Technical Report

UCAM-CL-TR-375
ISSN 1476-2986

Number 375

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Restructuring virtual memory to support distributed computing environments

Feng Huang

July 1995

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 1995 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-375*

Acknowledgements

I am most grateful to Jean Bacon, my supervisor, for her patience, constant encouragement, useful discussions and valuable support during the course of this research. Thank you, Jean.

I would like to thank members of the Computer Laboratory, specially Glenford Mapp, Sai-Lai Lo, Zhixue Wu, and Innes Ferguson for their valuable discussions and encouragement. Thanks to the past and present members of the System Research Group who develop and maintain the Wanda system which was used for the experimental work in this research. Also, thanks to Ken Moody and other members of the Opera project for providing me the opportunities to present this work and to get useful comments and practical assistance in the project meetings.

Many thanks to the following people who have read and suggested improvements to this dissertation: Jean Bacon, Glenford Mapp, Innes Ferguson, Mohamad Afshar and Shaw Chuang.

Special thanks to Roger Needham, the head of the Computer Laboratory, for his advice and help and making departmental support facilities available during this research. I also want to thank Martyn Johnson, Graham Titmus, Piete Brookes and Chris Hadley for providing much needed system support, and Lewis Tiffany and Paola Bishop for great assistance in the library.

I am very deeply indebted to my parents, school teachers and friends for their strong support. Without their support, I would not have been able to finish my school education, not to mention pursuing my PhD study at Cambridge. I also thank my parents for their incredible understanding and encouragement which kept me going for these years. Thanks also to my wife Chen Yue for her lovely letters and her emotional support during the last stage of this research.

This work is supported by a Sir Run Run Shaw Scholarship and a Caius Bursary from the Cambridge Overseas Trust, an Overseas Research Students Award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom, and a Research Studentship from the Cambridge Philosophical Society. All these financial supports are highly appreciated. I also thank the Computer Laboratory and Clare Hall for the financial support which enabled me to attend the Advanced Course on Distributed Systems in Lisbon and for two-month living expenses which enabled me to complete this research.

4 Design of the Coherence Mechanism **31**
4.1 Introduction 31
4.2 Design Considerations 32
4.2.1 Granularity 32
4.2.2 Remote Interprocess Communication 33
4.2.3 Writing Modifications to Backing Store 35
4.3 Public Interface and Related Issues 36
4.3.1 Public Interface 36
4.3.2 Page-Based Lock 36
4.3.3 Deadlock Prevention 37
4.4 Coherence Protocols 39
4.4.1 Write-Invalidate Protocols 39
4.4.2 Write-Update Protocol 41
4.4.3 Integrated Coherency Control 41
4.5 Coherence Manager and Coherence Server 42
4.6 Illustration of Coherency Control 43
4.6.1 Mapping An Object and First Access 43
4.6.2 Object Coherency 43
4.6.3 Unmapping An Object 47
4.7 Summary 47

5 Implementation **48**
5.1 System Environment 48
5.2 Object Management 49
5.3 Persistent Object Manager 50
5.4 Coherence Manager 50
5.4.1 Server Identifier 51
5.4.2 Object Management 51
5.4.3 Communication Management 52
5.4.4 Other Modules 53
5.5 Coherence Server 53
5.6 Storage Server Emulator 54
5.7 Summary 54

6 Performance **55**
6.1 Introduction 55
6.2 Performance of the RPC System 56
6.2.1 Simple RPC vs MultiRPC 57
6.2.2 MultiRPC Speed-Up 57
6.2.3 Summary 59
6.3 Memory-Mapping vs Non Memory-Mapping on Wanda 59

6.4 Performance of the Prototype COMMOS 61
6.4.1 No Coherency 61
6.4.2 Centralised-Control Protocol 61
6.4.3 Distributed-Control Protocol 62
6.5 Summary 63

7 Supporting Distributed Persistent Programming **65**
7.1 Introduction 65
7.2 Defining C++ Class Operators 67
7.3 Overloading C++ Operator 70
7.4 Supporting Fine Grained Objects 72
7.5 Other Issues 74
7.5.1 Working with Pointers 75
7.6 Summary 76

8 Related Work **77**
8.1 Apollo Domain 77
8.2 Mach 78
8.3 Chorus 80
8.4 The V System 81
8.5 Clouds 81
8.6 Choices 82
8.7 Spring 83
8.8 Comandos 84
8.8.1 Amadeus 85
8.8.2 COOL 85
8.8.3 Guide-2 86
8.9 Casper 86
8.10 Opal 87
8.11 Pegasus 88
8.12 Comparison 88

9 Conclusions **90**
9.1 Conclusions 90
9.2 Further Work 92
9.2.1 More Flexible Coherence Schemes 92
9.2.2 Support for Heterogeneous Architectures 93
9.2.3 Exploitation of COMMOS Functionality 93
9.3 Final Word 94

Bibliography **95**

A Public Interface 105

B VMM and POM Interaction 108

C Coherence Protocols 111
C.1 Write-Invalidate Protocols 111
C.1.1 Centralised-Control Protocol 111
C.1.2 Distributed-Control Protocol 113
C.2 Write-Update Protocol 116

D Coherence Manager 120

E Coherence Server 123

F Storage Server Emulator 126

G Performance Measurements 128
G.1 RPC Performance 128
G.2 Page Fault, Invalidation and Roundtrip IPC 132
G.3 Performance of the COMMOS Prototype 133

List of Figures

2.1 Monolithic Kernel vs. Microkernel 6
2.2 File Access in Monolithic Kernels and Microkernels 9
2.3 Buffer Management in DBMS 10
2.4 Double Paging Problem 11
2.5 Shared Virtual Memory 13
2.6 Various Approaches to Caching in Client Memory 15
2.7 An Example of Accessing a Persistent Object 18

3.1 The COMMOS Layers 23
3.2 COMMOS Architecture 28

4.1 Self-Deadlock When Fine-Grained Locks Are Requested 37
4.2 Inter-Thread Deadlock When Fine-Grained Locks Are Requested 38
4.3 Opening An Object and the First Access (All Protocols) 43
4.4 Illustration of the Centralised-Control Protocol 44
4.5 Illustration of the Distributed-Control Protocol 45
4.6 Illustration of the Write-Update Protocol 46

6.1 Prototype Configuration 56
6.2 Speed-up for One-to-Many Communications on Wanda 58
6.3 Speed-up for One-to-Many Communications on Ultrix 59

