if incidences of unrecoverable failures are relatively rare; that is, if the system is fault-tolerant.¹ If this is not the case, then one has to resort to the complex protocols that are typical of distributed storage systems [Birm84, Giff82, Thom79, Alsb76]. Despite our relaxation on internal consistency, we do, however, require that all operations on a replicated object satisfy the external consistency constraint that all such operations be

¹The system must, in fact, be a periodically renewed *open* fault-tolerant system supporting high hardware and software availability.

atomic.

The basic strategy adopted in FTSS is to replicate¹ objects, when necessary, for high availability, and to apply fault-tolerance in such a way that the notion of consistency is built *into* the system, rather than on top of it. FTSS is therefore unique in this respect, which serves to differentiate it from all extant backend or file server systems. This feature of built-in consistency provides high performance by allowing parallel operations on replicated objects in the system. This parallelism provides better efficiency during access of replicated objects, which is virtually independent of the replication factor used, for any given level of reliability.

Consistency is much less of a problem in FTSS than it is in distributed storage systems because of the centralized nature of the system -- there is only a single mechanism which accesses the representation of an object, regardless of the number of users that may be concurrently accessing the object. The problem in distributed storage systems is that there are several mechanisms that can access different copies of the same object. In FTSS there are really only two ways in which inconsistency can occur: 1) a faulty block write which is not detected, and 2) a failure of a block after it has been written. The first can be detected by reading the block just after it is written using self-checking hardware, and the second by a simple encoding scheme such as block checksums. FTSS augments these mechanism by others which will be discussed below.

¹Given the current state of the art for storage devices, we have little choice.

7.3 Location

Object blocks in the Internetwork are located using a 2-phase algorithm. The first phase begins at kernel level and the second takes place at the site(s) at which the object is stored. The kernel maintains a global directory object (the GODI - see section 3.4) and each site maintains a local object directory (the LODI). Location begins at kernel level when the requested block is not in core. It begins by first searching the GODI to determine on which site(s) the object is stored, followed in the second phase by a search of the LODI to determine the location of the logical block of the object at the storage site. The algorithm is independent of the fault-tolerance strategy used, although the access mechanisms are not. The algorithm is presented below, and Fig. 7.2 illustrates the relationships between various parts of the object and the various tables used in locating them.

Algorithm 7.1: Locating Object Blocks

1. If the capability for the object is active, go to step 3.
2. Use the UID in the capability to hash into the ICDI and check to see if the object is active. If it is, change the capability to active form by replacing the UID with the address of the object header in the ICDI and go to step 4; else go to step 5.
3. We assume the capability is active and the UID part contains the actual address of the object's header in the ICDI. Access the ICDI.
4. Scan the incore block list to see if the block is in core. If it is, stop; the block has been located.
5. The block is not in core. See if it is in the RAM buffer in the TL. If it is, stop; the block is located.
6. We must locate the block in the Internetwork. Using the object's UID, hash into the incore GODI table to determine on which site(s) the object is stored.
7. If the entry is not in the incore GODI table, get it and update the table.
8. Get the site list from the GODI entry (which indicates on which site(s) the object is currently stored).

9. Send a request to a site to locate the block
10. We assume the site receives the request correctly. Access the LODI and locate the block. Stop; the block is now located.

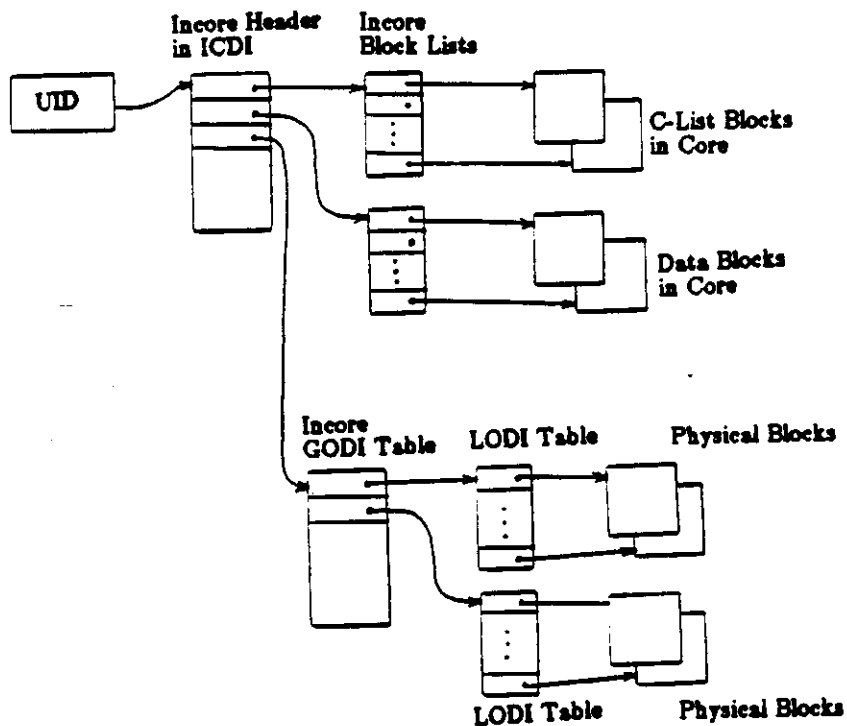


Figure 7.2: Object Block Location Hierarchy

There are a number of things worth pointing out about certain parts of the algorithm. For example, at step 6 it is assumed that the address of the incore GODI table is known; this is indeed the case - we may assume that the kernel knows this address since at system boot time the GODI is one of the objects initially loaded by the kernel. Rather than inserting this address into the header of each active object, the kernel simply keeps it in a known place. At step 7, in order to update the incore GODI table, the kernel must know

which logical block of the GODI contains the needed information so that it can have it brought in; the way this is done is to send a request to the storage site at which the GODI is stored (which the kernel knows since this location never changes - the GODI is always kept in fast storage and is never migrated) along with the UID of the object - at the site this is used as a key to index into the GODI and the block is returned. Locating a block in the GODI is exactly the same as locating a block in any other object; however, it is needed as a special case in the more general block location algorithm.

The performance of this algorithm is strongly influenced by the hit ratio of the ICDI and the RAM buffer. For good performance, these hit ratios should be very high.

7.4 Accessing Object Blocks

In this section we describe algorithms for the two most interesting access modes for objects in the Internetwork: write and read access.

7.4.1 Write/Update Access

As pointed out previously, replicated object components are identified by their GODI entry. Also recall from previous discussions that objects in the storage system are block-addressable, any further refinement is done by the requesting subsystem. When an object is therefore to be accessed for write, it is assumed that the block to be written has already been accessed (or allocated) and is under control of the OMS. The significant part of the write algorithm is: given a block to be written, write it atomically to all r replicates of the object. In general, performing an atomic write to r replicates is a complex procedure,

given the byzantine nature of the processors controlling each write [Lamp82]. However, since self-checking processors can be designed to be byzantine with low probability (they have well defined failure modes), an atomic write can be equivalenced by a *guaranteed* write by r self-checking processors. Under these conditions, consensus is not required since writes are guaranteed at each site independently. The atomicity of the write is accomplished by collecting write responses from each site, each of which verifies that the write was completed satisfactorily.

While there are several possible ways to perform an atomic write at a site (such as using shadow pages or multiple versions), the overriding concern is the need to abort a write and restore the object to the state it had before the write began. Related to this is the question: at what point should we change the state of the object being modified? There are three possibilities. At each block write, at the end of all writes (equivalent to closing a file), or at specific "commit" points. The first does not permit atomic updates, so it is discarded. The second is useful, but runs into problems if the object is kept open for a very long time (as would be the case with a transaction log for example). The third is the best approach since it permits read access to the object at various points in its modification history. The selection of a commit point may either be done under user control by explicitly making a commit request to the kernel, or automatically by the kernel based upon some simple algorithm as a function of the amount of information written to the object (this is similar in spirit to the concept of "checkpointing" a file as used by some text editing systems). In either case, it is the kernel that controls the commit actions, and not the local site processors. The need for the kernel to know the state of all ver-

sions is the real reason for not allowing local site commits -- at each commit point the kernel must update the GODI with the version ID of this latest committed version.

The two most popular techniques for allowing object state restoration during atomic updates is the use of *shadow pages* [Edwa82] and *multiple versions* [Svob81]. In the shadow page approach, modified pages are written to a separate area of permanent storage; pointers to these and other unmodified pages are maintained as a separate index until a commit point, at which time the new index is used to replace the old. In order to make things work correctly in the presence of failures, the old index is not removed until the write of the new index is confirmed; at this time the space used by the old pages can be reclaimed. In the multiple version approach, a new version of the object is created when it is first opened for write, and at each succeeding commit point. The new version is modified until a commit point is reached; at this time, the new version is first written to storage, and the old version deleted. The salient feature of the multi-version approach is that, whenever the object is being modified, there are always two versions at the same site. To illustrate the principles, consider Fig. 7.3.

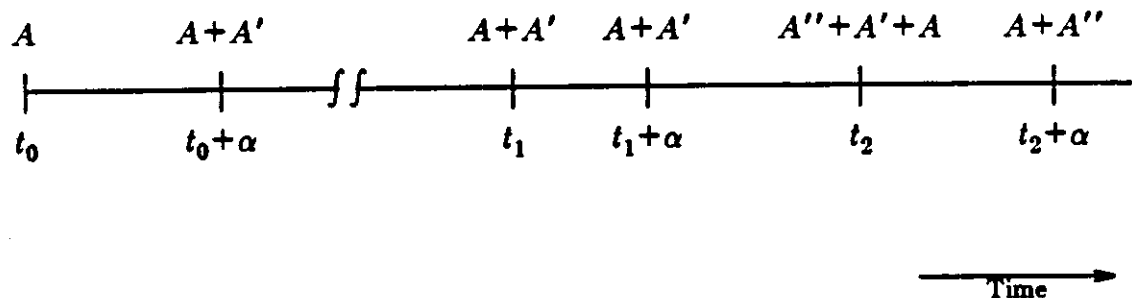


Figure 7.3: Object Update Using Multiple Versions

Assume the existence of an object A . At time t_0 , A is accessed for write. At

time $t_0 + \alpha$ A is copied to produce version A' . A is modified until time t_1 at which point a commit operation is requested. At time $t_1 + \alpha$ A is written to storage. At time t_2 A is copied to form A'' , and at time $t_2 + \alpha$ A' is deleted. In some systems, A' is not deleted, giving rise to a set of versions which represent the modification history arbitrarily far back in time; this method is proposed by [Reed78] as a means of implementing concurrency control when used with time stamps. This could be provided as an option in FTSS if its storage cost is tolerable.

In terms of the amount of mechanisms needed to implement either of the two schemes, there does not appear to be much of an advantage of one over the other; in both cases it is necessary to maintain separate indexes that point to modified and unmodified pages. The version technique has slightly better recovery semantics in the presence of failures since it is only necessary to detect the presence of more than one version of the object when recovery takes place. In the shadow page technique, it is necessary to locate and free those pages that have been modified -- a task which might be difficult if the index that pointed to these blocks were lost at the time of failure. We would therefore adopt the version technique for two reasons: a) it has better recovery semantics and, b) it provides the option of using multiple versions as a means of implementing concurrency control.

The write/update algorithm for object blocks may now be stated (The kernel portion of the algorithm is presented in appendix D):

1. Kernel sends a write request to all sites (in parallel) at which a replicate is stored.
2. Sites create a temporary version of the object to be modified and responds to the kernel.

3. Kernel sends block(s) to be written in parallel to all storage sites.
4. SS writes blocks to current version of object.
5. Kernel sends commit message to all sites.
6. Sites commit by writing all modified pages of the current version to storage and deleting the temporary version.
7. Sites respond with commit success plus a checksum of the new current version of the object.
8. Kernel increments version ID of each replicate in the GODI.
9. If continue, then go to step 1; else, stop.

Note that the commit mentioned above is simply an indication that the writes up to this point have been completed successfully; its semantics are much simpler than that associated with transaction mechanisms which are assumed to exist above this level.

In the algorithm presented above, there are several places where the OMS must make a decision on the correct progress of the write operation. For example, if at step 2 all sites do not respond to the write request within a reasonable amount of time, the OMS might decide that the site is unreachable, that is, failed. If repeated probes of the site indicate that the site is indeed unreachable, the OMS could either abort the write or migrate a copy of the object from an existing site to a new site (thereby preserving the availability property of the system) and allow the write to continue. This issue of what action is to be taken when a site fails is covered in more detail in a later section. At step 7, after a commit request, each site responds with a success code and a checksum of the version. Either all sites are in agreement, or some are not. If some sites are not in agreement, the OMS must make a decision on whether to abort the writes up to that point by rolling back to the temporary version, or attempt to update the disagreeing version(s) and continue. This depends upon the frequency of commits and possibly the size of the object (it

might be less costly to rollback and redo the writes to a very large object than to migrate a new copy).

The performance of the algorithm is greatly influenced by the amount of parallelism that can be exploited. Since all block writes are done to independent sites in parallel, the time to update a replicated object is not much more than the time to update a non-replicated one. A performance boost is also gained by not requiring an acknowledgement from each site on every block write; therefore, the kernel can send blocks as fast as permitted by the communication subsystem and the ability to buffer blocks at the SS - up to the commit point where a response is required.

7.4.2 Read Access

The external consistency of the read operation requires that when a read operation is performed, either the read is successful or nothing is returned. Unlike the write operation however, since the read operation is nondestructive, it is only necessary to verify that the information returned is correct. The internal consistency requirements of the read operation, given that the object is replicated, determines the success or failure of the operation. In view of our assumption about the failure properties of storage sites, and an additional assumption of low-level error checking and correction mechanisms at the block level, it is possible to satisfy the external consistency constraint at the block level without requiring majority consensus on each block read.

The strategy is as follows. Whenever an object block is to be read, the kernel chooses a storage site from which to read the block that will result in the best response time. If the block is read successfully (based on low-level

checking mechanisms), then it is assumed to be read correctly. However, as an aid in fault detection and as an optimization, the OMS interleaves the reading of subsequent blocks, if any, between the replicates. Interleaving block reads from different sites allows the reads to progress in parallel for the number of sites on which replicates are stored. The performance of the algorithm can still be improved by prefetching blocks; if the block access pattern is sequential, then performance will definitely be improved -- this may not be the case with random block access however. If a block is read incorrectly from a site, then an alternate site is chosen and a fault recovery routine is invoked (it is known that the block was correctly written to begin with).

The success of this simple read algorithm depends heavily on two things. First, the well defined failure properties of storage sites; if a site fails, it must fail in such a way that any information sent from the site can be determined correct (or incorrect) with high probability. Second, its failure latency must be extremely short with respect to its mean utilization time. This will limit the amount of corrupted information that is handled by the site.

The following sequence of steps is illustrative of the sequence of operations that are likely to be performed in the system when a user requests information to be read from an object stored in the Internetwork.

1. User sends read request for an object.
2. Request is processed by the LAN interface mechanisms in the kernel.
3. Request is sent to the user's process (or to a default service process) running in the kernel.
4. Process makes call to kernel to access the object for which a capability is presented.
5. Kernel determines if object is active by hashing UID into the ICDI. If the object is not in core, use the UID to hash into the GODI to locate the storage site(s) and go to step 13.

6. Check the ICDI block list to see if the requested block is in core. If it is not, go to step 11.
7. Access the block.
8. Pass the address of the block to the requesting process.
9. Send the block to the user via the I/O interface.
10. Repeat steps 4 through 9 for as many blocks as the user has requested and stop.
11. The block is not in core. Check to see if the block is in the RAM buffer. If it is not go to step 13.
12. Read the block and return it to the kernel. Go to step 17.
13. Send a request to a storage site for the requested block.
14. Site receives message and decodes it.
15. Site locates block from LODI entry.
16. Site reads block and sends to kernel as a message.
17. Kernel I/O module receives block and passes address in core to object module.
18. Kernel updates ICDI (and activates the capability for the object if this is the first block access). Go to step 7.

Again, the performance of this operation is strongly influenced by the hit ratios of the RAM buffer and the ICDI. For good performance, these should be high. The rate at which blocks can be returned to the user (assuming that the user has requested more than a single block) is dependent on the LAN interfacing the hosts to FTSS -- if it is being heavily utilized, then its throughput can be seriously impaired (again assuming a CSMA-CD access mechanism). Of course, the performance of FTSS itself will affect throughput, depending on the number of concurrent accesses in effect. These and other performance issues will be discussed in chapter 9.

7.5 Migration

It has been suggested by several authors that migration is a useful concept for maintaining spatial and temporal efficiency in online storage systems. There have been a number of recent studies performed, either to determine the efficacy of file migration policies, or to develop file migration algorithms [Lawr82, Smit81]. This thesis does not study either migration policies or

algorithms. Instead, it provides a mechanism whereby arbitrary policies or algorithms can be developed on a per-user basis. This mechanism is used by the OMS to provide either spatial or temporal efficiency on a demand basis. That is, when a file is accessed, it might be migrated if it is determined that a better response time can be achieved if it is stored at a more "convenient" site. The migration of one object might trigger the migration of another if, for example, space for the first must be made available. A natural consequence of migration is to automatically archive objects when they have not been referenced for a predetermined period of time; a number of storage sites in FTSS could be designated as special archive sites for this purpose.

In FTSS, the basic migration mechanism is a site to site copy of a (possibly replicated) object followed by its deletion after the copy has been successfully made. The external consistency constraint is that this operation must be performed atomically. Below we outline an algorithm that achieves this. However, before we do this, let us reflect upon the semantics of migrating a replicated object. It is indeed possible that an object could have a different replication factor after it is migrated. The rationale for this lies in the inherent reliability of storage sites, storage media, or both. It is generally accepted, for example, that magnetic tape or optical storage (write-once) media are much more reliable than magnetic disks for the online storage of information. If either tapes or optical disks are used at designated archive sites, we may be willing to store a smaller number of copies there than when the object is stored on disks. Thus, as an object migrates from a non-archive to an archive site, a smaller number of copies are stored; when it migrates in the other direction, its replication factor is increased accordingly. Another point of interest is: when a

replicated object is migrated, should all its components be migrated? The answer lies in the particular migration policy in effect. Clearly, if an object is being archived, all its components should be migrated to archive sites. On the other hand, if it is being migrated only for spatial efficiency, it might only be necessary to migrate affected components. Recall our earlier discussion; each component of a replicated object is treated independently by the storage system -- only the OMS is aware of the structure of replicated objects. Therefore, if a local decision has to be made about the migration of an object, this can be done by the local site processors, informing the OMS only of their intention to perform the migration.

From the above discussion, one might correctly conclude that the migration algorithm is two-phased. One phase involves the OMS which coordinates global object migration, and the other involves the local site processors which coordinate local object migrations. Whenever an object is to be migrated by the OMS (either from an OMS or a user policy), the OMS simply initiates the action for each source and destination site involved and waits for confirmation from each site. The local site processors are totally responsible for coordinating the transfers among themselves. Each site designated as the source of a "migrant" object is given the destination site and the name of the object to be migrated by the OMS. The local site, in turn, sends a message to the destination site informing it of the imminent migration and waits for an acknowledgement. Once this acknowledgement is received, the source site starts sending file blocks to the destination site (it is assumed that low-level communication protocols will ensure the correct delivery of each block, or report errors as necessary). When all blocks have been received at the destination site and the object

is stably stored, a final acknowledgement is sent to the source site. The source site then informs the OMS of the success of the transfer, waits for a deletion request from the OMS, and finally deletes the object. As a final note, if a local site wishes to initiate a migration, it must coordinate with the OMS so that the OMS can update its GODI structure on the location of the object when the migration is complete. Notice that, unless the OMS is specifically waiting on the migrant object to reach its destination, block accesses can continue since the source copy is not deleted until the migration is complete.

While there is very little data on object reference patterns and lifetimes in local networks, studies of extant time sharing systems (see [Smit81] and [Saty81]) seem to indicate that, if migration algorithms and policies are based on the size and reference patterns of objects, then small and frequently accessed objects (which tend to have short lifetimes) would tend to cluster near the root of the FTSS architecture, while large infrequently accessed files would tend to accumulate in the higher levels of the tree. We feel that this sort of behavior will improve the temporal and spatial efficiency of FTSS; small frequently accessed objects would be quickly accessed from the root, while large objects would occupy higher capacity storage devices and be migrated much less frequently. This behavior also justifies our choice of a tree architecture since it forms a natural sort of hierarchy for objects based on their size and access patterns. We feel that object access patterns and lifetimes in distributed processing systems is a prime candidate for further research.