B.1 VMM and POM Interaction Protocol 110

List of Tables

6.1 Simple RPC vs MultiRPC for One Recipient 57
6.2 Performance of MultiRPC on Wanda 57
6.3 Performance of MultiRPC on Ultrix 58
6.4 Performance of A Local Roundtrip IPC 60
6.5 Time to Serve A Page Fault and Invalidate A Page 60
6.6 Fetching a Page from the Storage Server 61
6.7 Fetching a Page for Read in the Centralised-Control Protocol . . 61
6.8 Fetching a Page for Write in the Centralised-Control Protocol . . 62
6.9 Fetching a Page for Read in the Distributed-Control Protocol . . 63
6.10 Fetching a Page for Write in the Distributed-Control Protocol . . 63

A.1 Characteristics for Mapping Objects 106

G.1 Hardware Configuration for Performance Measurements 128
G.2 Simple RPC vs MultiRPC for One Local Recipient 129
G.3 Simple RPC vs MultiRPC for One Remote Recipient 129
G.4 MultiRPC to Two Remote Recipients 130
G.5 MultiRPC to Three Remote Recipients 130
G.6 Simple RPC between Wanda and Ultrix on a DECStation 131
G.7 MultiRPC from Ultrix on DECStation to Wanda 131
G.8 Page Fault, Invalidation, Roundtrip IPC and System Call 132
G.9 Fetching a Page from the StorSvr 133
G.10 Fetching a Page for Read in the Centralised-Control Protocol . . 133
G.11 Fetching a Page for Write in the Centralised-Control Protocol . . 134
G.12 Fetching a Page for Read in the Distributed-Control Protocol . . 134
G.13 Fetching a Page for Write in the Distributed-Control Protocol . . 135

Glossary

This list defines abbreviations used in the text. Each entry ends with the number of the page on which the term is introduced.

ACL	Access Control List (24)
API	Application Programming Interface (12)
ATM	Asynchronous Transfer Mode (4)
COMMOS	Coherent Memory-Mapped Object System (1)
CoherSvr	Coherence Server (24)
CoherMgr	Coherence Manager (27)
CSCW	Computer-Supported Cooperative Work (93)
DSM	Distributed Shared Memory (2)
FFC	Flat File Custode (54)
LAN	Local Area Network (14)
LWP	Light-Weight Process (50)
MMU	Memory Management Unit (24)
MSSA	Multi-Service Storage Architecture (24)
POM	Persistent Object Manager (24)
ProcSvr	Process Server (29)
RAM	Random Access Memory (4)
RPC	Remote Procedure Call (12)
StorSvr	Storage Server (49)
SVM	Shared Virtual Memory (12)
VMM	Virtual Memory Management (23)

Chapter 1

Introduction

This dissertation considers traditional approaches to virtual memory and storage management and observes their limitations, particularly in the emerging distributed computing environments. A coherent memory-mapped object system (COMMOS) architecture is then proposed to overcome these limitations.

1.1 Motivation

Distributed systems, which consists of workstations connected by different types of networks are emerging as the mainstream architecture of computing. The decrease in cost of physical memory and advances in hardware, such as the widening of the address space and the increase of network speed, have led to new thinking in software design. At the same time, microkernel architectures have emerged to ease the engineering of operating systems and to facilitate the development of higher levels of software.

The traditional approach to virtual memory and storage management is based on the two-level store architecture. It provides an interface defined by programming languages to access conventional memory segments and another interface defined by the file systems to access persistent data residing in secondary storage. The fact that there are two different views of volatile data and persistent data causes inconvenience to programmers and inefficiency for constructing software systems. The two-level store architecture also compromises operating system efficiency because of mandatory data copying and unnecessary user/kernel boundary crossings (or context switches in the microkernel architecture). Also limitations exist when database management systems are

built on top of this kind of memory and storage architecture.

In a wide address space architecture, distributed shared memory (DSM), an abstraction used for sharing data between processes in computers that do not share physical memory, is becoming increasingly attractive because it is easier to program than a message-passing abstraction. Distributed file systems rely on caching at clients to improve their performance. The cache coherency problem is common to these two paradigms. Current DSM systems and distributed file systems typically provide only one coherence protocol. There exists a potential mismatch between the supplied protocol and some applications and it is desirable that a more flexible approach should be employed.

Memory mapping provides a uniform view of both volatile and persistent data. It also provides better performance for accessing persistent data because information copying is no longer mandatory and the number of user/kernel boundary crossings (or context switches in microkernel architectures) is reduced. This is not a new technique but it was not very successful in the early days. The main reason is that there was no suitable hardware technology and software environment. It is now time to reexamine this technique and to explore whether it can be used, with careful design, to overcome most of the problems outlined above.

1.2 Research Goal

The major goal of this research is to explore whether it is feasible to provide an integrated virtual memory framework which is general enough to support a range of higher level distributed system abstractions and distributed applications and which is flexible enough to meet different requirements for these abstractions and applications at a reasonable cost.

The approach taken combines the microkernel, memory-mapping and typed memory object principles to provide a minimum of support for distributed computing environments with the aim of making virtual memory and storage management integrated, flexible, and easy to use. It is proposed that virtual memory management cooperate with other services in an open system in order to satisfy all application requirements without sacrificing performance. Virtual memory and storage management are integrated by using the memory-mapping technique.

It is important to support both memory coherency and concurrency control in

a distributed system. However, it is recognised that no single universal coherence protocol can meet the requirements of all applications. In this work, the coherence server is separated from the external pager so that a variety of coherence protocols can be provided in a generic interface. Applications can advise the system which protocol would be used while a default protocol is supported. The low-level cache coherency control is integrated with the high-level concurrency control so that the system-wide object coherency and synchronisation are realised without severely impacting the system performance.

This dissertation shows how this goal can be achieved by a prototype implementation of COMMOS and demonstrates that the approach is feasible.

1.3 Dissertation Outline

Chapter 2 reviews the technological developments of hardware and software and discusses the limitations of existing approaches to memory and storage management when they are applied to distributed computing environments. It also gives a brief introduction to memory-mapping techniques. This chapter concludes that a new approach to integrate main memory with secondary storage and local memory with remote memory is desirable.

Chapter 3 discusses requirements and outlines an architectural framework for the coherent memory-mapped object system (COMMOS), including a brief description of those aspects not directly related to coherency control. A piece of previous work and its relation with COMMOS are also given.

Chapter 4 focusses on the design of the COMMOS coherence mechanism and Chapter 5 gives the details of the prototype implementation. Chapter 6 evaluates the performance of the prototype implementation and provides the experimental evidence for some of the discussions in Chapter 2 and Chapter 4.

Chapter 7 illustrates the use of COMMOS by discussing how to build a C++ class library to support distributed persistent programming.

Chapter 8 reviews the systems and projects which are closely related to this work and compares their approaches to important issues with those of COMMOS.

Chapter 9 concludes the dissertation by summarising the work and suggesting further research.

Chapter 2

Background

2.1 Introduction

Computer technology has been developing very rapidly. The performance of modern microprocessors is increasing at about 35% per annum [Bacon93] while the price is constantly decreasing. Distributed systems consisting of workstations connected by different types of networks are becoming a great challenge to traditional mainframe architectures.

Several recent developments will have significant impact on future computing environments. First, the cost of random access memory (RAM) is declining. The memory size of personal workstations quadruples every three years and hundreds of megabytes of physical memory will be commonplace [Bacon93, Needham91]. Second, wide address space machines, such as the 64-bit DEC Alpha [Sites92], HP PA-RISC [Lee89] and MIPS R4000 [Kane92] are becoming ubiquitous. These wide-address space architectures are interesting, not only because they allow applications to use almost arbitrarily large files and data structures, but also because they may fundamentally change the way operating systems are structured. Third, the speed of computer networks is increasing dramatically. Transmission rates of gigabits per second will soon be widely available. Even the components of a workstation might be connected by high speed ATM networks [Hayter93]. New thinking is necessary on how to build software, especially operating systems, to fully make use of all these new developments and to provide users with more functionality and better performance.

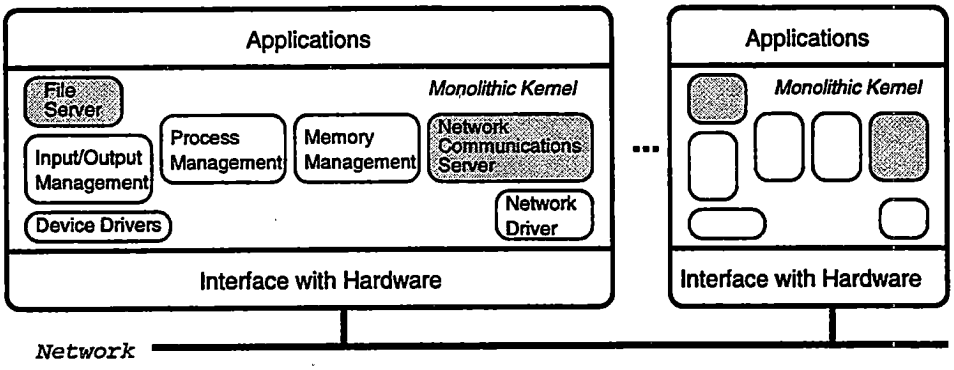
2.1.1 Microkernels

An important trend in operating system design to meet these technological developments is to restructure the operating system as a modular set of system servers sitting on top of a lightweight *microkernel* [Needham91, Gien91], rather than using the traditional monolithic structure. Conventional monolithic operating systems have become large and unwieldy. They are difficult to comprehend, develop and maintain. Meanwhile, distributed systems place different functions on different machines so not all operating system functions are needed in every copy of the operating system. This has resulted in a shift towards microkernels. The new approach promises to help meet systems and platform builders' needs for a sophisticated operating system structure that can cope with growing complexity, new architectures and changing market conditions. It maintains that the kernel should provide only the most basic functionality, with the bulk of the operating system services available from user-level servers. The microkernel provides system servers with generic services independent of a particular operating system, which typically include [Bacon93, Tanenbaum92, Coulouris94, Gien91]:

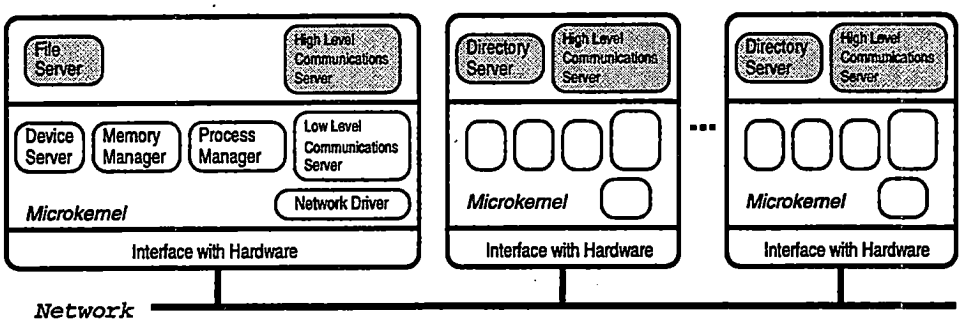
- low-level process management and scheduling;
- minimum memory management;
- simple inter-process communication facilities;
- I/O device management.

This combination of elementary services forms a base which can support all other system-specific functions. These can then be configured into appropriate system servers, managing the other physical and logical resources of a computer system, such as files, devices, much of memory management and high-level communications handling. The system servers usually run as user-level processes. Figure 2.1 shows the difference between conventional monolithic kernels and modern microkernels. In a monolithic kernel architecture, all the system services such as file service and network communication service run in the kernel and all kernels have the same configuration. By contrast, in a microkernel architecture, the high-level system services run at the user-level and different nodes may be configured in different ways.

The major advantages of a microkernel system are its openness and its ability to enforce modularity behind memory protection boundaries. There is a well-defined interface to each service and every service is equally accessible to every



(a) Monolithic Kernel



(b) Microkernel

Figure 2.1: Monolithic Kernel vs. Microkernel

client. In addition, it is easy to implement, install and debug new services, since adding or changing a service need not stop the system and booting a new kernel, as is the case with a monolithic kernel. The price/performance and reliability can also be enhanced because it is easier to incorporate new hardware connection technologies and new processors than in monolithic kernel architectures. A potential disadvantage is that the performance of accessing services at user-level might not be as good as accessing services in the kernel in a monolithic kernel system because of message handling, kernel boundary crossing and context switching overheads [Zahorjan91, Bricker91, Coulouris94].

Since being introduced by [Lampson79], microkernel architectures have been the subject of a great deal of operating system research, illustrated by projects such as Amoeba [Mullender90] (Vrije University and Centre for Mathematics and Computer Science, the Netherlands), Chorus [Rozier88] (Chorus Systems, France), Mach [Accetta86] (Carnegie Mellon University, USA), Spring [Hamilton93] (Sun Microsystems Laboratories Inc, USA) and the V System [Cheriton88a] (Stanford University, USA). It is likely that microker-

nel systems will gradually come to dominate the distributed systems scene, and monolithic kernels will eventually vanish or evolve into microkernels.