7.6 Deletion

One of the side effects of capability-based systems is what has been referred to as the *lost object* problem. This refers to the case where an object exists in the system for which there is no capability. A related problem is the case where there are two or more objects that are circularly referenced. The problem is detecting these objects and deleting them.

The primary problem with capability-based systems is to determine exactly when an object can be deleted. Once this problem is solved, deletion is relatively straightforward. In the case of FTSS, determining when an object can be deleted is done in the SMS. Once this is determined, a message to delete the object (or all components of a replicated object) is sent to the appropriate storage sites. When the object has been successfully deleted, the GODI is updated by removing the objects representative structure.

An object can be deleted by an explicit call to the kernel with a capability containing the right to delete, regardless of how many capabilities are still pointing to it. Therefore, we might have a situation where several capabilities are pointing to a nonexistent object. Clearly, it is desirable that when an object is deleted, all capabilities that point to it should be deleted as well. However, given the potentially large number of capabilities that can coexist, any hope of associating pointers with a given object will result in utilization of large amounts of storage; this situation is a direct result of the unconstrained copying of capabilities. Since capabilities can also be explicitly deleted, there could arise a situation where there is an object to which no capability points. Since capabilities are kept in objects, there is no convenient way to detect

these anomalous conditions except to periodically inspect all capabilities in all objects in the system, tracking down their referents.

There are two established techniques for determining when an object can be deleted. One is based on an *ownership* scheme, and the other is based on *reference counts*. In an ownership scheme, the right of deletion is entrusted to some holder of a capability with ownership rights. The primary difficulty of this approach is its difficulty of implementation in a dynamic environment. It could work in a system such as FTSS if the rate of object creation and deletions is relatively low, and given the capability sealing mechanism described in chapter 4. In fact, from an accounting point of view, the ownership scheme is very attractive. A second difficulty is the dangling reference problem described in the previous paragraph. In the reference count scheme, a positive integer is associated with an object (called a *reference count*) which is incremented each time a capability pointing to the object is created, and decremented each time such a capability is destroyed. The object is marked for deletion whenever the count is zero. The reference count scheme suffers from the drawback that it cannot detect circularly referenced objects. In these cases it is normal to associate a parallel garbage collector to detect, among other things, circularly referenced objects [Dijk78, Wulf81, Poll81]. The details of parallel garbage collection are beyond the scope of this thesis.

In FTSS, we would choose a reference count scheme combined with parallel garbage collection. While in a distributed storage system it is difficult to implement reference count schemes which permit recovery by moving storage volumes from one site to another, this does not pose a problem in FTSS because of its centralized nature. Furthermore, the automatic migration

features would tend to collect unreachable or circularly reachable objects on archive sites, and therefore out of the active portion of the system, at the expense of the storage required to keep these objects indefinitely. However, it is assumed that archive storage is cheap.

7.7 Recovery

The algorithms discussed previously in this chapter were presented in what was assumed to be a fault-free environment. They gain their simplicity and high performance from an assumption that the underlying hardware and software mechanisms are both highly reliable and highly available. In the next chapter, we present the implementation details of a hardware architecture that attempts to meet these requirements. The intent of this section is to first point out the expected failure modes in FTSS, how they are detected, and how one might recover from them.

A taxonomy of *failures*, *errors*, and *faults* has recently been proposed by Avižienis and Kelly [Avi84]. We share their view that a failure occurs when a user perceives that a resource has ceased to deliver the expected service, an error occurs when some part of a resource assumes an unexpected state, and the cause of the failure or error is a fault. Notice that a fault can cause an error but no failure -- this is one of the goals of fault-tolerance. We assume, for the sake of discussion, that faults are restricted to physical permanent or transient faults; we specifically do not consider design or specification faults in either hardware or software. For the present time we also assume that the root node of the architecture is perfectly reliable, but failures can occur elsewhere. The class of failures that we are interested in are site, device, link, media, and

host failures. We discuss each below.

7.7.1 Recovery Management

Recovery management in FTSS is hierarchical. At the level of the SMS, there exists a Global Recovery Manager (GRM) which supervises all recovery actions in the system. Below this is a set of Local Recovery Managers (LRM) that perform local recovery when possible, reporting to the GRM when a higher-level decision is necessitated. There is a LRM at each site in the system.¹ When a fault occurs at a site and is detected, local recovery will be attempted. If this is unsuccessful, then the GRM is informed. It is, however, possible that a site could fail in such a way that it cannot report failures to the GRM. In such cases other sites must detect the failure and report to the GRM; this is discussed in more detail below and in the next chapter.

7.7.2 Site Failures

The consequence of a site failure is the inaccessibility of information at the site. This type of failure is normally detectable by failure to establish communication with the site (the details are elided until the next chapter). The normal way to avoid unavailability in this event is to ensure that the information stored at the site is accessible elsewhere. This is the view taken in FTSS where, by default, every object stored in the system has a replication factor of at least two. Furthermore, two or more replicates of the same object are never stored at the same site. Given this scenario, the action of recovery is to restore

¹The root node is special in the sense that although it implements the SMS, and therefore the GRM, it will also perform local recovery among its components before reporting unresolvable errors (failures) to the GRM. Of course, it has to be designed in such a way that the probability of losing the services of the GRM is acceptably small.

the information at the site to an online status in a timely manner (we do not want the information to be offline for too long since a failure of another site during this time can lead to information availability).

There are two possibilities with respect to the information stored at the failed site. 1) Some or all of the information has been irretrievably lost, or 2) the site has been safely shut down, preserving all the information stored there up to the time of the failure (we assume that any object which was being written at the time of the failure is in an inconsistent state). In either case, the GRM must decide on a recovery action once it is informed of the failure. There are two attitudes that the GRM can take: 1) Do nothing in the knowledge that the site will return to an online status shortly, or 2) take immediate action in the knowledge that the site is not expected to return to an online status in the expected time before a next site failure could occur. If we assume the first attitude, then a suitable recovery action is to invalidate the copy of the object stored at the failed site if other copies are updated (recall that any object copy can be read). This is handled via the version mechanism of the write algorithm discussed previously. Whenever an object copy is written and committed, the OMS associates a version ID in the GODI structure with each replicate to indicate the most recent version. Out of date versions are easily identified by comparing their version ID with the current version ID. When the site returns online, the GRM updates out of date versions (after scanning the GODI) by copying the new latest versions to the site. If we take the second attitude, then a suitable recovery action is to either physically move storage volumes from the failed site to a good site and then updating the GODI, or locate a copy of every object stored at the failed site and make a new copy at a good site. Either solu-

tion would be costly in terms of the time it would take in making new copies and updating the local and global directories. Furthermore, the need to locate a copy of each object when a site fails implies that either the site's LODI must be replicated, or the GODI must be searched to locate each object-pointer to the failed site. Since the LODI is not an object under control of the SMS, its replication would have to be handled strictly locally between the storage sites. The goal, therefore, in FTSS is to design the system in such a way that when sites fail, the first attitude will predominate. If such is the case, searching the GODI for affected objects is tolerable.

7.7.3 Device Failures

It is assumed that each storage site manages one or more devices. A device failure, catastrophic or otherwise, renders all the information stored on the device inaccessible. This condition is detected by mechanisms at the site and is serviced by the LRM. Recovery is handled in a way similar to a site failure. If the OMS tries to access an inaccessible object copy, it will receive a message from the site to that effect. Again, the version mechanism would be used to keep track of which copies have not been modified. The LRM may attempt recovery by requesting an updated copy from another site (by sending an appropriate message to the OMS), or it may simply wait until the device returns online. The severity of any damage to the information stored on the device will determine which action should be taken. In any case, the LRM is responsible for maintaining the integrity of the good information at the site when a failure is detected.

7.7.4 Link Failures

A link failure implies that a site cannot communicate with its neighbor over the affected link. Transient link failures are handled at the level of the communication protocols, a topic which is discussed more fully in the next chapter. The FTSS architecture is designed in such a way that isolated permanent link failures should not result in a node becoming unreachable. There is no particular recovery action performed by the LRM except to update local tables which indicate the site's connectivity; no messages are sent over the failed link until it has been repaired. Thus, there is no automatic recovery mechanism.

7.7.5 Media Failures

Media failures, whether transient or permanent, will affect a portion of the information stored on a device. Transient media failures are detected using any of several standard mechanisms such as block checksums; permanent failures are detected by repeated errors when reading from the affected media. The local recovery action in the case of permanent failures is to recopy the object from another site by coordination through the OMS. For example, if a media failure is detected, the LRM could send a message to the OMS requesting the name of a site on which a copy is stored. The LRM would then contact the site requesting a copy of the object. This assumes that there is enough capacity at the requesting site to store the object. If this is not the case, then the GRM will request that a new copy of the object be created on another site. The advantage of letting the local site processors coordinate intersite copies among themselves is that it reduces communication overhead with the TL.

There is one special case of media failure that requires attention. Information which is stored locally at a site, such as the LODI and other information that the processor needs to maintain the integrity of its local environs, are not objects, and are therefore not under control of the SMS. This information must be stored in a way such that media failures (bad blocks due to head crashes on disks, or spontaneous delay) will not affect its availability. There are two ways in which this information can be protected -- both require that it be replicated. The first is to replicate the information at another site, the second is to stably store¹ it at the current site. The advantage of the first method is that the information is available if the entire site fails; this was discussed in the previous section. Its main disadvantage is that we must now make every effort to make sure that the replicates are consistent, or at least nearly so -- if blocks are propagated from the primary site to the replicates as each block is modified and written to storage, then this condition can be achieved. However, this is at the expense of some intersite communication bandwidth. The second method has the advantages that if a media failure occurs, duplicate blocks can be accessed more rapidly than if they had to be retrieved from another site. If the local site fails, however, all the saved information is lost. In view of the need for a mechanism that will both support site as well as media failures, replication at another site appears to be the better choice.

¹Using Lampson's definition of stable storage [Lamp79], with the augmentation that each replicated block be stored on a different device to guard against catastrophic device failures.

7.7.6 Host Failures

A host failure occurs whenever any fault external to FTSS occurs while a service is being performed for a given host. This can include a host operating system crash, host-FTSS communication failure, or a failure in the host hardware. Such failures will not usually affect the consistency or availability of information in FTSS, but any recovery action after such failures should clean up any partially completed actions. For example, if a host crashes while an object is being written into FTSS, the effects of the write should either be undone, or the host should be allowed to complete the writes when it recovers. In general, different users will have different policies on what should be done with partial reads and writes. Therefore, FTSS allows each user to specify a recovery policy as a part of its subsystem definition. This policy is automatically invoked by the GRM whenever host failures are detected.

7.8 Summary

In this chapter we have looked at several important mechanisms needed to manage objects in FTSS. The way in which each is implemented, in terms of the algorithms discussed, imposes requirements on the implementation of the architecture. In the next chapter we propose an implementation scheme for FTSS which permits these algorithms to meet their design assumptions. While this is only one of several implementation approaches, we feel it is the right approach, given current technology and the state of the art in fault-tolerant design.

CHAPTER 8

IMPLEMENTATION OF FTSS

In the previous chapters, we have looked at various issues relating to the design of a centralized information storage system. In chapter 2 we presented a basic system architecture which was based on the needs discussed in our introductory chapter. This "strawman" architecture served as a brassboard on which we developed a software base (SMS) which meets many of the functional requirements of the system. In the previous chapter we presented a conceptual overview of the FTSS architecture which is based on the architecture developed in chapter 6. We also developed a set of mechanisms needed to manage information in the Internetwork which imposed constraints on the implementation approach. In this chapter we review some of our decisions on the architecture of the basic system and propose an implementation of the FTSS architecture which offers better support for the SMS model while providing higher reliability, higher availability, and an improved level of performance.

8.1 Introduction

The need for a centralized storage system facility based on backend and file server principles was established in chapter 1. In succeeding chapters, we assumed this fact and looked at various issues relating to the implementation of such a system; some were related to software, others to hardware. We also established the importance of high reliability, high availability, and high

performance of such a system, not only because of its potential utility by a diverse community of users, but also because of its very architecture. In this chapter we take a closer look at how one might implement the system in order to meet its requirements.

As shown in chapter 5, high information availability can be achieved by replicating information such that each replicate has an independent probability of failure. In chapter 6, we showed the architecture of a storage system which provides high reliability and arbitrary extensibility while permitting high information availability. In chapter 2, we discussed the functional requirements of the system and developed the basis of an object/capability-based storage management software system which meets various system requirements. In chapter 7 we discussed several mechanisms that imposed additional constraints on the architecture. What we intend to do in this chapter is to bring together all these mechanisms under a single hardware architecture. In so doing, we extend the basic architecture presented in chapter 3 by providing features which will enhance its reliability, availability, and performance.

8.2 The Basic System

The reader will recall from chapter 2 (see Fig. 3.1) that the basic system consists of three functional blocks: a set of host processors, a storage system made up of a server CPU and a set of storage devices, and a high-bandwidth local network which interconnects the host processors with one another and with the storage system. This system has several drawbacks which will lead to unreliability, unavailability, and low performance. We explore these in the next paragraphs.

First we shall consider the local network. In the basic system, this local network is assumed to be a simplex (that is, non-redundant) high-bandwidth communication link of the ethernet class; approximately 10 million bits per second data transfer rate, and based on the well known Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [Metc80]. This "broadcast bus" is a bottleneck for two reasons. First, since the bus is not replicated, a failure here will seriously disrupt communication between the system components. Although ethernet systems are known to be quite reliable, with respect to the communication medium, bus failures must still be considered in the overall reliability of the system. Normally, the bus is interfaced with the various components through NI's that are physically isolated from the bus, so that they can be connected or disconnected (including disconnections due to failures) without disrupting bus communications. A reliable design must consider the consequences of failures either in the bus itself, or in the interfaces that communicate with the system components. Second, since the bus represents a simplex communication path, the bus bandwidth can be easily exceeded if not carefully accounted for in the design. The fact that the host processors must now perform most of their storage system I/O over the bus, in addition to normal host-to-host communication, puts additional requirements on bus bandwidth. In a normal ethernet-type system, it has been observed that even with a 10 MHz bandwidth, the average throughput is on the order of 2 MHz when the overhead of various layered communication protocols and the network interfaces are taken into account [Pope81]. From a performance point of view, therefore, it is desirable to design the local network so that it is bandwidth-extensible as the demand for throughput increases, either due to a larger number of host processors, or increased traffic with the storage system

due to higher levels of system activity.

Next, we consider the storage system. In the basic system, it was assumed that the server CPU was a simplex high-performance superminicomputer capable of implementing the SMS. There are several potential problems with a simplex CPU (recall the discussion in chapter 1), not the least of which is the fact that a failure here renders the storage system useless. Such single-point failure mechanisms are to be avoided at all costs. Although the SMS was designed to be robust to failures (via the object model and the transaction mechanism), hardware support is required in order to permit recovery when such failure processes manifest themselves. A simplex CPU is not modular, leading to great trepidation and expense when the system's capabilities must be improved. A proper design approach here is to modularize the server CPU and take advantage of the cost benefits inherent in the use of low-cost off-the-shelf components.

It was also assumed in the storage system that objects were stored on rotating magnetic media (disks) that were each attached to a high-speed serial bus via an intelligent interface processor, the bus also interfacing with the server CPU. Thus, all communication between the server CPU and the disks must take place over this (simplex) communication path. The arguments presented above about the simplex bus for the local network also apply here. While there was no logical reason to assume that the storage devices were distributed (they could easily have been assumed to be connected directly to the server, as is currently done in such systems), we did anticipate the need for extensibility in a manner which was somewhat independent of the server. Also, we did not wish to complicate the design of the object manager in the SMS.

We decided it would be best, in any extensible design, to offload as much as possible of the management mechanisms from the SMS and distribute it among the devices themselves. The correctness of this decision is borne out in part by the resulting simplicity of the object manager, and by the discussion of extensible storage in chapter 6. The primary problem with the basic storage system architecture is the fact that the serial bus could become a bottleneck under heavy usage. A good design approach is to take advantage of the usage pattern associated with mass storage systems (see, for example [Smit81]) by building the system as a hierarchy in which device speed decreases with the depth of the hierarchy, and information is automatically migrated within the hierarchy as a function of object reference patterns.

There is not much that can be done to alleviate the problems associated with host processors. In fact, we make no assumptions whatever about their reliability or availability. They may fail, go offline, or exhibit any of the idiosyncrasies associated with simplex host processors. The very purpose of FTSS is to provide the reliability for semi-permanently stored information which is lacking in these systems. We are, however, prepared to make some modest assumptions about communication interfaces between the host processors and the local network. For example, in the interests of reliability, we can insist that whatever reliability mechanism is used to increase the reliability of the local network should be extended to the interface between the network and the processor.

In the rest of this chapter we describe an implementation of FTSS which attempts to solve the problems discussed above.

8.3 FTSS Hardware

The FTSS hardware implementation is geared towards high reliability. As discussed in the previous chapter, it is assumed that the Top Level hardware is extremely reliable since it is crucial to the operation of the system. Similarly, the Internetwork is assumed to have a certain level of reliability; the site processors are assumed to be able to perform reads and writes correctly and manage information stored locally at the site without external monitoring, and it is assumed that the communication system between sites will deliver messages without error with very high probability. In this section we propose an implementation which will meet these requirements.

8.3.1 The Top Level

Logically, the TL is structured as a set of modules interconnected with a *Centralized* or *Short* bus. These modules represent *Processors* (PM)'s, *Memories* (MM)'s, *Network Interfaces* (NIM)'s, and *Internetwork Interfaces* (IM)'s (see Fig. 8.1). The short bus logically consists of a *Main Bus* (MB) and a high speed *I/O Bus* (IOB); each of these buses is made up of other buses, which we will detail in a later section. The MB connects all modules, whereas the IOB is not directly addressable by the PM's. As can be seen in the figure, there are also connections (buses in their own right) between the NIM's and the (external) network (to which the host processors are attached), and between the IM's and the Internetwork. This arrangement of modules and buses provides the basis upon which the detailed architecture rests. Before we progress to these details however, let us digress for a moment to briefly define the rationale for this logical organization.

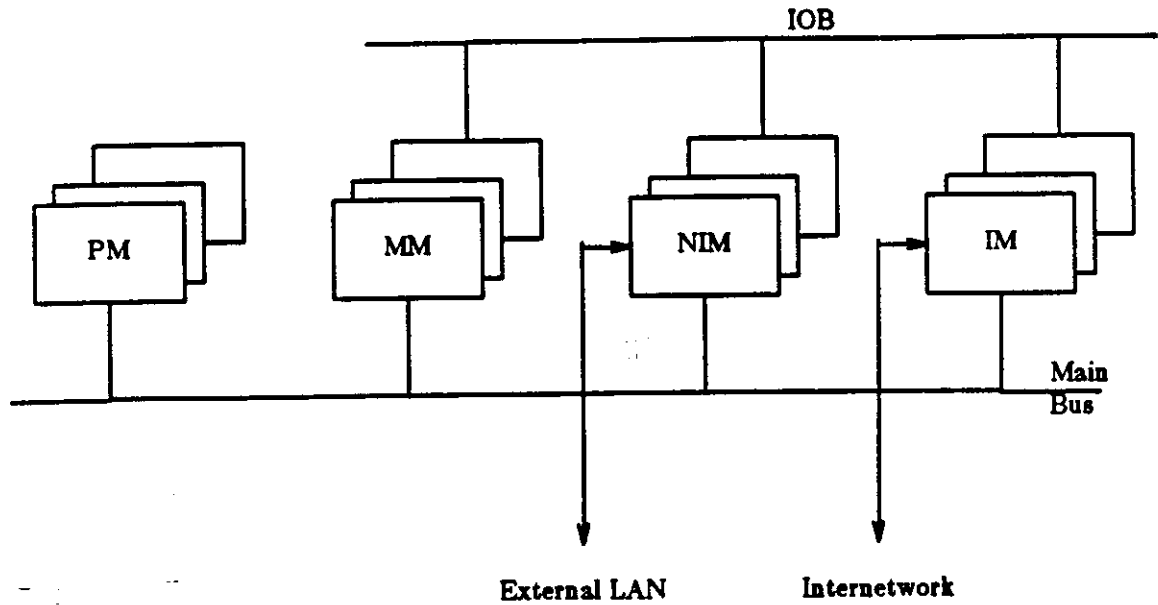


Figure 8.1: Logical Organization of the Top Level

The short bus architecture was chosen because of the need for relatively high performance at this level. It is anticipated that there will be a fair amount of data traffic between the basic modules during a typical object access. Since several such accesses are progressing concurrently, using the same data paths, it is important to provide some parallelism of functions in order to obtain higher throughput. The short bus meets these requirements by providing parallel data paths over which the modules might communicate. If one remembers that the Top Level implements the root of FTSS, it is immediately clear why attention to performance must be given here.¹ Despite the use of parallel data paths on the short bus, we still anticipate a bottleneck in performance if both high-speed data traffic representing object blocks, and low speed communication representing message traffic between modules, must take place on the

¹If it is still unclear, consider that all data traffic into and out of FTSS must traverse the root.

same bus. Therefore, we have partitioned the centralized bus into the MB (which services intermodule communications), and the IOB (which services high-speed block transfers between some of the modules). It is possible to take this approach if one can maintain a logical separation between pure object traffic and intermodule traffic. This is anticipated since it is expected that a significant percentage of traffic on the buses will represent pure object transfers (for example, a significant amount of the time, a user simply wants to read an object without performing any modifications on it).