Memory and storage management play an important role in operating systems and have a significant effect on the system performance and usage. There are still a lot of issues in this area to be explored in order to provide users more functionality and better performance in future distributed systems. In the following sections, existing virtual memory and storage management are examined and the advantages of mapping objects into virtual memory address spaces are discussed.

2.2 Existing Virtual Memory and Storage Management

This section firstly examines the existing memory and storage management approaches and then considers their distributed paradigms, namely *distributed shared memory* (DSM) and *distributed file service*.

2.2.1 Two-Level Store Interface

Currently, users of most operating systems use one interface defined by the programming language to access conventional memory segments and another interface defined by the file service to access objects residing in secondary storage.

Virtual Memory

Since its introduction in the Atlas computer system [Fotheringham61] in 1961, *virtual memory* has been employed in the design of memory management of most operating systems. User programs are allowed to use a large contiguous set of virtual addresses, called a *virtual address space*, which may be larger than the total amount of physical memory available on the computer. Portions of this virtual address space are loaded into physical memory as they are needed and the rest are kept on the backing store. The physical memory and the virtual memory are usually divided into *segments* and each segment may be further divided into fixed-size *pages*, which are the smallest units for transferring to and from the backing store. The physical memory is in fact

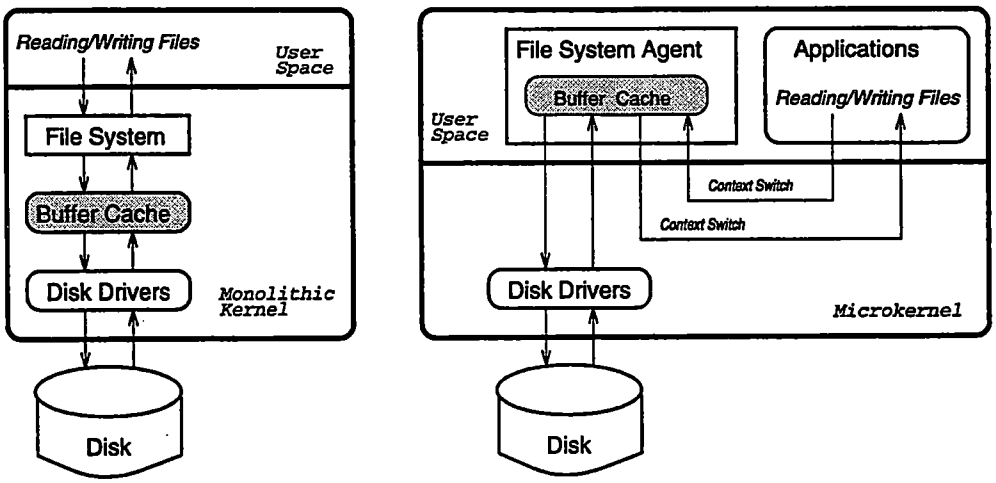
the cache of the virtual memory. Because the price of RAM is decreasing significantly, some suggest that by purchasing enough physical memory, virtual memory will not be needed in the future. However, this solution will not work for timesharing systems or for applications, such as databases, whose memory usages scale with CPU speed and scale faster than the decreasing rate of RAM price [Krueger93]. Meanwhile, because a virtual memory implementation may need to use a backing store on a separate computer from the one that contains the main memory and it is possible to share data which is simultaneously mapped into the address spaces of processes residing at different computers, virtual memory is of considerable interest as an aspect of the design of distributed operating systems.

In most of the commonly used systems, virtual memory segments are used by programming languages to store volatile information and the format of volatile information is different from that of persistent information stored in the file service.

File Service

Traditionally, information which needs to persist is stored on disks and other external media in units called *files*. The part of the operating system dealing with files is known as the *file system*. To access long-term objects stored in such a file system, data is first fetched to the system I/O buffers which typically reside in kernel space and is then copied to the users' address spaces as read operations are executed. On writing, data is copied from the users' address spaces to the system I/O buffers before it can be written to the secondary storage. If lots of small data items are to be accessed one by one, many unnecessary user/kernel crossings are involved. When accessing large amounts of data, on the other hand, significant extra time has to be spent on copying data between I/O buffers and the users' address spaces. Data copying and boundary interactions between kernel and user space make the system inefficient.

In the case that the file service runs as a user-level process in a microkernel operating system, if this traditional access model is still adopted, the overall performance becomes unacceptable. The major problem appears to be the need for costly context switching [Welch91]. Figure 2.2 shows file access using the traditional model in both monolithic kernel and microkernel operating systems. In order to perform comparably to monolithic systems, a microkernel system must either make a context switch much faster, or somehow avoid it



(a) Monolithic Kernel (b) Microkernel

Figure 2.2: File Access in Monolithic Kernels and Microkernels

wherever possible. Data movement must also be carefully designed to avoid extra copying of data. [Dean92]

Database Systems

Conventional database systems usually allocate buffer pools within their own virtual address spaces and provide their own buffer management facilities [Bacon93, Korth91, Ozsu91, Traiger82, Stonebraker81].

The overhead for data access in such a system is high. Figure 2.3 illustrates the data movement path in a DBMS system implemented on top of a two-level store system. Data pages are copied into a file buffer and then into a DBMS buffer before data fields are copied into an application program's variables. On writing, the path is traversed in reverse order.

Accessing the DBMS buffers may not only cause buffer faults to bring in data pages, but may also cause memory faults since the buffer page may not be mapped to a page frame and there may not be any free frames. This phenomenon is commonly called the *double paging* [Ozsu91, Chew92]. For example, if a DBMS is accessing page X which is not in its buffer pool, the buffer manager accesses the secondary storage where the database is located and then maps page X onto a virtual buffer page. This may involve replacing some page Y. However, at the virtual memory level, the frame corresponding

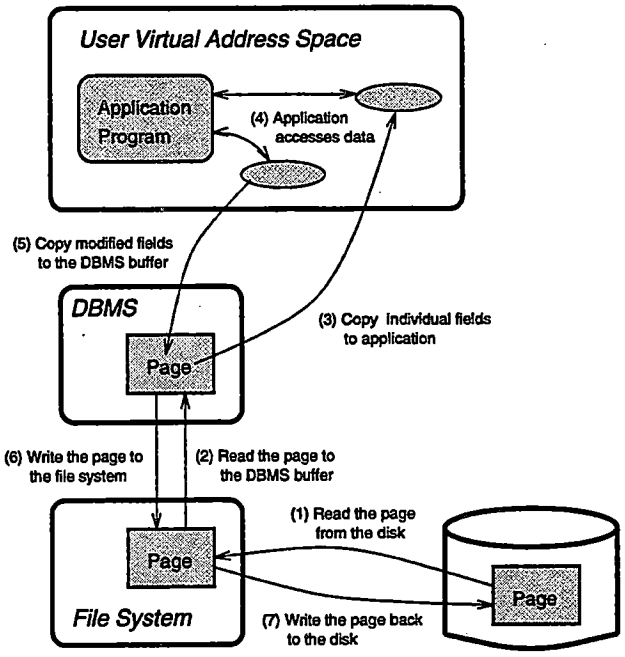


Figure 2.3: Buffer Management in DBMS

to Y may happen to be paged out. The virtual memory manager may have to service a real memory page fault to bring in the frame Y and then write it back to the permanent secondary storage (see figure 2.4).

There are also some other disadvantages. First, unmodified pages need to be written to the paging store at least once instead of simply being discarded. Data may be redundantly stored both in the database and the paging store. Second, double paging may lead to *double paging anomaly* [Goldberg74], where a significant increase in the number of page faults occurs with an increase in buffer pool size without a corresponding increase in physical memory. Besides, access control and address translation are accomplished in software. Available virtual memory hardware is not exploited.

Persistent Programming

Traditional programming languages provide facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. If data is required to survive a program activation some file I/O or database management system interface is used. Two views of data follow from this. Data can be classed as either short term and would be manipulated by the program-

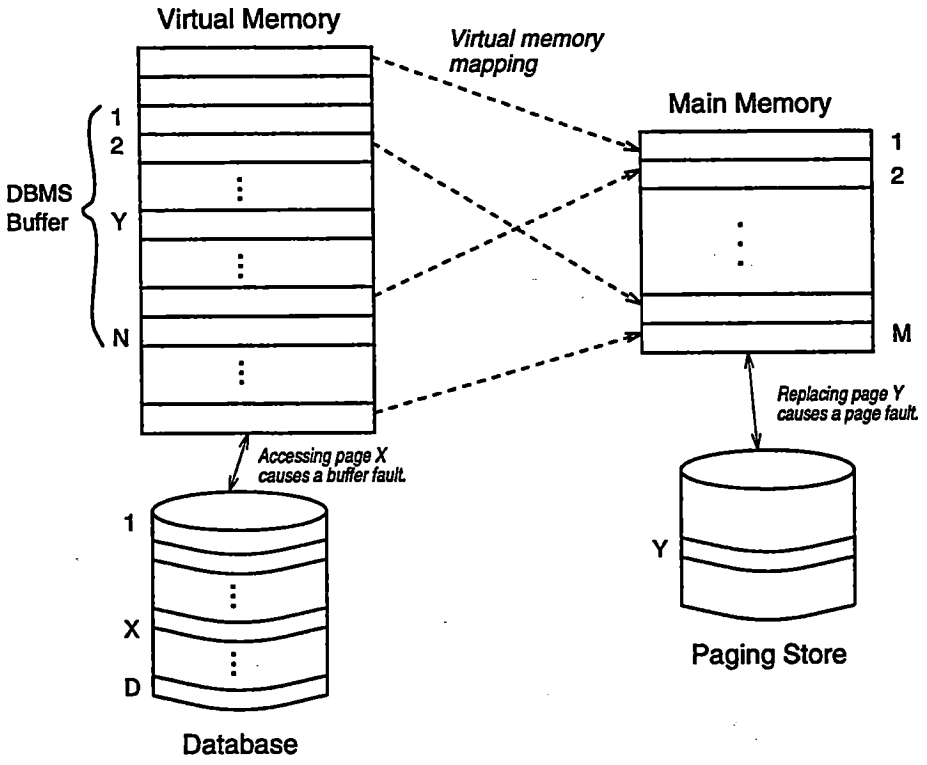


Figure 2.4: Double Paging Problem

ming language facilities or long term in which it would be manipulated by the file system or the database management system. The mapping between the two types of data is usually done in part by the file system or the DBMS and in part by explicit user translation code. The explicit user translation code has to be written and included in each program. There is usually a considerable amount of code, typically 30% of the total in many programs concerned with transferring data to and from files. This includes the code concerned with the explicit movement of data between main and backing store and the code required to change the representation of the data for long term preservation and restoration. An example of the former is input and output code and of the latter is code to flatten and reconstruct a graph before and after output and input respectively. This is unsatisfactory because of the time taken in writing this mapping code and also because the quality of the application programs may be impaired by the mapping. Frequently the programmer is distracted from his or her main task by the difficulties of understanding and managing the mapping [Atkinson83, Morrison90].

Persistent programming [Atkinson83] is a relatively new concept which makes data intensive application programming much easier. The idea is that data in a system should be able to persist for as long as it is required. It eliminates the differences between the DBMS and programming language models of data. Persistent programming systems support long lived data objects of arbitrary complexity. Such data objects may not only outlive instantiations of the program that created them, but also outlive versions of the program, or even the useful life of the program in all its versions.

To date, most persistent programming systems have been constructed above conventional operating systems. Implementors of persistent programming languages have to take care of the data movements and translations by themselves. If the two interfaces to volatile and persistent data can be harmonised in the operating system the language implementors can access objects in a unified manner without worrying about the need to move objects to and from secondary storage.

2.2.2 Distributed Shared Memory

In a typical distributed environment, since there is no shared physical memory, language-level user processes on different network nodes usually communicate with each other by using *message passing* or its higher level abstraction — *remote procedure call* (RPC) [Birrell84]. With a message-based application programming interface (API), programmers have to be concerned with transferring control between processes even if the interaction between these processes is limited to sharing data. A very simple situation is programmed in a complicated way. Second, the regular control flow of the system is mixed with the exchanges of control that implement the sharing of data, and the result is a system that is hard to debug. Finally, it is difficult and inefficient to pass complex data structures via message passing [Bisiani88, Herlihy82].

A *distributed shared memory* abstraction [Coulouris94, Hemmendinger92, Nitzberg91] has been proposed to cope with this problem. In this model communication recedes into the background. Programmers are spared from the concerns of message passing when writing applications that might otherwise have to use it. Applications see a single shared virtual address space as if shared memory were provided by the hardware or shared data objects. The former model is called *shared virtual memory* (SVM) [Li86].