The processing subsystem is implemented as a set of modules for a number of reasons. First, modularization is a first step to extensibility -- an important metric for FTSS. Each module is an independent computer. Second, from a reliability viewpoint, this kind of modularization provides natural fault-containment regions during system operations; by allowing faults to be contained within the affected module, a higher level of reliability can be achieved. Third, availability is enhanced since, if a module fails, it is possible to isolate the failed module from the system and perform repairs on it without unduly affecting the operation of other modules. Fourth, it allows one to practice the theory of isolation [Denn76]. By running processes in individual modules, they are not only logically isolated from errors, but physically as well, leading to an overall more secure system. There are additional benefits that result from a physical context, such as better power distribution, better cooling properties, and so on, which tend to reinforce our idea that modularization is indeed a good way to proceed.

The memory modules on the main bus represent a high-speed block-addressable buffer memory for *caching* object blocks that have been recently referenced. If this buffer is implemented using electronic components, as we anticipate it will be, then one can imagine it as an electronic disk which provides a performance several orders of magnitude higher than a conventional rotating media magnetic store (or, simply, a disk storage unit). At any given instant of time, a subset of the total information content of the Internetwork is available in the buffer. Note that it is a proper subset because of its volatility in the presence of power failures. The need for this buffer is again related to the anticipated performance bottlenecks of FTSS. By buffering blocks in this way, it is hoped that, on the average, access requests for object blocks can be satisfied from the buffer without requiring an additional access to the Internetwork. By modularizing the buffer, its design becomes simpler, it is easily extended as the need arises, and it presents better physical characteristics than a large monolithic design. In addition, the reliability and availability reasons given for the processor modules equally apply here.

The Network Interface Modules are modularized for two main reasons. First, it is assumed that the external Local Area Network (LAN) is modularized for reliability and availability reasons.¹ Therefore, there is at least one NIM for each bus of the LAN. Second, we replicate since this is clearly a critical path in the FTSS architecture. Failure to establish a line of communication with the external LAN is the sort of disaster which FTSS is designed to combat in the first place. We gain an additional benefit here since we can make use of the fact that, given that there are several correctly operating NI's, one can perform

¹To not assume this would put a fairly heavy responsibility on the reliability of the LAN since an unrecoverable failure in the LAN would totally isolate FTSS.

multiple simultaneous conversations with the LAN, thereby improving performance.

Finally, the Internetwork Interfaces are modularized so that failures can be isolated and quickly handled without disrupting system availability. These modules provide the necessary lines of communication between the TL and the Internetwork. They must respond quickly and accurately when called upon and, above all, they must be there when required. Modularization, then, is a primary contributor to high system availability. Since one is virtually forced to use off-the-shelf components for device controllers, high reliability can only be gained through redundancy -- the approach we take.

8.3.1.1 Bus Architecture

The FTSS TL bus architecture is driven by several factors:

- The need for an effective interprocess and intercomputer communication mechanism.
- The need for low latency, both from end to end on the bus and process to process via the bus.
- The need for high bus reliability and availability.
- The need for high bus utilization, even when the offered traffic is higher than the bus bandwidth.
- The need for a low Bit Error Rate (BER) on the bus.
- The need for high throughput in order to support a large number of

processes.

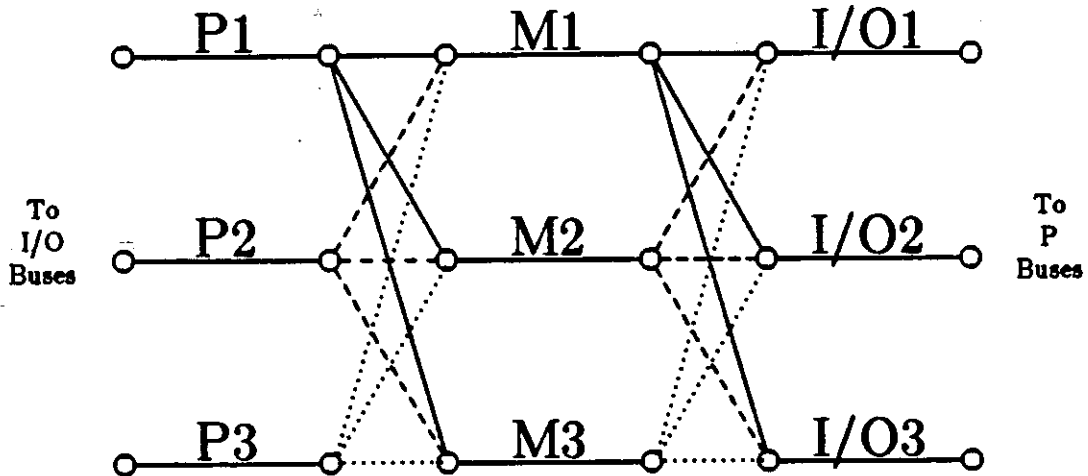
- The need for an effective communication mechanism with high-speed I/O devices.

In FTSS, the PM is structured in such a way that processors must communicate among themselves, and they must communicate with I/O devices occasionally. This requires that the busing structure be able to support these two types of communication concurrently and efficiently. There are two ways to solve this problem. 1) Build a monolithic bus which supports both types of communication, and 2) build a distributed bus where a separate section of the bus is dedicated for each kind of traffic. The first approach has the advantage that it is easier to build; the second has the advantage that system performance is improved since the bus can be optimized for each kind of traffic. FTSS takes the second approach by providing a distributed bus with one section optimized for low-speed traffic (interprocessor communication), and the other optimized for high-speed data transfers (block to block).

The availability requirement is a crucial one with respect to bus architecture. It is very desirable that should a module or the bus itself fail, it be possible to isolate the failed components and perform repairs, *without affecting the operational status of the system*. For example, if it is determined that a processor module has failed, it should be possible to remove the failed module, repair it, and return it to service without disrupting system service.¹ There are two possible solutions. First, the busing structure can be distributed with a fine grain so that only a very small number of modules are connected to any

¹It is permissible to have a degraded mode of operation during the repair interval, but service should not be totally disrupted.

given section. Thus, when a module fails, the entire bus section can be isolated and repairs performed. There are two main drawbacks to this scheme. 1) In order to permit alternate paths when failures occur, each bus section must be connected to several others. Consider, for example, Fig. 8.2 which shows a distributed bus consisting of 3 each of processor, memory, and I/O buses.



$P_i \rightarrow$ i th Processor Bus
 $M_i \rightarrow$ i th Memory Bus
 $I/O_i \rightarrow$ i th Input/Output Bus

Figure 8.2: A Fault-Tolerant Busing Structure Based on Distributed Buses

In order to provide a high degree of fault-tolerance, all three buses in each group must be connected to every bus in the other groups. In this case, every bus is connected to 6 other buses for a total of total of 27 interbus connections. Since connections are always a weak link in any design, this is not considered to be a workable strategy, particularly if the number of bus sections is large. Of course, one can compromise and reduce the number of connections at the price of lower fault-tolerance capabilities. 2) Depending on the number of modules assigned to each bus section, all modules assigned to a bus section are inoperative when either a module or the bus fails. For example, if 2 processor

modules are assigned to a processor bus, then a failure in one module requires that both modules be taken offline to repair the failed module. Despite these problems, this scheme is workable when the number of modules in the system is small (and expected to remain small over the lifetime of the system). It has been successfully employed in the design of the Pluribus system [Kats78].

The second solution is to provide a redundant busing structure so that bus failures can be tolerated and attach modules to these redundant buses in such a way that a defective module can be removed (and later replaced) without leading to system downtime. The scheme is illustrated in Fig. 8.3.

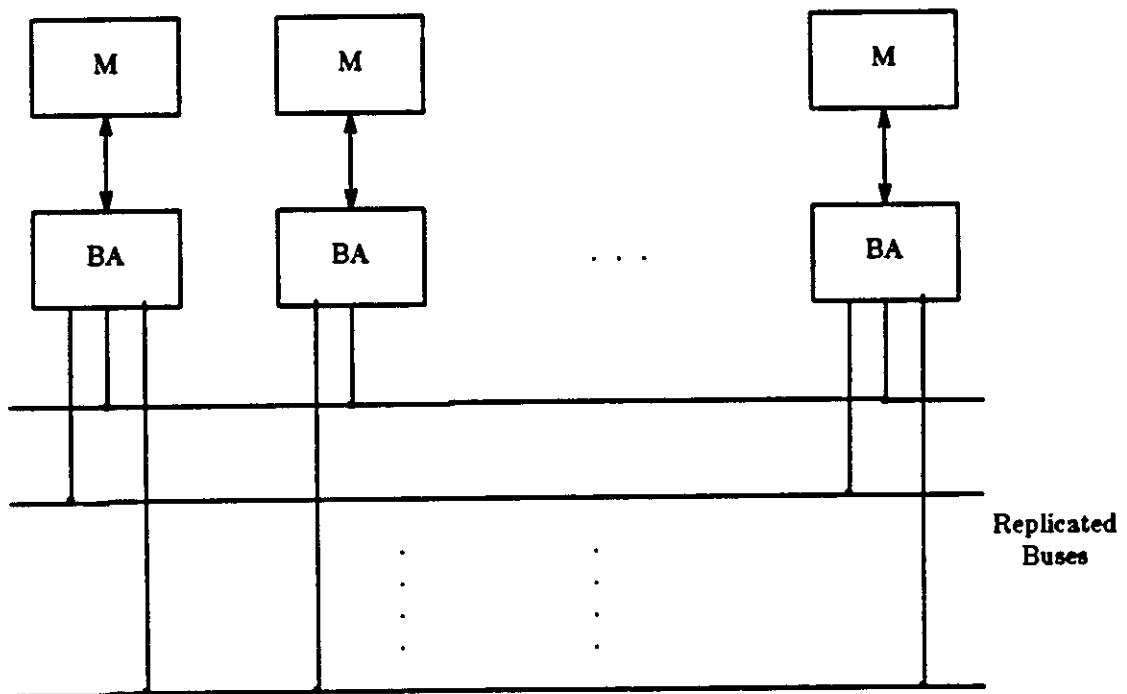


Figure 8.3: A Fault-Tolerant Bus Architecture Based on Monolithic Buses

The bus is replicated (but not distributed), and modules are attached to as many buses as needed by using special bus adapters between the modules and the buses. The bus adapters have the feature that they can be made to

transmit or receive on any bus, and they may be disconnected transparently from any bus without affect connections with another bus. The primary drawbacks with this scheme are: 1) There is now a bus adapter in series with a module; a bus adapter failure is effectively a module failure. Note that if a bus adapter fails, it is possible to simply unplug the module and reattach it to the network with a good adapter. 2) Bus adapters must not be permitted to fail in ways which would prevent the use of all the good buses in the system (this is largely a technological issue, but an important one nevertheless). This scheme has been used in a number of designs, primarily with serial replicated buses, and non-redundant parallel buses [Renn80, Ahuj83].

The FTSS bus architecture consists of two buses: A parallel replicated Main Bus which is optimized for interprocessor message traffic, and a parallel replicated I/O Bus which is optimized for high-speed data traffic. The detailed architecture of each bus is considered separately below.

8.3.1.1.1 The Main Bus Architecture

The FTSS main bus is a set of replicated *component main buses*. Each component main bus is an independent busing structure which is physically and electrically isolated from the other identical components. Every bus adapter is connected with each component bus, and may use any of them to establish communication with another adapter. A faulty bus is identified by a bus adapter after repeated attempts to communicate over that bus is always unsuccessful. Each bus adapter independently makes the decision whether a bus is faulty or not since the problem might be in the bus adapter itself (this prevents a faulty bus adapter from incorrectly preventing other good adapters

from using a perfectly good bus).

Each component main bus is further divided into three independent buses: A *message/data* bus, a *contention* bus, and a *control* bus. All three buses are synchronous, operating from a common bus clock that provides *bus slots* during which transmissions take place. One of the salient properties of the main bus is the fact that the end-to-end delay is much shorter than the time it takes to transmit one bit of information; it is this property which makes centralized buses attractive since it is usually possible to provide high bus utilization and short message delay, even when the offered load is very high with respect to the bus bandwidth. The message/data bus is a parallel bus that transmits n bits of information in one bus cycle. A bus cycle corresponds exactly with one bus clock cycle; however, several processor operations can be performed in a typical bus cycle by using a multiple-phase clock. The contention bus is used to resolve conflicts between modules that attempt to use the bus during a bus cycle. A separate contention bus is useful here since it permits contention to take place in parallel with message/data transfers. Modules normally contend for the bus during a bus cycle for transmission during the *next* bus cycle. Contention is guaranteed to be resolved in less than one bus cycle for all modules connected to the bus, up to some limit imposed by the physical characteristics of the bus. The control bus can be used for a variety of functions. For example, it can serve as a message transmission path whereby the bus adapters can communicate directly without using message bandwidth on the message/data bus (there is no possibility of using the contention bus for this purpose in our design). One critical use of the control bus is to assist in low-level flow control for message packets sent over the message/data bus. An

acknowledge (ACK) or negative acknowledge (NAK) signal can be sent over the control bus from a receiving to a sending bus adapter in much less than a bus cycle. Thus, a sending bus adapter can know if its packet was received correctly by interrogating the control bus for a ACK/NAK before the bus cycle is complete; if the packet was NAK'ed (a null response or an explicit NAK), the module can contend for the bus and retransmit it again. The control bus is also useful for forcing module disconnections while a bus is being repaired; this can be accomplished by broadcasting an appropriate message on the control bus. The main bus architecture is now shown in Fig. 8.4.

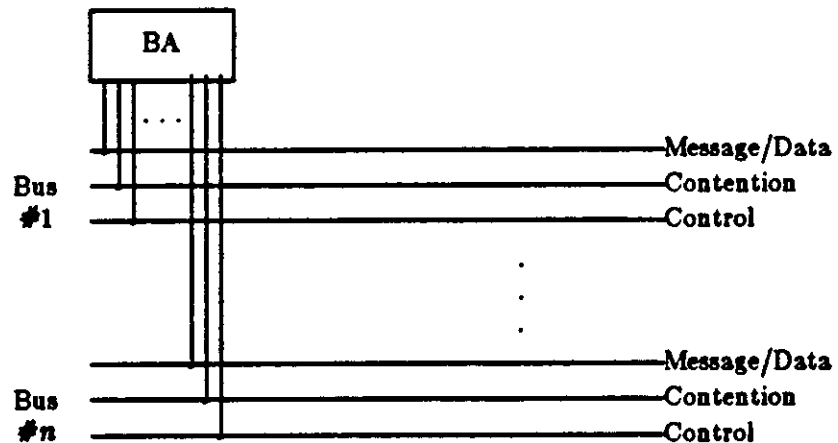


Figure 8.4: FTSS Main Bus Architecture

One of the attributes of a short bus (as defined above) is the fact that the media access efficiency is insensitive to the packet size used to transmit information over the bus [Acam84]. This is in contrast with media access schemes such as CSMA/CD and token passing where the media access efficiency decreases with decreasing packet size. One can take advantage of this by sending small packets over the bus in a pipelined manner. In fact, one can

dynamically vary the packet size, sending small packets when necessary, without affecting the bus utilization. While there is no upper limit on packet size in a short bus, it is useful to impose some limit depending on the contention time, and the need to provide fairness for all users of the bus. In FTSS, the upper bound on packet size is determined by the physical size of the packet buffers in each bus adapter. This is kept reasonably small (in the range of 1 to 2 kbytes). The minimum packet size is n , where n is the number of bits that can be transmitted (in parallel) in one bus cycle.

In order to keep the BER low, each word sent on the bus includes a check byte (or checksum) which is sent concurrently with the data bits. The check byte is computed and sent by the transmitting bus adapter; it is independently computed and checked with the transmitted check byte by the receiving bus adapter. If there were any errors that cannot be corrected, the receiving bus adapter informs the transmitting bus adapter of this via a NAK on the control bus. Packet framing is done in the bus adapters, or may optionally be done in the processors. However, each bus adapter has a unique address on the bus. To allow very high bus utilization through pipelining, each word transmitted in parallel on the bus contains the address of the receiving bus adapter. This allows a very fine degree of multiplexing in support of multiple-message transfers. Normally, two or more bus adapters would establish a virtual circuit through some call-setup procedure. Once this circuit has been established, packet transmissions may take place until the entire packet is transmitted. Therefore, although several messages may be simultaneously transmitted over the bus, each bus adapter can handle only a single message transmission at a time.

The bus architecture easily accommodates special operations such as out-of-band signaling, remote initialization of processors, and a message broadcast (to all bus adapters) facility. Signaling can be handled by reserving bus cycles for special transmissions between requesting bus adapters via the control bus. In a similar manner, a bus cycle (or any number of bus cycles) can be used to transmit special broadcast messages by "setting up" the bus adapters via the control bus.

8.3.1.1.2 The I/O Bus Architecture

Logically, the IOB is a constituent of the centralized bus of FTSS Top Level. It is distinguished from the main bus because it is meant to handle a different type of traffic at a much higher speed than can comfortably be accommodated by the main bus. The IOB is designed to primarily support block data traffic between the MM's, NIM's and IM's. The IOB is a synchronous back-plane (bus) that provides addresses, data, and control signals for the attached modules. In principle, it services the memory modules in the RAM Buffer (see below). Modules in the NIM and IM sets may directly address RAM locations in the Buffer during I/O transfers between them. Each module contends for the bus during a bus cycle and, having captured it, holds it for the duration of the transfer of a block before relinquishing it. Therefore, pipelining on the bus is at the block level. Note that, to use the bus, each module must provide the interface logic necessary to generate addresses, and buffer data from or onto the bus as necessary. The bus is independently clocked (from the modules); however, the clock is provided as part of the control bus on the IOB for interface logic purposes.

From an availability point of view, the survival of the IOB is not critical to system operations. Should the IOB fail, modules may transfer their information to the Ram Buffer more slowly by using the main bus (which is better protected). Since FTSS is a repairable system (with a reasonably short repair interval -- perhaps even on-demand maintenance), short periods of outages on the IOB are tolerable. However, in order to provide error control, each word (address or data) transferred on the bus is provided with redundant bits for error detection and correction. Interface logic is responsible for providing the necessary check bits on each word; these are regenerated and checked by each interface that receives the word. Bus errors are reported to an error monitoring process running in the PM's, via the main bus.

8.3.1.2 Processor Modules

Each processor module is an independent self-checking computer. As an aid to both efficiency and fault-tolerance, the modules are arranged in a two-level hierarchy as shown in Fig. 8.5.

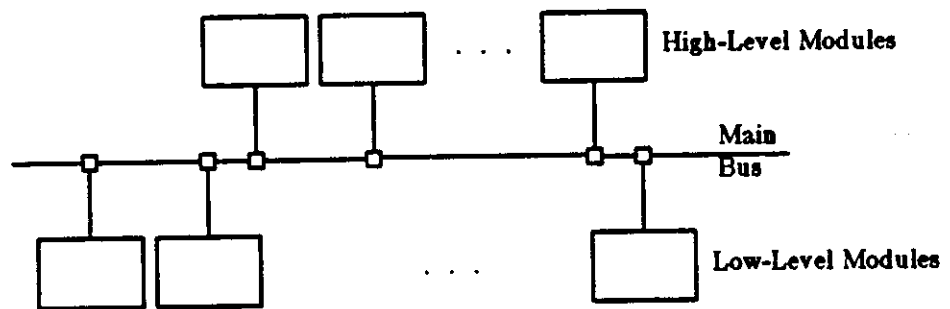


Figure 8.5: Processor Module Hierarchy

The modules are arranged as a set of High Level Modules (HLM), and a set of Low Level Modules (LLM). The HLM's execute critical (or security-related) por-

tions of the SSM, while the LLM's execute non-critical sections. In terms of the SMS model developed in chapter 3, the reader can imagine that the HLM's implement the kernel and the LLM's implement various user-level subsystems. The PM's communicate by using the communication facilities provided by the main bus and the bus adapters.