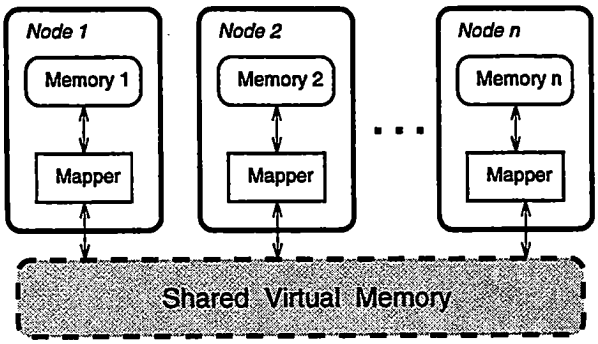


Figure 2.5: Shared Virtual Memory

Shared Virtual Memory SVM extends the idea of virtual memory to a distributed system, providing a single virtual address space which is shared among all nodes in a multicomputer system, as illustrated in figure 2.5. Any node in the system can access any memory location in the address space. Application programs can use the SVM in the same way as they use a traditional virtual memory, except that processes can run on different machines in parallel [Li86]. The memory mappers in a SVM system implement the mapping between local memories and the shared virtual address space. They maintain the coherency of the address space. In a system which supports strict memory coherency, the value returned by a read instruction is always the same as the value written by the most recent write instruction allowed to complete to the same address. The shared virtual address space is typically partitioned into pages. Each memory mapper views its local memory as a big cache of the shared virtual address space for its associated node. A memory reference may cause a page fault when the page containing the memory location is not in a node's current physical memory. To serve the page fault, the memory mapper retrieves the page from either the swap space (or paging store) in secondary storage or the memory of another node.

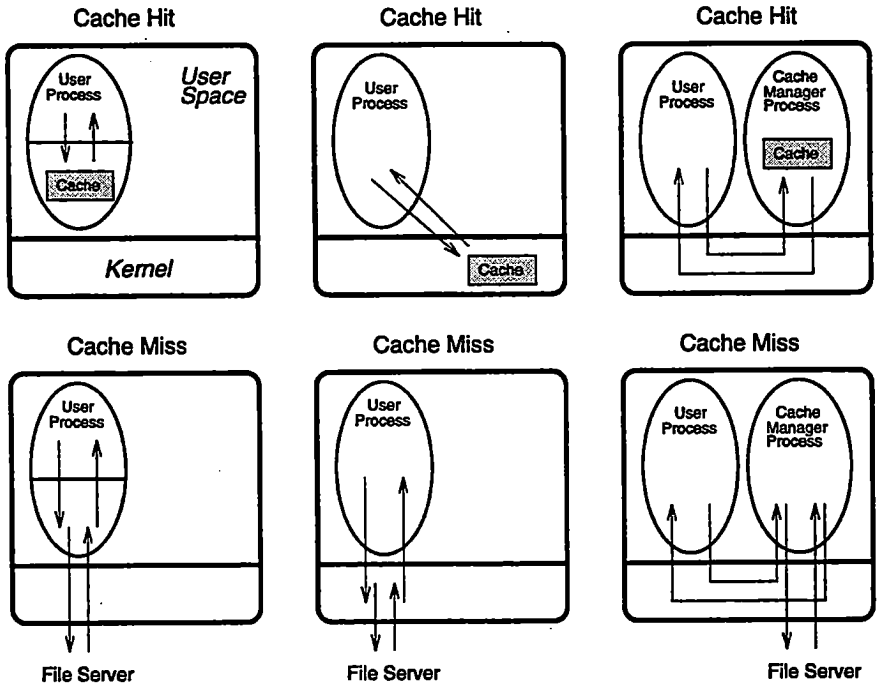
Shared Data Object Model Shared data object systems provide sharing at the level of user data structures rather than at the system level as SVM does. Shared data objects are supported by a language that may have common high-level features such as hierarchical organisation of data. Two approaches to shared data objects are Linda [Carriero89] and Orca [Bal93]. Linda provides a flat logically shared address space within which data structures are built out of primitive objects, so that a structure may be composed of many elements that are distributed by the compiler and run time system. Orca is a high-level

language whose run-time support manages the distribution of objects, providing structured distributed shared memory. The former provides distributed data structures that may be spread across physical memory on many nodes; the latter provides distribution of data structures.

DSM permits and encourages architecture-independent programming, using a memory model similar to that used by conventional programming languages. Moreover, object migration in a SVM is potentially easier than that in a message-based system. Much research in this area has been carried and has been successful to some extent. To date, DSM is still in the stage of research rather than practical use. The major fear is that the overhead involved in maintaining memory coherency over a *local area network* (LAN) is much higher than in a shared memory multiprocessor system. Also scalability remains an uncertainty.

Recent advances in network technology have yielded nearly an order of magnitude improvement in both latency and throughput. This means that the performance gap between loosely-coupled and tightly-coupled multiprocessors is not as dramatic as it once was. As a result, the overhead of maintaining memory coherency in DSM will not be a severe problem in future distributed environments. Also, the overhead can be reduced by integrating the low-level coherency control with the high-level concurrency control as discussed in Chapter 4.

The major factor which affects the scalability of DSM is the number of messages needed to keep DSM coherent. This factor is in turn affected by the access patterns of the applications to the DSM and the coherence protocol used. Most of the existing SVM systems to date, which provide a flat global address space abstraction, are unstructured and use a single memory coherence mechanism. This form of coherency control causes the coherence mismatch between the access patterns and the coherence protocol (see section 3.4 for details) and hence this type of system can not scale well. The shared object model is structured and potentially more flexible. However, almost all of the existing shared data object systems are implemented as run time libraries and are sometimes combined tightly with a specific programming language. They are not as readily provided for shared memory-based programming languages as the SVM. Furthermore, users of both kinds of DSM still see two different interfaces for volatile and non-volatile objects. Systematic and architectural support for coherent persistent shared data objects is desirable.



(a) Caching within User Process

(b) Caching in the Kernel

(c) Caching in a Cache Manager

Figure 2.6: Various Approaches to Caching in Client Memory

2.2.3 Distributed File Service

Caching of data at clients has been proven the architectural feature that contributes most to performance in a modern network or distributed file system [Coulouris94, Satyanarayanan93, Nelson88]. Caching in the clients' main memory provides best performance since it reduces the number of I/O accesses and network accesses. There are usually three approaches to implement the clients' main memory caching [Tanenbaum92] as shown in figure 2.6:

- cache files directly inside each user process' own address space.
- put the caches in the kernel.
- maintain the caches in a separate user-level cache manager process.

In the first approach, the cache is typically managed by the runtime library. When files are accessed, the library keeps the most heavily used ones around, so that when a file is reused, it may already be available. Although this scheme

has an extremely low overhead, there may exist multiple copies of the same file cache in different processes on the same client node. The disadvantage of the second scheme is that a kernel call is needed in all cases. Furthermore, it does not fit the microkernel philosophy. The third way of caching keeps the kernel free of file system code and is easier to program. As discussed before, however, it introduces more context switches. Besides, double paging may occur if the cache manager process runs on an operating system with virtual memory.

Client caching introduces inconsistency into the system. In most such systems, it is the file server which is responsible for maintaining coherence between different client caches of the same file and guaranteeing concurrency control [Nelson88, Satyanarayanan93, Coulouris94]. The file server may become a bottleneck when the system scales up.

2.2.4 Conclusion

The mandatory data copying and unnecessary user/kernel boundary crossings introduced by a traditional file system cause inefficiency. This is exacerbated in a microkernel architecture where most of the user/kernel boundary crossings become context switches. The difference between the existing virtual memory and storage systems' interfaces causes inconvenience for application programmers and impairs the quality and productivity of software constructions. Although persistent programming can solve this problem it is difficult to implement persistent programming languages in a productive and efficient way without operating system support. Sharing data between processes on different network nodes is difficult if only the low-level message passing interface is provided. DSM is attempting to overcome this problem but this is still not ideal. Finally, although caching of data at clients improves the performance in distributed file systems, how to maintain the client cache more effectively is still worth exploring and the file server might become a bottleneck in a large system. These observations lead to the conclusion that some more comprehensive operating system support for distributed programming is desirable.

2.3 Memory-Mapped Object Management

As discussed in the previous section, accessing files in a file system is cumbersome and inconvenient, especially when compared to accessing main memory. Starting with Multics [Organick72], some operating systems have provided a way to map files into the address space of a running process. After being mapped into the address space, data stored in files can be addressed directly by a processor and hence referenced directly by any computation. In a typical high-level language, the opened file appears as an array of bytes. The underlying system is responsible for fetching data from secondary storage as the array is accessed and for writing modified data to permanent store.

The fundamental advantage of direct addressability is that information copying is no longer mandatory. This means, for example, that core images of programs need not be prepared by loading and binding together copies of procedures before execution. Also, partial copies of data files need not be read, via requests to an I/O system, into the user address spaces for subsequent use and then returned, by means of another I/O request, to their original locations. Instead the processor executing a computation can directly address just those required data items in the original version of the file. One of the examples is shown in figure 2.7. In the two-level store approach, when a user attempts to assign values to the persistent object `date_object`, memory has to be allocated for the local variable `d` and values are assigned to it. Then the system call `write` is invoked and the control is changed from the user space to the kernel (or from the user process to the file system agent process in a microkernel architecture). After that, data is copied from the local variable `d` to the system I/O buffer and the control is changed back to the user space. In the case of reading, a system call `read` is invoked and the control is changed from the user space to the kernel (or from the user process to the file system agent process) to copy the data from the system I/O buffer to the local variable `d`. Control is then changed back to the user space where the data is consumed. By contrast, in the memory-mapping approach, if the page is already in the main memory, the persistent object `date_object` can be written or read directly without user/kernel boundary crossing and mandatory data copying. This shows that memory-mapping techniques can be used to improve the system performance since mandatory data copying is avoided and the number of user/kernel crossings or context switches is reduced.

The file store and the backing store for memory management are unified. Once the mapping is set up by the operating system, pages may be read directly from the file and updated pages may be written back to the file. The

```
struct date {
  char weekday[10];
  int day;
  char month[10];
  int  year;
};

void write_routine()
{
  int fd;
  struct date d;

  fd = open("date_object", RDWR);
  strcpy(d.weekday, "Monday");
  d.day = 6;
  strcpy(d.month, "June");
  d.year = 1994;

  write(fd, &d, sizeof(d));
  close(fd);
}

void read_routine()
{
  int fd;
  struct date d;

  fd = open("date_object", RDONLY);
  read(fd, &d, sizeof(d));
  printf("%s %d %s %d\n", d.weekday,
         d.day, d.month, d.year);
  close(fd);
}
```

(a) Two-Level Store System

```
struct date {
  char weekday[10];
  int day;
  char month[10];
  int  year;
};

void write_routine()
{
  struct date *dp;

  dp = mmap("date_object", RDWR);
  strcpy(dp->weekday, "Monday");
  dp->day = 6;
  strcpy(dp->month, "June");
  dp->year = 1994;

  munmap("date_object");
}

void read_routine()
{
  struct date *dp;

  dp = mmap("date_object", RDONLY);

  printf("%s %d %s %d\n", dp->weekday,
         d->day, d->month, d->year);
  munmap("date_object");
}
```

(b) Memory-Mapping System

Figure 2.7: An Example of Accessing a Persistent Object

problems of storage redundancy and the double paging in database systems are eliminated.

Direct addressability also hides the existence of the memory hierarchy and makes the system, rather than the users of the system, responsible for the movement of data between main memory and secondary storage. This kind of access to information promises a very attractive reduction in program complexity for the programmer. In a homogeneous system, data can be stored in their main memory format and hence there is no need to convert from the secondary storage format to main memory format and vice versa. This can simplify the job of implementing a persistent programming language and other

application programs. In a heterogeneous system, unfortunately, there exists the problem of different machine representations of data objects. The first problem goes beyond the byte order problem, since different processors are free to assign any given meaning to any given sequence of bits. A more difficult problem arises from software data types. Modern programming languages allow higher level types to be built on top of hardware types, for instance, in composing record structures with diverse component types. Quite often, the language definition does not specify how these types should be mapped to the hardware types, and the compiler is free to define this mapping. A well known consequence is that the different fields of a structure in the C language may be allocated at different offsets by different compilers, sometimes even among compilers for the same machine architecture. Solving the heterogeneity problem is not easy because it requires that the server has knowledge of the application's data types. This leads to undesirably close links between the application's runtime system and the compiler. Fortunately this problem can be separated into two categories: hardware data types such as integers and software data types like C structures. A general purpose server can solve the former problem and can be extended to cope with the second class of types [Forin88].

Since all data are accessed via the same interface, a single model of protection may be employed. This may be based purely on type security or may be hardware supported in order to provide multi-lingual support [Morrison90]. The user need only be concerned with one mechanism as distinct from the multi-level protection involving processes and files on conventional systems.