As mentioned above, each PM is implemented as a self-checking computer with well-defined failure properties. In particular, they are designed so that the PM can detect internal failures through data inconsistencies, and can disable its outputs if it is unable to resolve these inconsistencies. Rennels [Renn80] has shown that by using a few building blocks, easily implementable in VLSI, it is possible to cost-effectively design a self-checking computer that meets our needs, especially if one takes advantage of off-the-shelf microprocessors, memories, and other support devices. Each self-checking PM is configured as shown in Fig. 8.6. The organization consists of duplexed processor units, operated in lockstep, whose address and data paths are continuously compared using morphic self-checking comparators. If an inconsistency is detected, a program rollback is attempted. If this is unsuccessful, then the module decouples its outputs via the bus adapter. Note that, should a processor be unable to decouple its outputs, this can be done from another module by sending a disconnect command to the bus adapter. Unlike Rennels' SCCM, however, it is not possible for a bus adapter to perform Direct Memory Access operations with the PM. This is not a limitation since it is assumed that each PM contains "bootstrap" routines programmed in ROM which can reload a correct processor state should it become corrupted.

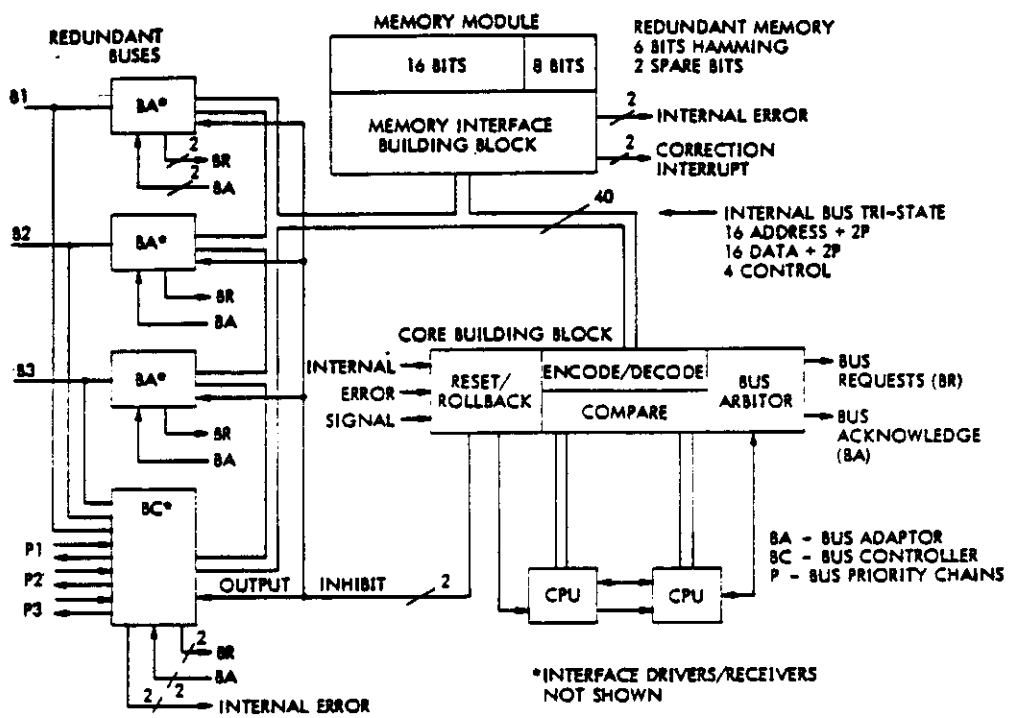


Figure 8.6: Self-checking Processor Module

In order to tolerate processor failures among the HLM's, each HLM operates with a *hot spare*. The hot spare executes exactly the same program as its counterpart; its outputs, however, are used only if the primary fails. Since LLM's execute non-critical processes, it is sufficient to simply restart them on a non-faulty processor when a processor fault affects the execution of a process.

8.3.1.3 RAM Buffer Modules

As previously explained, the RAM Buffer (RB) is largely a block addressable memory that emulates a disk storage unit. Its primary purpose is to improve the performance of the system by acting as a cache for the most recently referenced object blocks from the Internetwork. Logically, the RB is structured as shown in Fig. 8.7.

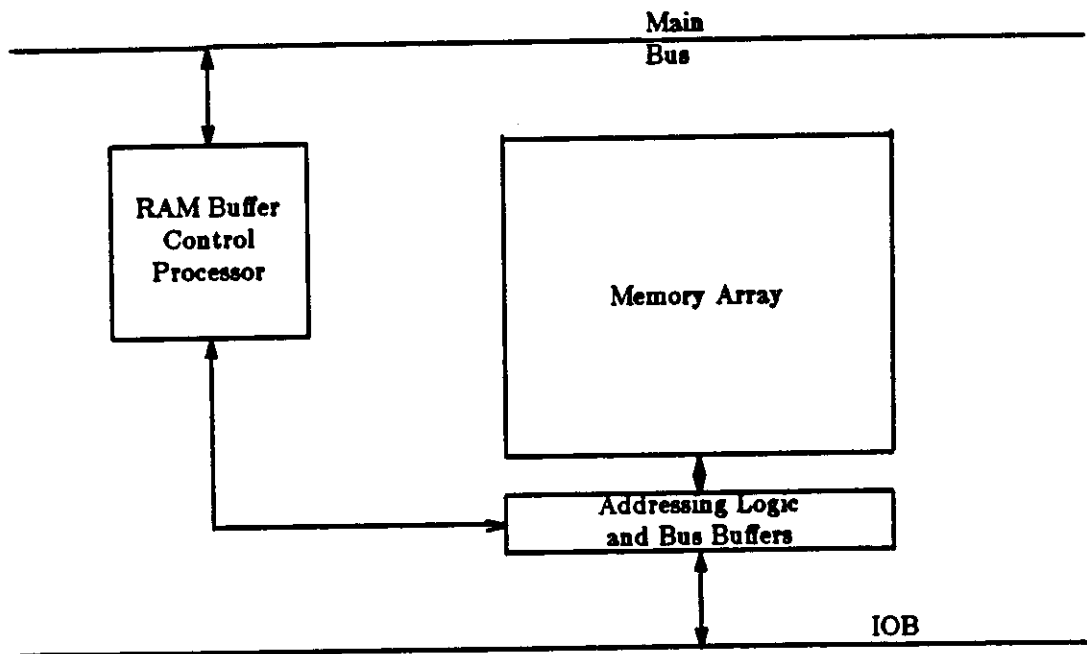


Figure 8.7: Logical Architecture of the RAM Buffer

As can be seen from the figure, the RB consists of a large randomly-accessible memory array which is interfaced to the IOB via addressing logic and bus

buffers. The RB is actually a dual-ported memory since it can also be directly addressed by the RB Control Processor (RBCP). All accesses to the RB must be coordinated by the RBCP. It is interfaced to the rest of the system through the main bus, in a similar manner to all the other modules that interface with this bus.

Since the RB emulates a disk storage unit, information stored in it is in terms of those parameters that are normally associated with disk units. Information is stored in *blocks* and are accessed as such. The information grain is therefore the block. When data is to be written into, or read from the RB, it is done so in blocks. The RBCP actually uses a small portion of the total memory space for its own volatile storage needs, such as maintaining mapping tables for object blocks. The IOB is a means whereby other system modules can rapidly access information in the RB. It allows these modules to directly specify memory locations under control of the RBCP -- this is in direct analogy with the concept of Direct Memory Access (DMA). To illustrate how the IOB is used, let us consider how object blocks are moved between the NIM's and the RB. Recall that the NIM's interface the TL with the external LAN. To make things simple, we will assume that there is a single NIM, a single memory module, and a single RBCP.

Normally, information is sent over the external LAN in packets. We will assume that such a packet is much smaller than the size of a block in the RB. Whenever a data transfer is to be initiated by an external host processor, a message to that effect must first be transmitted to the TL -- this message would be processed (in the PM's). As a part of this processing, the RBCP is notified that an object transfer is about to commence and that space should be

made available in the RB to hold the object blocks. The RBCP would make the necessary arrangements, one of which would be to inform the NIM where in the RB's memory data space has been made available; that is, the address where the first block (and perhaps subsequent blocks) should be stored. Again this is much the same principle used in DMA transfers where the DMA controller is given the starting address in memory and the number of words to transfer. All this "initialization" mechanism is handled by using the communication facilities provided by the main bus; notably, messages and signals. Once initialization is complete, the RBCP signals the NIM to commence the transfers. As packets arrive from the external LAN, they are packed into object blocks (buffers in the NIM). When a block becomes full, it is written into the allocated area of the RB directly by the NIM using the facilities of the IOB. As discussed before, the IOB provides parallel address, data, and control buses which are used to address locations in the RB and then to access data in these locations. When the transfer is complete (or, alternatively, when the NIM needs more RB blocks, the RBCP is signalled via the main bus. Notice that it is possible for the NIM to communicate with the memory array and the RBCP simultaneously, since different buses are being used. Thus, as an optimization, the next block setup can occur in parallel with the current block transfer over the IOB.

As explained in a previous section, the information in the RB is only a subset of the information in the Internetwork. In particular, this means that if information is lost from the RB because of its volatile nature (we assume construction from fast semiconductor devices), it must be replaceable. To allow this, we take the philosophy that any information written into the RB (which represents an object block that is not considered temporary) must also be writ-

ten into the Internetwork. Therefore, in the example above, whenever the NIM write blocks to the RB, this block must also be written to the Internetwork. In order for things to work correctly, information received by the NIM is not ACK'ed until it has been written correctly into the internetwork. There are two ways in which this can be done: 1) Since the IM is also incident on the IOB, it could read blocks directly from the bus if it knew that the information was destined to the RB. In this way, writing to the Internetwork is done transparently. The drawback of this approach is that the IM architecture is unnecessarily complicated by the need for address decoding logic. 2) When all blocks have been transferred from the NIM to the RB, the RBCP can instruct the IM to move the blocks into the Internetwork. Recall that an ACK is not sent until all blocks have been transferred to the Internetwork. However, since the transfer is done usually at a high data transfer rate, the delay should be minimal. As an optimization, the RBCP can perform *partial writes* to the Internetwork as the object blocks are being assembled; thus, if a failure in the RB should occur, only a few blocks need to be retransmitted.

Transfers from the PM's to the RB are handled in a similar manner with the RBCP coordinating block transfers into the RB. Note, however, that all transfers between these two module sets take place over the main bus -- the need for high speed data transfers between the PM's and the RB is much less critical than between the RB and the other system modules.

From an availability point of view, it is desirable that the RB and RBCP be available a high percentage of the the time since their presence impacts the performance of the system. For this reason, it is important that failures here be tolerable and do not lead to downtime. In order to prevent downtime due to

failures in the RB, it is designed as a set of modules, each of which interfaces to the IOB. Information is written into at least two modules so that any single module failure will not result in unavailability. To make things straightforward, each module is *shadowed* by writing information in identical location in both modules.¹ To protect against failures in the RBCP, it is also duplexed with a hot spare both of which operate in parallel. The modular organization is shown in Fig. 8.8. Notice that each RBCP can access any of the modules.

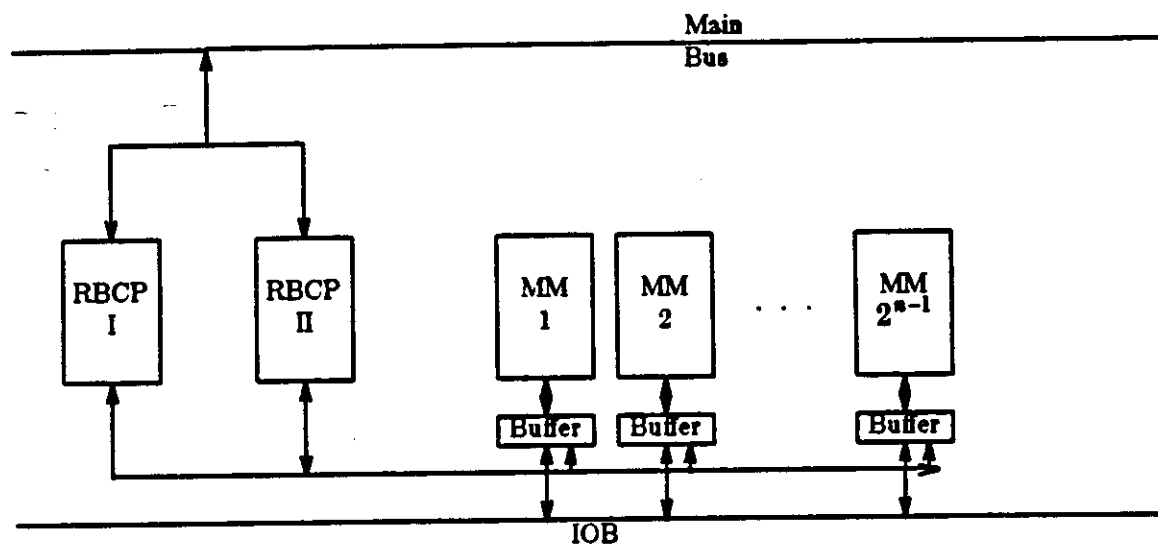


Figure 8.8: Modular Organization of the RAM Buffer

8.3.1.4 Network Interface Modules

The reader should at this point have a fairly good idea of the operations that are performed by the NIM's. To recap, there is one NIM per external bus that takes place in information transfers (there may be others available in "standby" mode should the active module fail). In this section we will provide more details on the internal operations performed by a NIM.

¹Notice that this requires that there be an even number of modules configured in the system (assuming no modules have failed).

Internally, each NIM is logically structured as shown in Fig. 8.6. Each NIM is a self-checking processing module, similar in design to those used as PM's but possessing less processing capability and less internal memory.¹ The NIM architecture, however, possesses additional hardware in the form of addressing logic and bus buffers for the IOB, and I/O logic that interfaces the NIM to the external LAN.

The NIM performs all packet management for information being transferred into or out of the system. As packets are received, they are decoded if they have been encrypted in some way to provide additional security, checked for correctness, and the data in the message retained in memory buffers. Outbound messages from various system modules are packetized by the NIM and sent to their destination over the LAN. In addition to encryption/decryption, the NIM may also perform a limited amount of information manipulation, based upon the criticality of the operation.²

As explained in a previous section, the NIM is configurable from other system modules. In particular, the NIM must be made aware of the destination of a packet within the system for inbound packets, and in a similar key, it must be made aware of the destination of outbound packets for the LAN. Normally, this is done by establishing a virtual circuit between FTSS and an external host. The NIM retains these virtual circuit addresses in internal tables

¹For economy reasons, it is desirable to prevent duplication of effort. Therefore, it is prudent to assume that the NIM's and the PM's share a common ancestry.

²One possible application of the NIM is in performing data conversions on information being transferred into or out of the system. For example, translations from ASCII to EBCDIC characters, and vice versa, for text files being transferred between inhomogeneous hosts.

for its use, should problems arise elsewhere in the system. Apart from this, all internal messages between the system modules and the NIM's must include a destination address if the message is destined for the LAN. As explained previously, multiple block transfers can be optimized by having the sending module inform the NIM of the starting block and range of blocks to be transferred from the RB. The NIM then uses the high speed IOB to perform this transfer without involving the sending module.

From an availability point of view, the NIM's are replicated, but not by the hot spare approach as used for several other modules. As long as there is at least one NIM that can communicate with the LAN, the system can be considered available. FTSS accomplishes this by replicating the external LAN and providing one NIM for each LAN bus. If a NIM fails, since it is self-checking, it can detect these failures and disconnect its appendages to the various buses. Since each sending or receiving module maintains information on virtual circuits, the transfer may either be continued or restarted on another NIM.

8.3.1.5 Internetwork Interface Modules

Although we have not discussed the IM's very much, they are nevertheless critical to the operation of FTSS. These interfaces form a "gateway" between the TL and the Internetwork. They are primarily device controllers and serial I/O controllers that interface to head-per-track disks and the Site Processors of the Internetwork, respectively (see the discussion on the Internetwork architecture below). The IM's communicate primarily with the RAM Buffer subsystem, although they communicate with all other modules in the system from time to time. For example, processes running on the PM's might

send messages into the Internetwork requesting that some operation be performed. All message traffic is handled by the serial I/O controllers. On the other hand, the device controllers are responsible for rapidly moving object blocks between the TL and the Internetwork.

The IM's are actually simple processing systems (based on microprocessor technology) which provide the basic functionality of the interface. Logically, the device controller module consists of a processor, a small amount of processing and buffer memory, interface logic to the IOB and an interface to the main bus (via a bus adapter). Messages destined to the device controller module typically specify data to be written, and the object to which it belongs. A directory mapping logical object block numbers to physical disk addresses is maintained on the disk units, and is accessed by the module whenever data is to be read or written. As mentioned previously, the module can be set up to access multiple blocks which are stored directly in the RB. In a similar vein, the module may directly access information from the RB which is to be stored on the disks.

The serial I/O module is similarly structured to the device controller module. It consists of a simple processor, a small amount of processing and buffer memory, a serial I/O interface to a similar interface in the Internetwork, and an interface to the main bus. It does not interface with the IOB. All messages (that is, non-object transfers) between the TL and the Internetwork are routed through these modules. The serial I/O data transfer speed is orders of magnitude less than that of the main bus; therefore, messages are buffered until they can be handled. The amount of buffer space provided is a function of the expected message traffic between the two major subsystems.

The device controller and the serial I/O controller modules are duplexed. However, each performs independently of the other. If both modules (of each type) are operating correctly, multiple transfers can be taking place simultaneously.

8.3.1.6 Other Modules

FTSS may use other modules connected to its main bus as the need arises. For example, the PM's, when involved in transaction processing, will need to save their state in stable storage. Although FTSS is designed to be a reliable and highly available system (thereby obviating the need for stable storage), a small private disk unit accessible to all processors via the main bus would be appropriate. In another case, if the PM's are implemented with demand paging hardware, swapping devices will be needed; these may be attached directly to the PM, or directly to the main bus (an alternative is to do demand paging via the RB/Internetwork combination). In either case, modules that allow these devices to communicate across the bus would be required. There is also the possibility that, for monitoring purposes, various devices such as terminals, or printers will need to be attached to various portions of the system; these may also require specialized modules. In short, the extensibility of the architecture allows these devices to be added and easily integrated into the system without perturbing system operation unduly.

8.3.2 The Internetwork

As explained in section 8.3.1, the Internetwork or BL of FTSS consists of an interconnected set of SS's organized as a hierarchical symmetrical 1-FT tree. Each SS further consists of a SP and one or more storage devices. In this

section, we expand upon the architectural concepts of the SS's and the network which interconnects them. We also discuss various mechanisms used to manage objects in the system.

8.3.2.1 Storage Site Architecture

Each SS stores and manages a (possibly overlapping) subset of the total-ity of information in FTSS. In a later section we shall have more to say about storage and object management; for now, we shall concentrate on the architec-tural features necessary to implement SS's. While there are several ways in which the Internetwork can be physically designed, it is natural to use point-to-point connections between SS's. The reasons for this are the same given in chapter 6 on the extensibility of $T_{2,p,1}$ topologies, since FTSS is based on a $T_{2,4,2}$ topology. If point-to-point connections are not used, it is very difficult to extend the topology in an incremental way. The next decision which influences the design of the SS's is the nature of the communication on these point-to-point *links*. There are three alternatives: *circuit*, *message*, and *packet* switching. Circuit switching has the major disadvantage that it does not permit high-speed statistical multiplexing of data, a condition which is useful when traffic is bursty in nature. Furthermore, this lack of statistical multiplexing hinders the establishment of multiple virtual connections between SS's. Message switching has the disadvantage that it is not possible to effectively pipeline messages on the links; this leads to a delay (the *excess delay*), which is proportional to the message size, in excess of the situation where the sending and receiving sites are directly connected [Acam84]. In view of the potentially large sizes of objects (which would be transmitted as messages in FTSS), this delay would seriously impact system performance when the affected sites are busy. Packet

switching provides a solution to the disadvantages of circuit and message switching systems. By breaking messages into smaller packets, the excess delay is minimized (since it is proportional to packet size), and statistical multiplexing techniques can be employed to provide high link utilization. FTSS therefore uses packet switched communications on the intersite links.

In addition to providing storage for objects, each SS must also act as a packet switching communication node in the Internetwork. Packets destined for other sites must be stored and forwarded (based on a pre-defined routing algorithm); packets destined for the current site must be stored and interpreted (a packet can represent either data or a message). The dual role of the SS therefore implies fast access to locally-stored object data as well as fast packet switching in order to meet performance goals. The logical architecture of the SS is shown in Fig. 8.9.

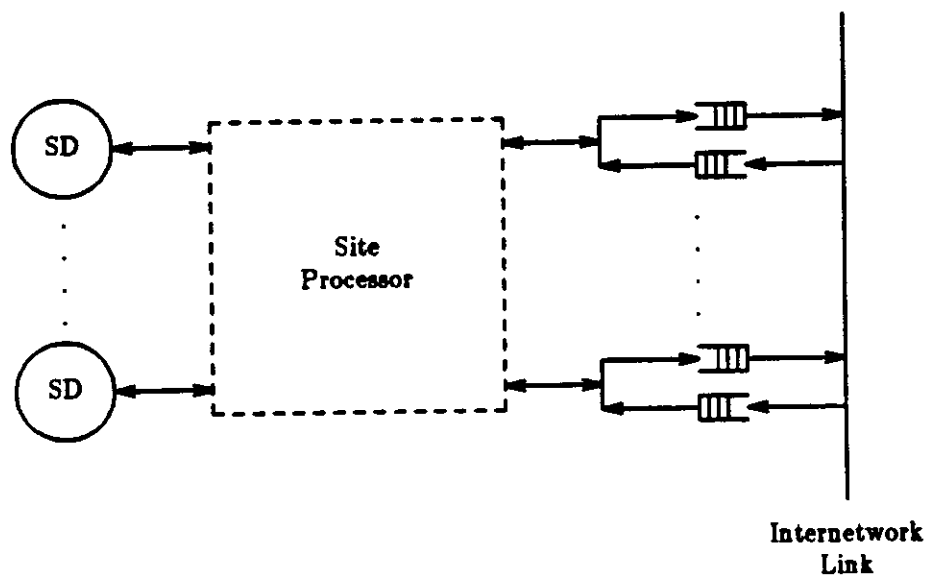


Figure 8.9: Logical Architecture of a Storage Site

On the front-end are a set of input and output queues that buffer packets from what is considered the logical network. On the back-end are a set of storage devices. The SP is responsible for all information flow between the various queues and the storage devices. In the next paragraphs we propose an architecture for an SS. For cost/performance reasons, it is assumed that each SS in the Internetwork has the same architecture.

The architectural model chosen for the SS is similar to that used by [Acam84], and for much the same reasons. It is based on a short bus which provides all the advantages of LAN-based architectures without the disadvantages. As illustrated in Fig. 8.10, the architecture consists of a set of buses to which various interface and special modules are attached.

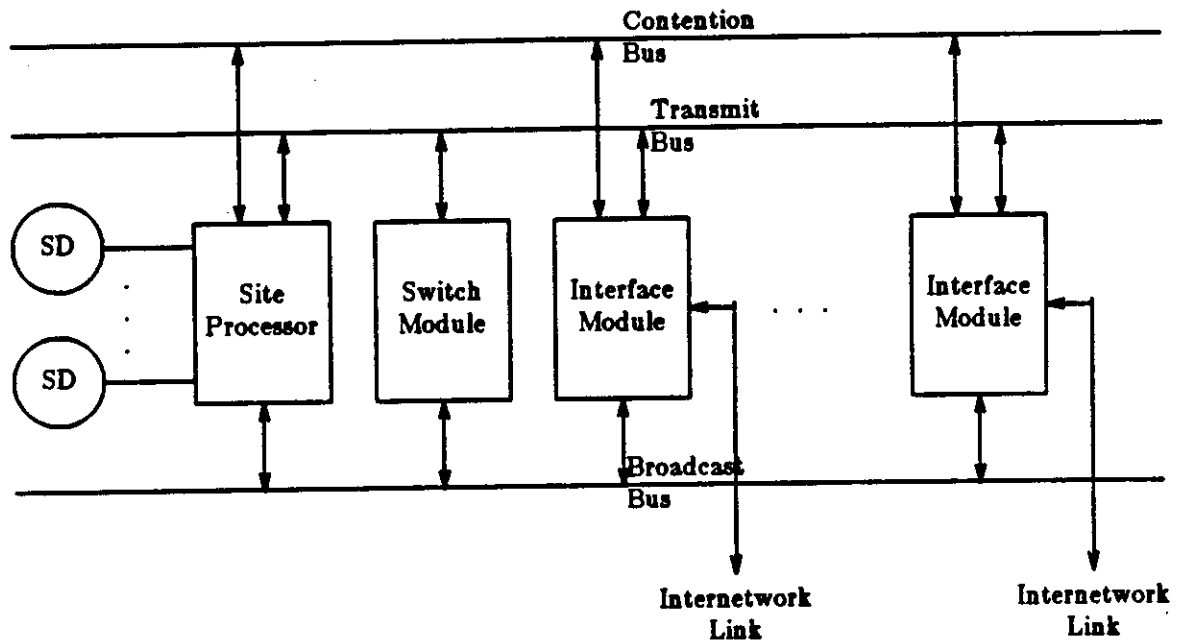


Figure 8.10: Storage Site Architecture

All data enters and exits the SS via the interface modules, each of which contains bus contention logic and packet buffers, along with any functionality

associated with the particular interface. The buses are clocked by a common clock on a special clock module. The switch module traps all packets sent on the transmit bus, translates addresses as necessary based on a distributed routing algorithm in the module, and rebroadcasts the packet on the broadcast bus -- the packet is either destined for the Site Processor or one of the interface modules. The SP, in addition to providing Internetwork management features such as object management, fault-detection and isolation, system configuration, and performance tracking, provides the necessary interface with the storage devices. Note that in this organization, the storage devices must communicate with the SP for all object transfers; in fact, one need not think of the SP as being a homogeneous processor, it can be distributed over several intelligent modules, each dedicated to a specific task (for example, there can be an intelligent interface for each storage device with the capability to provide a high-bandwidth path directly to the appropriate interface module). However, in order to keep the SS as simple as possible, we assume that the SP is a relatively high-performance homogeneous processor.

This architectural model provides the following benefits:

- Various communication technologies can be used and mixed at the same SS. For example, in those areas where extremely high bandwidth is needed, fiber optics can be used. In less bandwidth-sensitive areas, either coaxial cables or twisted wire pairs can be used. As one descends the FTSS hierarchy (with increasing level numbers), there is less of a need for high bandwidth -- those object frequently referenced will be at the top of the hierarchy (near the root). Thus, one would want to use fibers near the root, and twisted wire pairs at the highest level of the

hierarchy.

- Extensibility is supported since one can add interface modules to the architecture relatively easily (as simple as plugging in a new module to a backplane). In $T_{2,p,1}$ topologies, the number of interfaces to a SS is fixed (the node connectivity). Thus there is a reasonable upper bound on the number of slots that need to be provided for interface modules. Therefore, despite arbitrary extensibility, the architecture of the SS need not change.
- The short bus has the advantages of dynamic bandwidth allocation as a function of need, the utilization of flexible contention algorithms, and a simple media access scheme which permits perfect scheduling (no destructive collisions or idle periods with packets awaiting transmission), and small packet sizes (improving multiplexing).
- Better distributive properties due to geographical separation of nodes because of the use of various communication technologies. Although FTSS is logically centralized, SS's do not have to be physically centralized; therefore, the ability to distribute sites (say, in a building) is an important attribute of the system.

8.3.2.2 Internetwork Communications

All communications in the Internetwork is message-based. A message consists of an uninterpreted string of bytes which can represent data or control/status information. In this section, we describe the intersite communication protocol and give examples of its use.

Since FTSS operates as a packet switching system, it requires a protocol that will provide reliable packet transmission from one site to another. The protocol should prevent lost, out of sequence, or duplicate packets. Furthermore, the protocol should provide high performance by minimizing protocol overhead.¹ Saltzer [Salt80] has proposed simple end-to-end protocols, and Popek [Pope81] has proposed the use of simple problem-oriented protocols, both of which provide good performance in distributed systems. Understanding this, FTSS implements a 4-layer protocol structure which is essentially the first 4 layers of the International Standards Organizations' (ISO) open systems interconnection (OSI) protocol. Specifically, the physical, data-link control, network, and transport layers [Zimm80].

At the level of the transport layer, one site sends a message of arbitrary length to another. There is no name service in FTSS; each site is known by its unique *site number*, site numbers being assigned based on relative position in the hierarchy, which is in turn based on the node numbering scheme for $T_{2,4,1}$ trees. Each message contains a message header which identifies the destination site, the sending site (if a response is desired), and the message type. To minimize protocol overhead, FTSS depends upon the reliability of the communication system; in particular, it depends upon the reliability mechanisms implemented in the data-link control layer. Therefore, the philosophy adopted is that there is no explicit response for a message -- the information requested is considered the response. However, an additional response-request field is added to the message header for those site-messages that *do* require some sort of confirmation. This response-request field can be used to implement a

¹It was shown in the design of the Cambridge File Server [Dion80] that one of the largest contributors to performance was a protocol with minimal overhead.

transaction-message service when necessary. To do so, the message header contains a Transaction ID (TID) field which identifies the message uniquely (each TID is generated uniquely at a site -- this along with the source site number is sufficient to identify the transaction uniquely). The destination site maintains a list of outstanding TID's, and if a message appears with one of these TID's, it is discarded. This mechanism both guarantees that a message is acknowledged, and that its action is performed only once.

The network layer breaks messages into packets and routes each packet to its appropriate destination. Each packet contains a packet header which has the following information in it.

- destination site number
- source site number
- list of sites visited
- outbound link ID
- sequence number
- message ID

The list of sites visited is needed by the distributed routing algorithm (see Appendix B) so that packets have a bounded number of hops in the Internetwork. The outbound link ID is used to select the next outbound link in the SS and is changed in each SS. The message ID identifies the message to which the packet belongs; this is useful to distinguish packets that may have been delayed in the network, arriving during a different message. The sequence number is used to prevent against lost, delayed, or duplicate packets. Packets are sent out in sequence; a low-level acknowledgement is provided by the data-link control layer which is passed on to this layer -- if a NAK is received, or a timeout occurs, the packet is resent, otherwise the next packet is sent (if repeated NAK's are received, the layer sends a probe packet to the suspected

site; if no response is received, the site is assumed down and a note is made of it). Note that sending packets in sequence does not guarantee that they arrive in sequence since it is possible for a SS to route consecutive packets on different outbound links as a function of link availability. However, out-of-sequence packets should be a rare occurrence given the point-to-point nature of the links and the use of a low-level acknowledgement mechanism. Given this, FTSS takes the simple approach of discarding any packet that arrives out of sequence at a destination site. Flow control is implemented by using a *sliding window* technique. This window indicates how many packets may be transmitted before an acknowledgement is received; packets that are not acknowledged during the window are resent (based on a retransmit timer at every site).

The data-link control layer is responsible for sending packets reliably from site to site in the Internetwork. It frames each packet with enough information that will allow the packet to be delivered to the destination site in a reliable way, with high probability. Although there are several bit-oriented data-link control procedures available, FTSS attempts to minimize the communication overhead as much as possible so that a small number of frames need to be sent between sites relative to the message being sent. FTSS proscribes to a balanced, two-way simultaneous point-to-point bit-oriented control procedure. An example of such a procedure is given in [Carl80]. Simplification of the control procedures is possible because of the specialized nature of FTSS -- much of the mechanisms needed to support general network operations are not needed. Appendix C suggests a data-link control procedure for use by FTSS.

The physical layer is concerned primarily with the electrical characteristics of the transmission medium and the access protocols used to gain access to the medium. Since FTSS is designed to support many types of transmission media, depending upon the desired communication bandwidth, several types of access protocols can be in effect. Since it is not expedient to cover all possible access protocols, we will elide any further discussion of the physical layer. However, because of the portending importance of optical fiber transmission media, especially in the areas of reliability and security, we strongly support the use of fibers as a transmission medium throughout FTSS.

8.4 FTSS Software

In this section, we show how the SMS architecture discussed in chapter 3 can be mapped onto the FTSS hardware.

The SMS executes on the TL hardware discussed in section 8.4.1.2, that is, on the set of PM's arranged as a two-level hierarchy of HLM's and LLM's. The HLM's execute the kernel of SMS, and the LLM's execute user-level processes. In view of the distributed nature of the processor architecture, the software architecture of SMS must be adjusted so that it can execute on these distributed PM's. Furthermore, the nature of the architecture of each PM, as well as the *modus operandi* of the HLM's (for reliability reasons) imposes further structure on the SMS. In the following sections we discuss these constraints and propose solutions to them.

8.4.1 Distributed SMS Architecture

In the normal scheme of operations, kernel functions are executed in the HLM's and user-level processes in the LLM's. Therefore, an execution environment must be provided in the LLM's by the HLM's in which processes can execute. When a user-level process is to be executed, the process image is loaded into one or more LLM's (depending on the criticality of the operation), its addressability established, and control passed to the LLM where execution begins. As the process executes, kernel calls (including I/O operations) are packaged as messages and sent to the kernel in the HLM's where they are executed and the results returned. This operating scenario requires a number of things. First, it must be possible for the kernel to establish an execution environment in an LLM. Since this is, in part, a kernel function, it requires that a (small) part of the kernel be resident in each LLM. This *Local Kernel Component* (LKC) serves as a "bootstrap" whereby server mechanisms can be established for communication with the Main Kernel Component (MKC). In order to avoid security breaches that might result because of corrupted LKC's, it is recommended that the LKC be thoroughly tested and implemented in microcode; this both prevents tampering as well as improving performance. Second, there must be a way for the two kernel components to communicate. This has already been implied from the first consideration. Note that the message facility discussed in chapter 3 is primarily an interprocess communication mechanism; in this case, we need a mechanism by which two kernel components can communicate without involving processes, therefore a specialized communication facility is built into both kernel components. Third, the kernel must have a mechanism to choose an appropriate processor when a process

must be created -- this in fact is conceptually simple if processes are treated as objects and placed under control of the scheduler. The logical architecture of the distributed SMS is illustrated in Fig. 8.11.

8.4.2 Effect of FT on SMS Architecture

Apart from the need to be able to start more than one copy of a process on a LLM (a decision which is left to the system implementor) and monitor their progress, the hardware architecture of the PM's for fault-tolerance purposes impacts the SMS architecture in a minor way. In fact, there is more of an impact on the style of programming than on the architecture itself. For example, the kernel code must be written in such a way that rollback points are provided for the eventuality of a PM failure. Since the HLM's operate in duplex with a hot spare whose outputs are not used unless a failure occurs in the primary, no special software modifications are necessary. In short, beyond the need for the kernel interfaces, not much more needs to be done with the basic software architecture presented in chapter 3.

8.5 Summary

In this chapter we have reviewed various hardware and software issues that impact the design of FTSS. The need to provide a very reliable root for the hierarchical tree system resulted in a distributed root architecture which provides high availability through modularization. It is permissible to tolerate sporadic failures, as long as the mean time to repair these failures is much less than the mean time between failures. Rather than postulating a very reliable design, FTSS takes advantage of the modularization concept and the fact that the system is repairable to provide high availability. This is to be contrasted

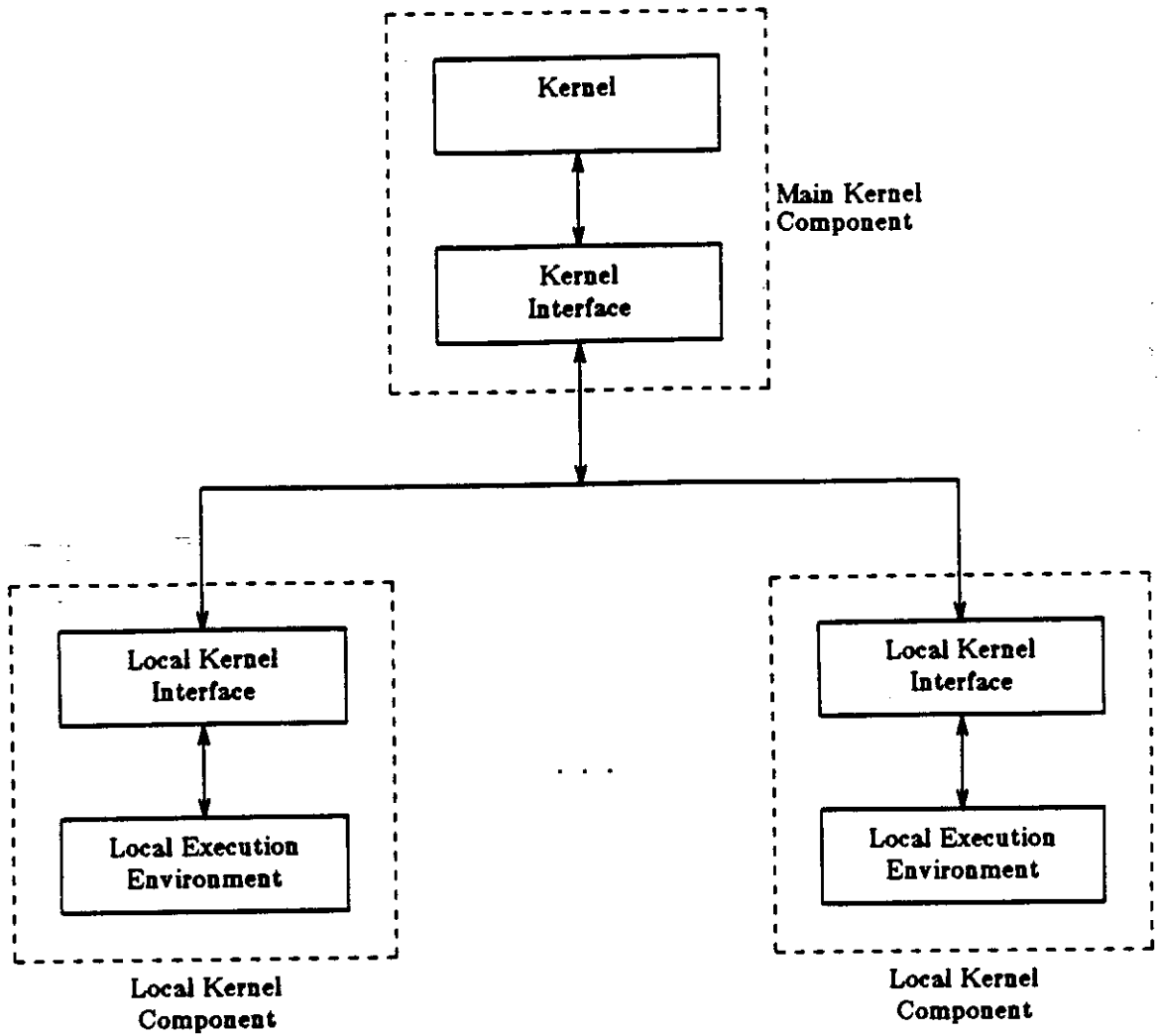


Figure 8.11: Logical View of the Distributed SMS

with a closed (that is, non-repairable) design where online repair is not possible. To make this work, great effort was made to ensure that the architecture permits online repair without service disruption. We feel that if carefully constructed, this design achieves that goal. Beyond the need to distribute a small component of the kernel which provides a message service for kernel calls from LLM's to HLM's, the hardware architecture imposes minimal constraints on the software architecture. This, in part, tends to support our choice of an object-based software architecture, which naturally lends itself to modularization -- each hardware component can indeed be considered an object in its own right, and treated as such by the operating system.

CHAPTER 9

PERFORMANCE ISSUES

In this chapter we discuss various aspects of the performance of FTSS based on our implementation of the previous chapter. We show how the performance is affected by various system variables. We feel that the performance can be adjusted to any reasonable upper bound based on available technology. The results give a very good indication of the potential of storage systems such as FTSS.

9.1 Introduction

The ultimate performance criterion of any centralized storage system can be expressed in terms of the mean response time to a request from a user's perspective. For example, if the user is requesting data out of the system, it is the mean time from the initiation of the request until the data arrives at the host. Ideally, we would like this time to be commensurate with, or less than, that which would be obtained if the user was accessing data from storage local to the host.

In the following discussion, we limit the scope to only a consideration of system bandwidth which is consumed by data (or file, if one prefers) traffic in direct response to user requests. That is, we do not consider internal "house-keeping" operations performed by the system.

9.2 Bandwidth Considerations

In order to assess the performance of FTSS, one can think in terms of available bandwidth. For example, since it is expected that the bandwidth of the object store (the IL and the BL) will be the critical factor in the system's performance (they are the slowest components), it is important that this bandwidth not be exceeded. If we assume that there are two types of requests prevalent in the system, one which we call *server* requests primarily due to users making requests for random blocks of an object, and *file transfer* requests primarily due to users uploading and downloading information, then it is possible to express the bandwidth requirements of the system in terms of the reads and writes implied by each type of request. If we assume that the bandwidth of the object store is B blocks/sec, then the following should hold in the majority of cases for good performance:

$$B \geq Sw + FTw + FT r + (1 - HRRB)Sr \quad 9.1$$

where S represents a server operation, FT represents a file transfer operation, r and w represent read and write operations, respectively, and $HRRB$ represents the hit ratio of the RAM buffer.

In order to assess the impact of these components on the system bandwidth, we have constructed a simulation model which permits a wide range of alternatives to be explored. In the next and following sections, we describe the simulation model, the exercising of the model, and the results which were obtained.

9.3 The Simulation Model

The model is constructed based on the block diagram of Fig. 9.1. Each block in the diagram is modeled separately; the model therefore represents a "gross" view of the system's operation. For example, no attempt is made to account for the use of multiple processors in the processor block, or of multiple storage devices in the IL and BL blocks. It is instead assumed that the processor or storage devices spend a certain amount of time processing a request, independent of how this is actually done. Because of limitations in the modeling tool used, no attempt is made to distinguish between individual hosts in the system - they are treated collectively in the "host" block.

The active entities in the model are *messages* which either represent control information or data. There are two types of control messages generated by the hosts: namely, server (S-messages) and file control (FT-messages); these are mapped into other control messages by the processor block. Data messages typically represent information stored in objects and are assumed to be of a fixed size. S- and FT-messages may vary in size (up to the size of a block); other control messages are assumed to be of fixed size.

9.3.1 Model Description

In this section we describe some of the key features of the model in order to give the reader a flavor of how it works. To begin with, each block in the model has a unique address. Messages carry with them source and destination addresses, the message type (control or data), the request type (read or write), the message mode (S or FT), the size (in bytes), and other control attributes specific for the requirements of the simulator used (for example, we simu-

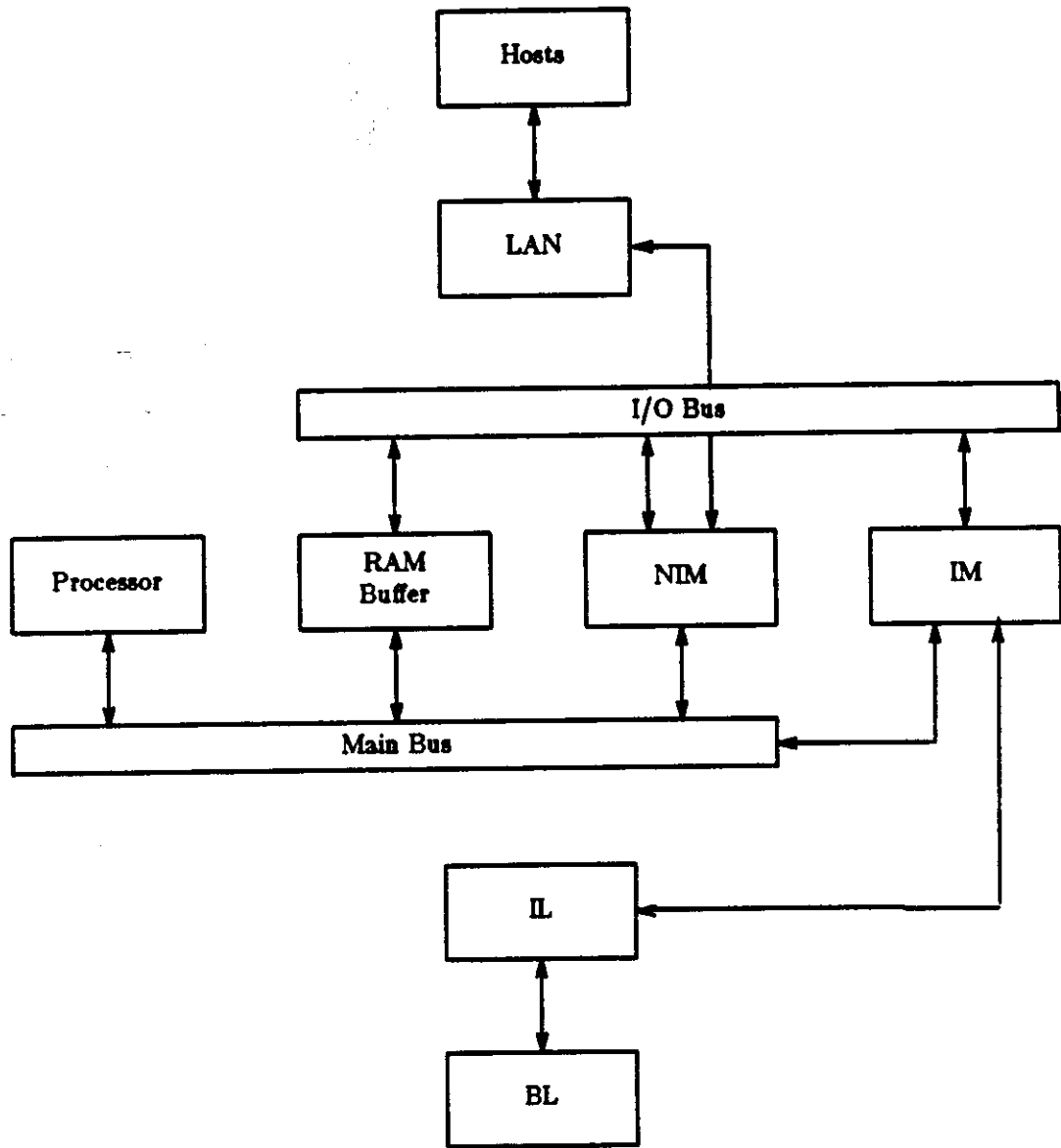


Figure 9.1: Block Diagram for Simulation Model

late the number of blocks affected by a read or write request as a random variable carried in the message itself - normally the amount of data to read in an FT-message is determined by the size of the object in the object store). The following algorithms describe the sequence of steps that occur in response to an S- or FT-message.

Algorithm 9.1: S-message

1. Host generates an S-message
2. S-message is sent over the LAN to the Processor.
3. S-message is intercepted by the NIM and transmitted to the Processor via the Main Bus.
4. Processor receives the message and checks to see if it is data or a control message. In this case we assume it is an S-message.
5. Processor determines if the object is in core. If it is, go to step 7. Otherwise, the processor sends a message to the RAM buffer to retrieve the GODI block. The processor then suspends processing of the message until the block is returned. When the block is returned, go to step 7.
6. The processor determines if the requested block of the object is in core. If it is, go to step 8.
7. Send a control message to the RAM buffer requesting the block of the object. The processor suspends processing of the message until the block is returned. When the block is returned, continue.
8. If the request was a read operation, the block returned from step 7 (which is now data) is sent back to the host via the Main Bus, NIM, and LAN. If it is a write operation, the data sent by the host in the write request is written to the object store.

The above algorithm treats S-messages in a manner which suggests interactive use. This would correspond to the case where a remote host is using a subsystem in FTSS to perform storage management functions; in this case, each S-message then operates on a block of an object, either modifying the block and writing it back, or requesting some data from the block. The model permits a variable number of blocks to be affected by each S-message.

Algorithm 9.2: FT-message

1. Host transmits an FT-message to the Processor via the LAN.
2. Message intercepted by the NIM and passed on to the Processor via the Main Bus.
3. Processor determines the type of the control message. In this case we assume it is an FT-message.
4. If the message is a read operation, go to step 5. Otherwise, the processor sends a control message to the host instructing it to begin the data transfer. A similar control message is sent to the IL instructing it to receive the data sent by the host. Stop.
5. The processor sends a control message to the IL to transfer the blocks of the requested object directly to the host.

Notice that FT-messages once interpreted by the processor, do not again involve the processor directly, except for possible error recovery (the model does not deal with this event however). The processor sets up both the host and the IL, and the transfer takes place independently.

The RAM buffer operates as a write-through cache for data blocks flowing between the processor and the IL. When the processor writes to the IL, it in fact writes to the RAM buffer, which in turn passes the data to the IL. Similarly, when data is read from the IL by the processor, the returned data is written into the RAM buffer and then passed to the processor. Thus, there is some probability that if locality effects hold true, the next block to be referenced by the processor will be in the RAM buffer. All data blocks flowing between the RAM buffer and the IL use the I/O Bus; control messages always use the Main Bus. Thus, the Main Bus is used for both data and control messages, while the I/O Bus is strictly for data messages. Notice that data messages in response to FT-messages take the general path: Host-LAN-I/O Bus-IM-IL. This simplistic model provides us with enough functionality to at least get an appreciation for the location of bottlenecks in the system's performance

while maintaining some concept of realism.

Table 9.1 shows the variables that we have used in the model.

Variable	Description
t_{SM}	Interarrival time of server requests
t_{FT}	Interarrival time of file transfer requests
t_{NIM}	Processing time per byte at a NIM
t_{BUF}	Block access time at the RAM buffer
t_{KERNEL}	Kernel processing time per request
t_{IL}	Block access time at the IL
t_{BL}	Block access time at the BL
t_{IM}	Processing time per byte at the IM
t_{HR}	Block access time at a host following a write request
t_{LAN}	Time to transfer one byte on the LAN
t_{MB}	Time to transfer one byte on the Main Bus
t_{IOB}	Time to transfer one byte on the I/O Bus
PRW	Ratio of reads to writes for S- and FT-messages
SMBS	Message size for S-messages in bytes
NBL	Number of blocks affected by each FT-message
HRIO	Hit ratio for incore objects
HRIB	Hit ratio for incore blocks
HRRB	Hit ratio for RAM buffer blocks
HRIL	Hit ratio for IL objects

Table 9.1: Simulation Model Variables

9.4 Using the Model

To determine where the bottlenecks are in the system, we have assigned values to the simulation variables which form a *baseline* from which we can say more about the system's performance. The values were chosen in such a way that the system provided what we consider to be an acceptable response time without maximizing utilization of any resource (block in the model). No attempt was made to balance the utilization of resources. Once this baseline was established, we selectively adjusted model variables, one at a time, to determine its effect on the system response. From this we are able to postulate where the bottlenecks are in the system, and offer suggestions on how they can

be alleviated. The baseline values are shown in Table 9.2.

Variable	Value	Distribution
t_{SM}	10-100	Uniform
t_{FT}	500-600	Uniform
t_{NIM}	.001-.016	Uniform
t_{BUF}	2-3	Uniform
t_{KERNEL}	.1-1	Uniform
t_{IL}	20-30	Uniform
t_{BL}	200-300	Uniform
t_{IM}	.001-.016	Uniform
t_{HR}	40-80	Uniform
t_{LAN}	.008	Constant
t_{MB}	.016	Constant
t_{JOB}	.016	Constant
PRW	4/1	Discrete
SMBS	10-1024	Uniform
NBL	1-10	Uniform
HRIO	.5	Discrete
HRIB	.7	Discrete
HRRB	.9	Discrete
HRL	.95	Discrete

Table 9.2: Baseline Simulation Values

The result of running the model for 100000 units of simulated time is shown in Table 9.3. The table shows mean queueing time at selected resources, the percent of the resources utilized, and the mean response time to user requests. For the queue and response time statistics, the 95% confidence interval is also indicated as $\pm n^1$. The response time is broken down into the mean response time for a single block for all requests, for S-messages only, and for FT-messages only - for both reads and writes. Additionally, we have shown the mean response time for S-messages written to the RAM buffer; that is, the mean time to write a block to the RAM buffer in response to an S-message.

¹The 95% confidence interval is shown in a similar way in the tables that follow.

Queues				
LAN	Main Bus	IL	I/O Bus	RAM Buffer
3.61	13.83	23.1	7.83	1.214
± 0.193	± 0.327	± 1.526	± 0.641	± 0.477

Percent of Resource Utilization									
LAN	NIM	Kernel	RAM Buffer	IL	BL	Main Bus	I/O Bus	IM	Hosts
21.8	21.8	1.9	6.3	39.3	14.9	71.6	25.9	13.1	16.9

Response Time						
Read			Write			
Total	S-Msgs	FT-msgs	Total	S-Msgs	FT-msgs	S-Msgs to RAM
178.08	100.97	43.64	235.43	159.32	70.07	103.41
± 13.85	± 5.05	± 0.557	± 11.52	± 10.6	± 1.35	± 10.43

Table 9.3: Baseline Simulation Results

The results indicate that there is no significant queuing at any of the resources. Blocks were queued at the IL for a mean time of 23.1 units of simulated time. The Main Bus was the most heavily utilized resource (at 71.6%) while the kernel was utilized the least (only 1.9%). Since the frequency of requests is skewed towards S-messages, it is not surprising that the main bus is relatively heavily utilized since it must handle both control and data messages. Notice that the I/O bus is relatively lightly utilized since it handles only data messages. In order to give some meaning to the response time statistics, we assume an arbitrary (but convenient) time base of 100 microseconds. With this time base, the speed of the LAN is on the order of 10Mbps (million bits per second), the I/O and main buses are each 5 Mbps, and S-messages are sourced at the rate of 100 to 1000 messages per second, uniformly distributed. Other timing parameters can be interpreted similarly. Given this time base, the mean response time for all read messages is 17.8 milliseconds per block; for S-messages it is 10.097 milliseconds per block, and for FT-messages it is 4.364

milliseconds per block. These numbers require some additional explanation. The read response time is the mean time from the initiation of a message until the requested block(s) is(are) returned to the user. Since FT requests are passed off to the IL subsystem, the response time is largely the mean time it takes a block to move from the IL to the requesting user - this accounts for the relatively fast response of FT-messages. S-message responses generally take longer because of the processing overhead in the kernel (there is a probability that two block references may be required, and that both times the requested block will have to be fetched out of the IL). The write response values are similarly interpreted. They are higher due to queueing effects at the IL and the Main Bus. The read response time is certainly the more important to the user since it usually is inconsequential how long it takes to write a block, as long as it is guaranteed to be written correctly, and a read is not performed immediately after a write.

For the time base given above, the level of activity in the system is characteristic of about 150 hosts accessing the system simultaneously. We base this on measurements taken on our local distributed processing system consisting of 4 VAX-750's connected by a 10Mbps Ethernet and running the distributed operating system LOCUS. The background load on the Ethernet is about 30000 blocks per hour relative to file transfer operations. We have prorated this to 50000 blocks per hour to account for FT operations which are not an explicit part of our local operating environment, although there are some operations which approximate this.¹ The next step is to vary individual parameters and observe their effect on response time.

¹Such as executing a load module which is not in a local file system. The entire load module is transferred to the local site before execution commences.

9.4.1 Varying LAN Bandwidth

We begin by determining the effect of LAN bandwidth on response time. Table 9.4 shows the read and write response times to both types of messages as the LAN bandwidth is modified.

t_{LAN}	Percent utilization of LAN	Response Time	
		Read	Write
.008	21.8	178.08 ±13.85	235.43 ±11.52
.009	24.8	206.6 ±22.6	231.65 ±10.15
.01	27.6	161.8 ±7.88	228.36 ±9.94
.01143	30.1	160 ±8.93	229.6 ±10.06
.0133	35.3	227.3 ±20.09	232.35 ±12.6
.016	44.2	168.5 ±9.18	240.27 ±10.29
.02	56.3	223.2 ±15.2	247.6 ±11.2
.0267	71.8	227.86 ±11.5	265.3 ±11.5
.04	99.9	5871.9 ±177.5	5118.7 ±344.7
.08	100	32262.3 ±1307.3	24626.4 ±1796.4

Table 9.4: Effect of LAN Bandwidth on Response Time

Reasonable response times are achievable until the LAN utilization gets to about 80%, after which it increases rather rapidly. Although the LAN is external to FTSS, it is necessary to consider its contribution to performance since bottlenecks here will affect the user's perception of FTSS response time. Although this model clearly shows that a LAN bandwidth as low as 3Mbps ($t_{LAN} = .0267$) would suffice, it also indicates that the model can support a much higher offered traffic load. A 10Mbps CMA-CD multi-channel bus is

feasible using current technology; therefore, we will assume that the external LAN bandwidth can be adjusted to meet the response time requirement of the user. Notice that it is possible to assume other time bases that will result in either a lower or a higher upper bound. However, given current technology, we do not feel that speeds much higher than 100Mbps are cost-effective for generalized Local Area Networks.

9.4.2 Varying the IL Bandwidth

Next, we investigate the effect of the bandwidth of the IL on overall system response. We consider this to be a critical part of the system's performance since it is here that information is semi-permanently stored. Table 9.5 shows the effect of IL bandwidth on response time, starting with the baseline system. The results indicate that given this level of offered traffic, the IL must be able to process a block read or write in 5 to 6 milliseconds in order to provide a mean response time of about 40 milliseconds per block. This level of performance is certainly within the capabilities of contemporary storage technology. For example, a fixed head disk with a data transfer rate of 2 million bytes per second can read or write a 1Kbyte block of data in 510 microseconds with an average latency of 5 milliseconds for randomly addressable blocks. Notice that in the case of FT-messages, which request an entire object, the mean time to process a block is much less if the blocks are stored in such a way that rotational latency can be minimized. One technique for improving the response time for S-messages (which are assumed to access blocks randomly) is to pre-fetch blocks to the RAM buffer in order to improve the RAM hit ratio.

t_{IL}	Percent utilization of IL	Response Time	
		Read	Write
20-30	39.3	178.08 ± 13.85	235.43 ± 11.52
30-35	52	183.35 ± 11.06	239.1 ± 9.85
35-40	59.4	206.49 ± 13.3	248.17 ± 10.46
40-45	66.1	185.85 ± 7.85	262.77 ± 10.29
45-50	79.6	245.71 ± 15.46	323.91 ± 12.57
50-55	85	281.55 ± 15.44	350.99 ± 14.24
55-60	90.2	393.43 ± 26.16	410.84 ± 18.13
60-65	98.2	1222.86 ± 100.68	1215.1 ± 48.69
65-70	99.1	2554.62 ± 195.97	2546.37 ± 91.76
70-75	99.9	3936.46 ± 349.18	4477.6 ± 168.22

Table 9.5: Effect of IL Bandwidth on Response Time

9.4.3 Varying the Processor Bandwidth

The processor bandwidth has a pronounced effect on the response to S-messages since some processing by the system is required on each block affected by the message. In this simulation model we have restricted the number of blocks affected by each S-message to 1, typical of users that treat FTSS as a guest layer on their local operating systems. The processor bandwidth also affects FT-messages, but to a much lower degree since it only needs to initiate transfers between the hosts and the IL. In our model, we have chosen a certain processing time in the baseline system which provides minimal queueing in the processor block. We now investigate the effect of this processor bandwidth on

response time. The results are shown in Table 9.6.

t_{KERNEL}	Percent utilization of KERNEL	Response Time	
		Read	Write
0.1-1	1.9	178.08 ±13.85	235.43 ±11.52
1-5	10.5	174.52 ±11.88	237.23 ±10.04
5-10	25.6	171.77 ±9.69	231.47 ±9.77
10-15	43.2	198.41 ±12.5	238.74 ±9.93
15-20	59.3	171.91 ±6.33	262.46 ±10.25
20-25	76.5	222.5 ±12.67	269.38 ±9.56
25-30	96.9	560.68 ±26.15	607.35 ±35.43
30-35	99.8	3580.91 ±145.11	3238.2 ±221.96
35-40	100	8318.24 ±369.1	8308.61 ±704.74

Table 9.6: Effect of Processor Bandwidth on Response Time

From the results it might be concluded that a processing time of about 2.5 milliseconds per block (given our time base of 100 microseconds) is needed to produce a block response time on the order of 50 milliseconds. To put this in perspective, if the processor executes one instruction per byte in each block (an optimistic assumption), it would require a 0.5 Million Instruction Per Second (MIPS) processor to be able to process a block in about 2.5 milliseconds. However, it is not the raw processing speed of the processor which is important - rather, it is the ability to be able to process blocks from the processor queue at the rate of about one block every 2.5 milliseconds. This level of performance is possible through multiprocessing using a multi-server queueing discipline. This approach has the advantage that much slower processors can be used, as long

as there is a sufficient number of them to minimize the mean queueing delay. This is the approach we proposed in chapter 8. Several microprocessors in the 1 MIPS regime could be used to implement the high- and low-level processor modules suggested in chapter 8.

9.4.4 Varying the RAM Buffer Bandwidth

Under the assumption of a 90% hit ratio in our baseline system, the RAM buffer can have a significant impact on the read response time of S-messages. Here we investigate the ramifications of buffer bandwidth on response times. Although in the model we have placed no constraints on the size of the buffer, we assume that if a block is to be displaced due to size limitations, it is simply written over since the buffer is write-through. The results of varying the buffer bandwidth is shown in Table 9.7. The results indicate that one can tolerate a bandwidth of about 3 milliseconds per block while providing less than a 50 millisecond block response time. This level of performance is roughly equivalent to accessing one 32-bit word every 10 microseconds - easily achievable using current semiconductor technology. It does not appear that the RAM buffer will present any significant performance obstacles.

9.4.5 Other Variables

We have varied many of the other variables listed in Table 9.1; we report here on their impact on response time. The performance of the NIM is upper-bounded by about 500 microseconds per byte to keep its utilization about 90%. We would expect its performance to be somewhat close to that of the LAN for good performance, and this appears to be the case. The performance of the IM is less critical, since it does not handle as much traffic as the

t_{BUF}	Percent utilization of BUF	Response Time	
		Read	Write
2-3	6.3	178.08 ±13.85	235.43 ±11.52
4-6	13.2	163.83 ±9.32	231.66 ±8.87
6-9	18.8	174.57 ±11.26	225.81 ±9.39
8-12	25	175.42 ±11.2	258.87 ±10.28
10-15	32.1	177.34 ±8.79	252.8 ±9.52
12-18	37.7	174.46 ±6.92	243.6 ±9.63
16-24	51.7	202.7 ±12.61	240.05 ±8.78
22-33	70	211.72 ±12.81	276.2 ±9.3
30-45	97.8	445.5 ±20.36	739.23 ±42.4
40-60	100	4968.5 ±364.3	8395.8 ±878.3

Table 9.7: Effect of RAM Buffer Bandwidth on Response Time

NIM and less processing per byte transferred is required. Skewing the offered traffic so that FT-messages dominate S-messages tends to favor the performance of the system since less processing overhead is required. We found that acceptable response times (less than 50 milliseconds per block) are achievable if the rate of generation of FT-messages approaches about 1 request every 15 milliseconds, while holding the rate of S-messages at the baseline value. The observed trend in extant local network architectures with file servers is that about 70% of network traffic is related to file server activity [Shoc80]. Therefore, a skew towards FT-messages is perhaps more inline with the real state of affairs. However, the view that S-messages will dominate leads to an upper bound in performance.

We have also varied the four hit ratios indicated in Table 9.1 between 0.1 and 1. The most significant impact on response time was HRIL, the hit ratio between the IL and the BL. Here, a hit ratio greater than 65% is required in order to provide response times less than 50 milliseconds per block. None of the other hit ratios proved to be very significant to performance, due no doubt to the generous performance margins assumed in the baseline system (as evidenced by the low utilization of some of the resources). A better measure of the impact of hit ratios would be observed if the utilization of the resources were forced to be in the 70-85 percentile range. This was actually observed to be the case, although we do not report on those results here.

9.5 Summary

In this chapter we have investigated some of the performance issues of FTSS using simulation modeling. Based on what we consider representative data we have shown that FTSS can meet the performance requirements of a large class of users in a local network environment representing a mix of file transfer and server requests. Although our analysis was skewed towards server requests, we think this represents a worst-case scenario, and therefore represents an upper bound on performance. Clearly, the results are not conclusive since many simplifying assumptions were made in the model. However, the functionality of the model is close to that of a real implementation. In particular, we have not accounted for the internal housekeeping functions of FTSS which will certainly consume resource bandwidth, thereby affecting performance. Indications from other file server systems are that the level of housekeeping is only a small percentage of the system's operation. We have based our response time requirement on about 50 milliseconds per block which is a

figure we have measured in our own distributed environment. We feel comfortable with this level of performance given the class of processors we are using, and our operating environment (VAX 750's and 780's and a typical research environment supporting about 400 graduate students).

CHAPTER 10

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

In this dissertation, we have developed a new approach to the design of storage systems for use in distributed processing environments. In this chapter we review some of the key concepts used in the approach, their applicability, and their limitations. Finally, we discuss possible directions for further research which will remove some of these limitations.

Our motivation to create FTSS grew out of our desperation for a reliable storage facility that could grow or shrink as the need demanded it, and one that would not isolate our data because of continual operating system or other non-storage related failures. Early in its conceptualization, it was recognized that such a facility, if implemented with the "right" mechanisms could provide much more than a simple repository for information. Indeed, it could serve as the locus of information management in an environment where users either did not have such capabilities, or could not afford to implement them. It was therefore our goal to provide such a facility.

Our first key concept was to centralize our storage system resources. The primary motivation for this was to unbundle storage from the processors that use this storage. A side-effect of this decision is that storage system mechanisms become less complex because of the homogeneous nature of the system, and less costly because one can take advantage of economies of scale

by providing a common base on which all mechanisms are built.

Our second key concept was the use of a generalized data abstraction mechanism, first introduced in chapter 2, upon which arbitrary storage management or other user-derived policies can be built. While data abstraction has been used in several programming languages and in a few operating systems, its application in storage systems management is new. We feel that this is the right approach since it supports the general notion of extensibility so important to our goal.

The adoption of the object model is not a key concept, but is worth mentioning in this concluding chapter since it is the fundamental mechanism on which security is built. The introduction of the mechanism of prioritized capability sealing on top of the object model adds a new dimension to security granularity in storage systems, and permits arbitrary security policies to be implemented.

The development of an extensible architecture to physically support the storage management mechanisms and the information stored in the system was fundamental to achieving our goal. The architecture provides a natural form of redundancy which supports both high availability and high reliability. This is unique in storage system architectures. We also feel that this was the right approach.

The application of fault-tolerance and self-checking designs in the hardware is a key concept since it permitted us to relax some of the constraints on the internal consistency of replicated information, resulting in simpler and therefore more efficient and less costly mechanisms. In addition, the use of

redundancy in data paths and computing elements permits a higher level of performance through concurrency. Most importantly, the application of fault-tolerance is the only way in which the user can obtain a high degree of confidence in a centralized storage facility.

FTSS is not without limitations however. The most critical assessment is the fact that it is a paper design. The logical next step therefore is its implementation. It is not necessary to implement all the features at once; some of them may be retrofitted to existing local network systems to verify their applicability. For example, the SMS could be retrofitted to an existing operating system to handle the storage management functions. The next logical step would be to unbundle storage from host machines on the network, thereby creating the back-end structure of FTSS. The next step would be to extend the backend structure into the fault-tolerant architecture discussed in chapter 6. Finally, fault-tolerance can be added to the system to improve its reliability. Of course, the ideal way to implement FTSS is to do so in a manner which is independent of any particular host characteristic; that is, from scratch, taking advantage of economies of scale and existing technology.

A further area of research is performance tuning. It has been suggested that the overhead of the object paradigm can lead to poor performance. This area needs more study to determine the access patterns of typical users, the frequency with which they access data, and the types of requests that they make. Information security in the system is also a fruitful area for research, particularly in the areas of data encryption which protects data which is outside the protection domain of the system. For example, when information is transmitted to a host, or when, say, a disk is physically removed from the sys-

tem.

Despite these obvious shortcomings, FTSS portends to be an important contribution to the study of reliable online systems, especially in view of the trend towards the decentralization of computing resources.

APPENDIX A

Diameter of $T_{2,p,1}$ Trees

Theorem: The diameter D_T (defined as the largest distance between any node pair in terms of the number of links separating them) of any $T_{2,p,1}$ tree is

$$D_T = \begin{cases} i, & i \leq 5 \\ 2i - 5, & i > 5 \end{cases} \quad (\text{A.1})$$

Proof: The common assumptions we make in this proof is that a link is never traversed twice, all links are operational, and the tree is *complete*. From an investigation of the topology, it is easy to see that the largest distance between two nodes occur when those nodes are in the highest level of the tree, and they are in different halves of the tree. Consider, for example, the 4-level tree shown in Fig. 6.2. The diameter is found when any node in the set {8-11} must communicate with any node in the set {12-15} (where we are using the j in $x_{i,j}$). Notice also that the path defining the diameter does not include the root node (except possibly for nodes in the first three levels) -- a cross-link is usually traversed first. To characterize the diameter, we make use of a descriptive (and somewhat functional) tool which we call a *Recursive Structure* (RS). A RS is a $T_{2,3,1}$ tree given by the first three levels of Fig. 6.2 (shown in the dashed lines). If one examines the structure of Fig. 6.2, it can be seen that every node in a level *below* level $p-1$, $p > 2$ is the root node of an RS. Therefore, the bound on

the number of RS's is $2^{p-2} - 1$ -- the number of nodes in the tree below level $p-1$.

If one considers any two nodes in the tree, they either belong to the same RS, or they have a chain of ancestor RS's that end in a common root. Consider, for example, nodes 5 and 13. They belong to different RS's whose common ancestral root is node 1. Similarly, node 1 is the common ancestral root of nodes 11 and 15.

Within any RS, the maximum number of link traversals to get from one node to another is 3 (for example, from node 5 to node 7 via the path 7-3-6-5), and at most 2 link traversals to exit the structure either through the root or via a cross-link. The problem then of finding the diameter is equivalent to finding the maximum number of link traversals between any two RS's. If one considers the "outermost" RS's in the highest levels of the tree, $p-3$ traversals are required to reach the third level of the tree, 1 traversal to switch from one half to the other half of the tree, and another $p-3$ traversals to reach the leaf node. Thus, for a tree with p levels, $2(p-3)+1 = 2p-5$ traversals are required, for $p \geq 5$. If $p < 5$, it is fairly easy to see that the diameter is p through exhaustive analysis.

APPENDIX B

A Simple Distributed Routing Algorithm for $T_{2,p,1}$ Trees

Much of the mechanism needed for describing the routing algorithm has been developed in Appendix A. In particular, the notions of a Recursive Structure (RS), the number of RS's in a tree with p levels, the ancestral root of two RS's, and the number of link traversals in each RS. The algorithm is stated below.

Algorithm: Routing in a $T_{2,p,1}$ tree.

Given: A source node numbered S and a destination node numbered D .

Step 1: If $S = D$ stop. The trivial case.

Step 2: Find the ancestral root of S and D .

step 3: If S and D are in the same RS, then move directly from S to D and stop.

step 4: If S and D are in adjacent RS's (in the same level with an ancestral root one level above), switch to the adjacent structure using a cross link, move to the destination node, and stop.

step 5: Move UP the tree until a node is reached which is in the RS of the ancestral node and switch to the adjacent RS using a cross-link.

Step 6: Move DOWN the tree directly to the root node of the RS in which D resides, move directly to D , and stop.

The algorithm as given above assumes that all links are operational, and that each RS is complete. It does not depend upon the completeness of the tree. Under these assumptions, it will traverse the minimum number of links necessary to go from a source to a destination node. Notice that this minimum path is not necessarily unique. The algorithm can be made robust in the presence of link and/or node failures by making an arbitrary decision on which link to traverse and continuing. However, in this case, it is needed to keep a list of nodes previously visited in order to avoid cycles.

APPENDIX C

FTSS Data Link Control Procedure

Frame Format: F,A,T,[INFO],FCS,F

F → Flag

A → Site Address

T → Type Field

[INFO] → Optional Information Field

FCS → Frame Check Sequence

There are 2 frame types:

Information (I)

Control (C)

The individual fields in the frame are structured as follows:

Frame Field: 8 bits (01111110)

Address Field: 8 bits (allows 256 possible sites)

Type Field: 8 bits

Information	0	S	S	S	C/R	R	R	R
Control	1	L	L	L	C/R	R	R	R

S-bits - send sequence number

R-bits - receive sequence number

L-bits - control frame ID (8 possible)

C/R-bit - indicates whether the frame is a command or
a response

FCS Field: 16 bits - CRC based on CCITT V4.1 generator polynomial

$$x^{16} + x^{12} + x^5 + 1$$

There are presently 4 control frames in the protocol.

- Probe: Determines status of the receiver on link. The INFO field of the frame may contain additional information on the type of probe being performed.
- Retransmit: Retransmit all I frames starting from a designated point.
- Reset: Reset state variables. The frame may contain additional information for the receiver on which state variables are to be reset.
- Info: General information. Transmits general status or control information to the site at the other end.

APPENDIX D

OBJECT-LEVEL ALGORITHMS

Creation Algorithm

Function: `O_create(capability, index):capability | signal`

Parameters: capability must contain creation rights and must point to a TDO. An object of the type of the TDO will be created.

1. If capability does not have creation rights, return signal.
2. Manufacture a capability for the new object
 - 2.1 Allocate space for the capability (a structure)
 - 2.2 Get a UID
 - 2.3 Insert the UID in the capability
 - 2.4 Insert access rights in the capability with all bits on
3. Allocate space for the object. Return the addresses of the C-list and data part.
4. Update the ICDI (incore directory)
 - 4.1 Allocate a structure representing the new object. This structure maintains information on the object while it is in core.
 - 4.2 If no room in the ICDI (assumed static) passivate the least referenced object (one with the lowest reference count)
 - 4.3 Save the UID and set the object's incore reference count to 0.
 - 4.4 Save the addresses of the C-list and data part in the structure. This section of the structure maintains a list of addresses of blocks of each part currently in core.
 - 4.5 Save a byte count of the length of each part of the object.
 - 4.6 Save the type of the object.
 - 4.7 Replace the capabilities UID with the address of the structure.
5. Return the capability to the caller. Store the capability in the location specified by index.

Function: `O_create_*(index):capability | signal`

Parameters: index is the location in which the capability returned is to be stored.

/* This algorithm is generic for a set of kernel calls that create predefined kernel objects. The semantics are the same as the previous algorithm except there is an initialization step where the kernel inserts capabilities and data (if any) into the newly-created object.

***/**

- 1. Manufacture of capability for the new object**
- 2. Allocate space for the object**
- 3. Initialize the object**
- 4. Update the ICDI**
- 5. Return the capability**

Algorithm for reading a block from the object store

A block is read from the object store in response to one of the get*() kernel calls or directly by the kernel. There are two cases:

- 1) The object is active**
- 2) The object is passive**

Case 1:

/* Since the object is active, there is already an entry for it in the ICDI. We only need to read in the new block and update the ICDI, possibly displacing a block if the buffer pool is exhausted (see the write algorithm)

***/**

- 1. Allocate space to receive a block. If buffer pool full, select the least frequently used object and free some space.**
- 2. Send a message to the object store for the block.**
 - 2.1 Call an I/O routine directly that sends messages and pass it the appropriate parameters (strictly internal to the kernel) - this includes the address where the block is to be written.**
- 3. Wait for an interrupt from the I/O section when the block comes back.**
- 4. Update the ICDI with the address of the new block.**

Case 2:

/* We must first update the ICDI with the incore header of the object being read.

***/**

- 1. Create an entry in the ICDI for the object**
 - 1.1 Allocate a structure for the incore header**
 - 1.2 Update all local parameters (see the create algorithm)**
 - 1.3 Copy the structure into the directory**
- 2. Repeat steps 1-4 from case 1 above.**

Algorithm for writing objects

/* This algorithm describes how objects are written from SMS into the object store. There are 3 cases under which object blocks are written.

1. Modified blocks are written to free up incore buffer space,
2. Space is needed for a new object and the current object is the least recently referenced, and
3. An explicit request is made to write the object back to the object store.

The strategy we use is that the kernel performs all block writes, but the OMS assists by making decisions about the location of the object in the object store.

The first two cases are not visible outside the kernel. The last is available as a kernel call.

*/

Case 1: writing modified blocks

/* It is assumed that the kernel maintains a pool of blocks that is shared by all incore objects. Blocks are written back to the object store as the pool becomes full

*/

1. If the disks on which the object is stored has not been allocated, get it and update the incore object header in the ICDI. This is true only if this is the first time the object is being written.
 - 1.1 Pass the capability for the object to the OMS in user space. This is done by placing the capability in a designated index in a kernel-defined object for which the OMS has a capability, and signalling the OMS which we assume is running as a server process.
 - 1.2 OMS gets the free list of all sites and determines which sites are available.
 - 1.3 OMS updates the GODI via a kernel call (OMS is given a capability for the GODI at system initialization). This call triggers the kernel to write out the object.
2. Modified blocks are passed to the I/O module directly (in the kernel).
3. Each block is encapsulated as a message and transmitted to the object store.
4. The incore buffer pool is updated by putting written blocks onto a free list.

Cases 2 and 3:

/* There is no difference between case 2 and 3, only the mechanism that triggers them

*/

1. Repeat step 1 from case 1 above
2. Pass modified block pointed to by the ICDI to the I/O module.
3. Encapsulate as a message and send to object store.
4. Repeat steps 2 and 3 for each modified block in core for the object.
5. Update the incore buffer list (freed blocks to free list)

6. Remove the entry for this object from the ICDI.

APPENDIX E

EXAMPLES OF OPERATING SYSTEM PRIMITIVES

This appendix lists a set of services that illustrate how the basic system primitives can be used to create higher-level operating system functions. Table 1 lists these services and the subsystems and/or kernel modules that are involved in their execution.

Operation	Subsystems	Kernel Modules
Run()	PMS,OMS	PM,OM,CM
Send()	MS,PMS	PM,OM,CM
Receive()	MS,PMS	PM,OM,CM
submit()	TS,PMS	PM,OM,CM,DM
Make_Directory()	OMS,PMS	PM,OM,CM,IOM
List_Directory()	OMS,PMS	PM,OM,CM,IOM
Get_Object()	OMS,PMS	PM,OM,CM,IOM
Put_Object()	OMS,PMS	PM,OM,CM,IOM
Append()	OMS,PMS,IOS	PM,OM,CM,IOM
Create()	OMS,PMS,IOS	PM,OM,CM,IOM
Delete()	OMS,PMS,IOS	PM,OM,CM,IOM
Create_Subsystem()	OMS,PMS,IOS	PM,OM,CM,IOM,DM
Destroy_Subsystem()	OMS,PMS,IOS	PM,OM,CM,IOM,DM
Modify_Subsystem()	OMS,PMS,IOS	PM,OM,CM,IOM,DM
Change_Protection()	SMS,PMS	PM,OM,CM
Create_Protection_Policy()	SMS,PMS	PM,OM,CM

Table 1: SMS Interface Operations

We now give a brief description of each service.

```
Run(program_name)
/*
  This service allows a user to execute the named program module.
  program_name is a string
*/
{
  (Search default user or public directories for program_name
```

```

    and return a capability for the program object)
    (Create an execution environment for the program object
    using the primitives of the kernel's process module)
    (Use the message system primitives to attach the processes
    output to an output object)
    (schedule the processes execution using the PMS)
}

Send(user _name,message _object _name)
/*
  This service allows a user to send a message to another user.
  It is a crude approximation of an electronic mail system.
  user _name is the string name of the user the message is being
  sent to, and message _object _name is the string name of the
  object holding the body of the message. It is assumed that the
  user has already created message _object _name.
*/
{
  (Lookup user _name in a name list for which a publicly
  accessible capability exists. Such a capability is made
  available to a user when his default environment is
  created, or anytime thereafter)
  (Get a capability for the message _object by searching the
  user's directory)
  (Create a message by using the MS primitive M _create _message()
  using the capability for the message _object)
  (Connect with user _name's port object using the
  M _channel _connect primitive)
  (Send the message using M _send _message)
  (Tear down the connection using M _channel _disconnect)
}

Receive(port _object,message _id,destination _object)
/*
  This service receives message _id from port _object and saves
  it in destination _object. This is the complement of send().
  port _object and destination _object are strings. message _id
  is an integer.
*/
{
  (Search the users current directory (or that specified) and
  get the capabilities for the two objects)
  (get the message from the port object using the primitive
  M _receive _message from the MS. M _receive _message expects
  the three parameters of Receive as capabilities and an
  integer)
  /* the user may view the message returned using the Get _Object()
  command */
}

Submit(program _list)
/*

```


This service allows the user to submit a set of program modules to be executed as a transaction. `program_list` is a list of string names representing the program modules. `Submit's` may be nested to some unspecified depth. For example, if a user wanted to create a number of objects as a transaction, the following would suffice:
`submit(create(file1),create(file2),.....,create(filen))`

```
*/  
{  
    /* The submit() itself is started as a top level transaction  
       using the transaction primitives of the TS and each member  
       member of the program_list is started as a subtransaction.  
       The submit completes successfully only if each subtransaction  
       commits successfully.  
    */  
}
```

`Make_directory(directory_name)`

```
/*  
    This service allows a user to create a new directory object  
    and link it into his directory structure. We assume a rooted-  
    tree structure.  
*/
```

```
{  
    (call the kernel to create an object and return a  
     capability for it)  
    (place the capability in the capability list of the current  
     directory at this level of the tree)  
    (associate directory_name with the capability by writing it  
     into the data part of the current directory object - thus  
     there is a one-to-one correspondence between the relative  
     position of the directory_name and the relative position  
     of the capability that points to the directory)  
    (return a success code to the user)  
    /*  
     since directories are expected to be create quite often,  
     and since they might be large, Make_directory might create  
     directories as special indexable objects which can be  
     searched rapidly. This creates an additional level of detail  
     since it is necessary to associate the index (object) with  
     the directory object. However, its implementation is  
     straightforward.  
    */  
}
```

`List_directory(directory_name)`

```
/*  
    This service simply lists the entries in the specified directory  
    along with useful object parameters in some default format.  
    directory_name is a string.  
*/
```

```
{  
    (perform a linear or indexed search of the string name in
```

```

        the data part of the directory object and retrieve the
        capability from the corresponding position in the C-list)
        (with the capability returned from above, get information
        about the object pointed to by the capability using the
        kernel primitive O_info())
        (repeat the first two steps for each entry in the directory
        object)
        (return suitably formatted results to the user)
    }

Put_object(object_name,object_specs)
/*
  This service allows a user to download an object to FTSS.
  object_name is the string name of the object and object_specs
  is a list of parameters specifying the characteristics of the
  object to be stored. object_specs will specify such things as
  the users blocking factor, the type of data (text,binary,etc.),
  and the amount of data to store.
*/
{
    (create an object to store the user's data and update the
    user's directory object)
    (establish communication with the user's host system using
    the message system primitives and the communication facilities
    between FTSS and the host - this is discussed in more detail
    in a later chapter)
    (get user blocks and pack them into FTSS blocks. As each FTSS
    block becomes full, write it to the object using the OM)
    (repeat the last step until all data is transferred)
    (write the object to the object store using the OMS)
    (tear down the connections established in the second step)
}

Get_object(object_name,object_specs)
/*
  This service allows a user to upload an object from FTSS.
  object_name is the string name of the object to be gotten,
  and object_specs is a list of specifications describing
  how much of the object is to be gotten.
*/
{
    /* the mechanisms are just the obverse of Put_object
    and will not be described further
    */
}

Append(object_name_list,new_object)
Create(object_name)
Delete(object_name)
/* The three services specified append a number of objects in
object_name_list to form a new object new_object, creates
a new object with the string name object_name, and deletes

```

an existing object with the string name `object_name`. Normally these operations are done within the hosts, but they are provided here for those hosts that wish to optimize their performance by having the operations performed by FTSS directly. Their details are elided since their semantics are simple - Create and Delete simply retrieves the capability and makes a direct call to the kernel's object manager. Append simply retrieves the capabilities for the objects and copies their contents into the destination object in the order specified in the name list.

*/

`Create_subsystem(subsystem_name)`

/*

This service permits a user to define an abstract data type which we call a subsystem. It is with this mechanism that the user can create any environment desirable.

*/

{

- (Create a new type object to represent the subsystem)
- (Create the instruction objects that define the operations of the new subsystem and initialize them)
- (Install the capabilities for the instruction objects into the type object)
- (Install capabilities for other objects which will be components of the type object)
- (Return capabilities for the new subsystem to the caller)

}

`Destroy_subsystem(subsystem_name)`

/*

With this service a user can destroy a subsystem which is no longer needed. It is not much different from the Delete command except that it takes care of any special cleanup operations required.

*/

{

- (delete the type object using the kernel primitive `O_destroy`)
- (remove the entry from the user's directory)

}

`Modify_subsystem(subsystem_name)`

/*

This service allows a user to modify an existing subsystem definition.

*/

{

- (Retrieve all capabilities from the existing type object and save in a temporary object)
- (Destroy the subsystem)
- (Create a new subsystem object with the same name)

```

        (interactively permit the owner to install capabilities
         from step 1 or new capabilities pointing to new
         components into the new type object)
        (return all capabilities to the caller)
        (update the user's directory)
    }

Create__protection__policy(policy__object__name)
/*
  This service permits a user to define protection policies to
  be applied individually or globally to owned objects. There can
  be any number of such policies in effect simultaneously (but not
  applied to the same object). A protection policy is an abstract
  type that determines how an object can be accessed. This is
  based on the technique of capability sealing.
*/
{
    (Create an object to represent the protection policy -
     this is a type definition object)
    (Install capabilities for instruction objects and other
     component objects into the object from the previous step -
     the instruction objects define the policy)
    (return capabilities for the policy object to the caller)
    (update the user's directory)
}

Change__protection(policy__object__name,object__name)
/*
  This service allows a user to install or change protection
  over owned objects. A policy object is required (see
  Create__protection__policy)
*/
{
    (get the capability for object__name from the user's
     directory)
    (seal this capability using the seal primitive from
     the SES - sealing requires the capabilities for the
     policy object and the object being protected as well
     as the information to be sealed in the capability -
     the reader is referred to chapter 4 for the details)
    (return the (sealed) capability to the user's directory)
}

```

REFERENCES

- [Acam84] A.S. Acampora and M.G. Hluchyj, "A New Local Area Network Architecture Using a Centralized Bus," *IEEE Communications Magazine*, Vol. 22, No. 8, August 1984, pp. 12-21.
- [Ahuj83] S.R. Ahuja, "S/NET: A High-Speed Interconnect for Multiple Computers," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, No. 5, November 1983, pp. 751-756.
- [Alsb76] P.A. Alsberg and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," in *Proceedings 2nd International Conference on Software Engineering*, October 1976, pp. 562-570.
- [Avi78] A. Avizienis, "Fault-Tolerance: The Survival Attribute of Digital Systems," *Proceedings IEEE*, Vol. 66, No. 10, October 1978, pp. 1109-1125.
- [Avi84] A. Avizienis and J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol. 17, No. 8, August 1984, pp. 67-80.
- [Bana79] J. Banarjee, D.K. Hsiao, and K. Kannan, "DBC - A Database Computer for Very Large Databases," *IEEE Transactions on Computers*, Vol. C-28, No. 6, June 1979, pp. 414-429.
- [Batc80] K.E. Batcher, "Architecture of a Massively Parallel Processor," in *Proceedings 7th Annual Symposium on Computer Architecture*, May 6-8, 1980, pp. 168-173.
- [Bern80] P.A. Bernstein and N. Goodman, "Fundamental Algorithms for Concurrency Control in Distributed Database Systems," Computer Corporation of America, February 1980.
- [Birm84] K.P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi, "Implementing Fault-Tolerant Distributed Objects," Department of Computer Science, Cornell University, Ithaca, New York, Tech. Rep. TR 84-594, March 1984.
- [Bour69] W.G. Bouricius, W.C. Carter, and P.R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," in *Proceedings 24th National Conference of the ACM*, 1969, pp. 295-383.

- [Carl80] D.E. Carlson, "Bit-Oriented Data Link Control Procedures," *IEEE Transactions on Communications*, Vol. COM-28, No. 4, April 1980, pp. 455-467.
- [Cham80] G.A. Champine, "Back-End Technology Trends," *IEEE Computer*, February, 1980, pp. 50-58.
- [Clar78] D. Clark *et al.*, "An Introduction to Local Area Networks," *Proceedings of the IEEE*, Vol. 66, No. 11, November 1978, pp. 1497-1517.
- [Cost78] A. Costes, C. Landrault, and J.C. Laprie, "Reliability and Availability Models for Maintained Systems Featuring Hardware Failures and Design Faults," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 548-560.
- [Denn82] D.E. Denning, *Cryptography and Data Security*, Menlo Park, California: Addison-Wesley, 1982.
- [Denn76] P.J. Denning, "Fault-Tolerant Operating Systems," *ACM Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 359-389.
- [Denn66] J.B. Dennis and E.C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, Vol. 9, No. 3, March 1966, pp. 143-155.
- [Desp78] A.M. Despain and D.A. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," in *Proceedings 5th Symposium on Computer Architecture*, April 1978, pp. 144-150.
- [DeWi79] D.J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Transactions on Computers*, Vol. C-28, No. 6, June 1979, pp. 395-406.
- [Dijk68] E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, F. Genuys, Ed. New York: Academic Press, 1968.
- [Dijk78] E.W. Dijkstra *et al.*, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, Vol. 21, No. 11, November 1978, pp. 966-975.
- [Dion80] J. Dion, "The Cambridge File Server," *Operating Systems Review*, Vol. 14, No. 4, October, 1980, pp. 26-35.

- [Edwa82] D. Edwards, "Implementation of Replication in LOCUS: A Highly Reliable Distributed Operating System," Master's Thesis, UCLA Computer Science Department, Los Angeles, California, 1982.
- [Ekan79] K. Ekanadham and A.J. Bernstein, "Conditional Capabilities," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, September 1979, pp. 458-464.
- [Ewin82] C.W. Ewing and A.M. Peskin, "The Masstor Mass Storage Product at Brookhaven National Laboratory," *IEEE Computer*, Vol. 15, No. 7, July 1982, pp. 57-66.
- [Fabr74] R.S. Fabry, "Capability-Based Addressing," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 403-412.
- [Frid81] M. Fridrich and W. Older, "The FELIX File Server," in *Proceedings Eighth Symposium on Operating System Principles*, Asilomar Conference Grounds, Pacific Grove, California: 14-16 December 1981, pp. 37-44.
- [Giff82] D.K. Gifford, "Information Storage in a Decentralized Computer System," Xerox Corporation, Palo Alto Research Centers, Palo Alto, California, Tech. Rep. CSL-81-8, March, 1982.
- [Glig79] V.D. Gligor, "Review and Revocation of Access Privileges Distributed Through Capabilities," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 6, November 1979, pp. 575-586.
- [Good81] J.R. Goodman and C.H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Transactions on Computers*, Vol. C-30, No. 12, December 1981, pp. 923-933.
- [Gray78] J.N. Gray, "Notes on Database Operating Systems," in *Operating Systems, An Advanced Course*, Springer-Verlag, 1978, pp. 393-481.
- [Grna81] A. Grnarov and M. Gerla, "Multiterminal Reliability Analysis of distributed Processing Systems," in *Proceedings 1981 International Conference on Parallel Processing*, August 1981, pp. 79-86.
- [Haye76] J.P. Hayes, "A Graph Model for Fault-Tolerant Computing Systems," *IEEE Transactions on Computers*, Vol. C-25, No. 9, September 1976, pp. 875-884.

- [Heym82] D.P. Heyman and M.J. Sobel, *Stochastic Models in Operations Research, Volume 1*, San Francisco, California: McGraw-Hill, Inc., 1982.
- [IBM78] IBM, "System/38 Technical Developments," International Business Machines Corporation, 1978.
- [Jone73] A.K. Jones, "Protection in Programmed Systems," PhD Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, June 1973.
- [Jone79a] Anita K. Jones, "The Object Model: A Conceptual Tool for Structuring Software," in *Operating Systems: An Advanced Course*, R. Bayer *et al.*, Ed. New York: Springer-Verlag, 1979, pp. 7-16.
- [Jone79b] A.K. Jones, "Protection Mechanisms and the Enforcement of Security Policies," in *Operating Systems - An Advanced Course*, R. Bayer, *et al.*, Ed. New York: Springer-Verlag, 1979, pp. 228-250.
- [Kats78] D. Katsuki *et al.*, "Pluribus - An Operational Fault-Tolerant Multiprocessor," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1146-1159.
- [Kern78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1978.
- [Khan81] K.C. Khan *et al.*, "iMAX: A Multiprocessor Operating System for an Object-Based Computer," in *Proceedings 8th Symposium on Operating System Principles*, December, 1981, pp. 127-136.
- [Knut73] D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*: Addison-Wesley, 1973.
- [Kwan81] C.L. Kwan and S. Toida, "Optimal Fault-Tolerant Realizations of Some Classes of Hierarchical Tree Systems," in *Proceedings 11th International Symposium on Fault-Tolerant Computing, FTCS-11*, June, 1981, pp. 176-178.
- [Lamp82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.
- [Lamp69] B.W. Lampson, "Dynamic Protection Structures," in *Proceedings AFIPS FJCC*, Montvale, N.J.: 1969, pp. 27-38.

- [Lamp81] B.W. Lampson, "Atomic Transactions," in *Distributed Systems - Architecture and Implementation, An Advanced Course*, B.W. Lampson, M. Paul, and H.J. Siebert, Ed. New York: Springer-Verlag, 1981, pp. 246-264.
- [Lapr75] J.C. Laprie, "Reliability and Availability of Repairable Structures," in *Proceedings FTCS-5*, November 1975, pp. 87-92.
- [Lawr82] D.H. Lawrie, J.M. Randal, and R.R. Barton, "Experiments with Automatic File Migration," *Computer*, Vol. 15, No. 7, July 1982, pp. 45-55.
- [Lisk74] B.H. Liskov and S.N. Ziller, "Programming with Abstract Data Types," *Proceedings ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices*, Vol. 9, No. 4, April 1974, pp. 50-59.
- [Maka81] S.V. Makam and A. Avizienis, "Modeling and Analysis of Periodically Renewed Closed Fault-Tolerant Systems," in *Proceedings FTCS-11: The Eleventh International Symposium on Fault-Tolerant Computing*, Portland, Maine: June 1981, pp. 134-141.
- [Maka82a] S.V. Makam and A. Avizienis, "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault-Tolerant Systems," in *Proceedings FTCS-12: The Twelfth Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California: June 1982, pp. 267-274.
- [Maka82b] S.V. Makam and A. Avizienis, "Aries 82 Users' Guide," University of California, Los Angeles, Tech. Rep. CSD-820830, 1982.
- [Maka83] S.V. Makam and C.S. Raghavendra, "Dynamic Reliability Modeling and Analysis of Computer Networks," in *Proceedings 1983 International Conference on Parallel Processing*, 1983.
- [McCr80] T.D. McCreery, "The X-Tree Operating System: Bottom Layer," in *Proceedings IEEE Spring COMPCON*, 1980, pp. 340-343.
- [Metc80] R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Xerox, Palo Alto Research Center, Palo Alto, California, Tech. Rep. CSL-75-7, February 1980. Also appears in *Communications of the ACM*, Vol.19 no.7, July 1976.

- [Moss81] J. Eliot B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," M.I.T. Laboratory for Computer Science, Tech. Rep. MIT/LCS/TR-260, 1981.
- [Muel83] E.T. Mueller, "Implementation of Nested Transactions in a Distributed System," MS Thesis, University of California, Los Angeles, 1983.
- [Neum77] Peter G. Neumann *et al.*, "A Provably Secure Operating System: The System, Its Applications, and Proofs," Stanford Research Institute, Menlo Park, California 94025, Tech. Rep. SRI Project 4332, February 1977.
- [Orga72] E.I. Organick, *The Multics System: An Examination of its Structure*, Cambridge, Massachusetts: MIT Press, 1972.
- [Park82] D.S. Parker and R. Ramos, "A Distributed File System Architecture Supporting High Availability," in *Proceedings Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, Asilomar, California: February, 1982, pp. 161-183.
- [Poll81] F.J. Pollack, K.C. Khan, and R.M. Wilkinson, "The iMAX-432 Object Filing System," in *Proceedings 8th Symposium on Operating System Principles*, December, 1981, pp. 137-147.
- [Pope81] G. Popek *et al.*, "LOCUS: A Network Transparent, High Reliability Distributed System," in *Proceedings 8th Symposium on Operating Systems Principles*, December 1981, pp. 169-177.
- [Pope78] G.J. Popek *et al.*, "UCLA Data Secure Unix: A Securable Operating System - Software Architecture," UCLA Internal Memorandum, Los Angeles, California, August 1978.
- [Pope79] G.J. Popek and C.S. Kline, "Issues in Kernel Design," in *Operating Systems - An Advanced Course*, R. Bayer *et al.*, Ed. New York: Springer-Verlag, 1979, pp. 209-227.
- [Ragh83] C.S. Raghavendra, A. Avizienis, and M. Ercegovic, "Fault-Tolerance in Binary Tree Architectures," in *Proceedings Thirteenth Annual International Conference on Fault-Tolerant Computing*, Milano, Italy: June 28-30, 1983, pp. 360-364.
- [Rede74] D.D. Redell, "Naming and Protection in Extendible Operating Systems," M.I.T. Project Mac, Tech. Rep. MAC TR-140, November 1974.

- [Reed78] D.P. Reed, "Naming and Synchronization in a Decentralized Computer system," M.I.T. Laboratory for Computer Science, Tech. Rep. TR-205, September, 1978.
- [Renn80] D.A. Rennels, "Distributed Fault-Tolerant Computer Systems," *Computer*, Vol. 13, No. 3, March 1980, pp. 55-65.
- [Ritc74] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.
- [Salt80] J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design," in *Proceedings 2nd. International Conference on Distributed Systems*, Paris, France: April, 1980, pp. 509-512.
- [Saty81] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes," in *Proceedings 8th Symposium on Operating Systems Principles*, December 1981, pp. 96-108.
- [Sche80] R.R. Schell and L.A. Cox Jr., "A Secure Archival Storage System," in *Proceedings 1980 Fall Compcon*, San Francisco, California: 1980, pp. 679-682.
- [Sequ79] C.H. Sequin, "Single-Chip Computers, The New VLSI Building Blocks," in *Proceedings Caltech Conference on VLSI*, Pasadena, California: January, 1979, pp. 435-445.
- [Shoc80] J.F. Shoch and J.A. Hupp, "Measured Performance of an Ethernet Local Network," Xerox, Palo Alto Research Center, Palo Alto, California, Tech. Rep. CSL-80-2, February 1980.
- [Smit81a] A.J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 4, July 1981, pp. 403-417.
- [Smit81b] A.J. Smith, "Long Term File Migration: Development and Evaluation of Algorithms," *Communications of the ACM*, Vol. 24, No. 8, August 1981, pp. 521-532.
- [Spec83] A.Z. Spector and P.M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing," *Operating Systems Review*, Vol. 17, No. 2, April 1983, pp. 18-35.
- [Su79] S.Y.W. Su, A. Emam, and G.J. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Transactions on Computers*, Vol. C-28, No. 6, June 1979, pp. 430-445.

- [Svob81] L. Svobodova, "A Reliable Object-Oriented Data Repository for a Distributed Computer System," in *Proceedings 8th Symposium on Operating System Principles*, December, 1981, pp. 47-58.
- [Svob82] L. Svobodova, "File Servers for Network-Based Distributed Systems," IBM Zurich Research Laboratory, Switzerland, Tech. Rep. RZ 1186 (#42689), November 1982.
- [Swin79] D. Swinehart, G. McDaniel, and D. Boggs, "WFS: A Simple Shared File System for a Distributed Environment," Xerox Palo Alto Research Center, Palo Alto, California, October, 1979.
- [Thom79] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979, pp. 180-209.
- [Thor80] J.E. Thornton, "Back-End Network Approaches," *IEEE Computer*, February, 1980, pp. 10-17.
- [Wats80] R.W. Watson, "Network Architecture Design for Back-End Storage Networks," *IEEE Computer*, February, 1980, pp. 32-48.
- [Wulf81] W. Wulf, R. Levin, and S. Harbison, *Hydra/C.mmp: An Experimental Computer System*: McGraw-Hill Book Company, 1981.
- [Zimm80] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications - Special Issue on Computer Network Architectures and Protocols*, Vol. COM-28, No. 4, April 1980.