Finally, in a distributed environment, file mapping is in effect caching files at clients. The cache can be accessed directly within user processes without the overhead of user/kernel boundary crossings or context switches. With proper concurrency control, the system need not to maintain multiple copies of data as in the first file caching scheme shown in figure 2.6.

Despite the advantages discussed above, persistent programming language and database implementors have repeatedly rejected the idea of using the mapped file facilities offered by operating systems and have chosen to manage buffering and disk storage themselves. The main reasons are as follows [Shekita91, Tanenbaum92]:

- Operating systems typically provide no control over when the data pages of a mapped file are written to disk. The major concern here is that the file contents may be changed as the program runs and special steps must be taken, either by the programmer or by the system, to ensure

that sufficient information is retained for restarting after a failure. The overheads involved might be expensive. On the other hand, data may not be written back into the disk even after the program finishes. It might cause data loss if system failure occurs.

- Operating systems only know the length of a file at page granularity.
- The virtual address space provided by mapped files, usually limited to 32 bits, is too small to represent a large file or database.
- Page tables associated with mapped files can become excessively large [Stonebraker81].

With the right operating system support, however, it is possible to control when the pages of a mapped file are written to disk. Meanwhile, in most of the database applications, *shadow paging* and *write ahead log* approaches [Traiger82, Korth91] are employed in order to be able to recover from system failure. They can still be used in memory-mapped object systems. For the second problem, the programmer can easily know how many bytes have been written to the file in most cases. Thirdly, the emerging 64-bit address space is extremely large. Even if memory is allocated at the rate of one gigabyte per second and never deallocated, the address space will last for 500 years before it is used up [Chase93]. In any case, it is clear that the address space size will continue to grow over time. Large address spaces and relatively slow disks combine with programming language developments to make this approach increasingly attractive [Bacon93]. Also, as the cost of RAM decreases, large page tables will become less of a concern. Meanwhile, *inverted page tables* [Bacon93] and *guarded page tables* [Liedtke94a] techniques can be used to reduce the page table size. Inverted page tables which can be found in the HP Precision Architecture may become more common with the increase in memory sizes. In this scheme, a single main memory page table, which is organised as a hash table, is used by the operating system to record which page of which process occupies each page frame of physical memory. Since physical main memory is likely to be smaller than virtual memory this is potentially economical in both space and searching time compared with holding a page table per process. In the guarded page tables scheme, each page table entry is supplemented by a variable length bit string called *guard*. A page table entry is selected by the highest part of the virtual address upon each transformation step in the same way as with the conventional multi-level page table method. If the *guard* in the entry is a prefix of the remaining virtual address, the translation process either continues with the remaining postfix or terminates with the postfix as the page offset. In a

sparsely occupied huge address space, this scheme reduces the size of memory occupied by page tables and the number of address transformation steps.

Though memory-mapped interfaces have been implemented on several operating systems both in the traditional monolithic systems such as Multics and Sun OS and modern microkernels like Mach [Accetta86], Chorus [Rozier88] and the V system [Cheriton88a], few of them give added functionality to the user as they do not provide the facilities for users to easily implement logical abstractions [Mapp91]. This has been explored in [Mapp91] by providing a typed interface to the virtual memory manager. His work was mainly concerned with a single machine. How to extend this approach to structure virtual memory in order to support distributed computing environments, especially how to maintain coherency between mapped copies of the same object on different nodes and how to guarantee concurrency control, are worth investigating.

2.4 Summary

In this chapter, new developments both in hardware and software were presented, the limitations of the existing memory and storage management approaches were discussed and the memory mapping techniques were examined. This leads to the conclusion that a new approach to virtual memory and storage service, which integrates main memory with secondary storage and local memory with remote memory, to support the emerging distributed computing environment is worth investigating and the memory mapping technique could be used with a carefully designed framework.

Chapter 3

Architecture Framework

As discussed in the previous chapter, a new approach is desirable to integrate virtual memory with secondary storage and local memory with remote memory in distributed computing environments. This chapter proposes an architectural framework to meet such requirements.

3.1 Introduction

The terms *type* and *object* are frequently used in this dissertation. Here **type** means different logical data abstractions managed in different ways, such as executable code, data, stack, file and persistent object types. This provides the flexibility for a number of aspects of virtual memory management and persistent object management, such as the implementation of paging algorithms and object coherence protocols.

An **object** is defined as a logical entity which is an instance of a *type* and can be mapped into a contiguous region of a process virtual address space. When an object is mapped, it can be read or written by simply reading or writing an address location within the process address space corresponding to the offset of the byte in the object. If the object is shared between multiple processes, every modification made by any of the processes is immediately visible to the others on its completion.

The use of the term "object" here does not imply any sophisticated concept used in object-oriented programming languages. It is solely an array of uninterpreted bytes, or more precisely, an array of pages, possibly associated with some backing secondary storage, which can be used to contain and enclose

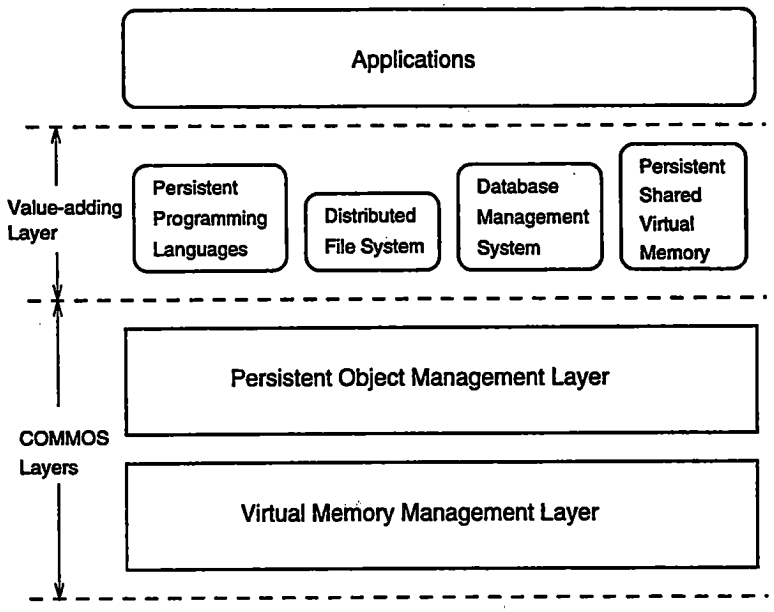


Figure 3.1: The COMMOS Layers

language-level objects. Class code can be bound to it by an object-oriented programming language runtime library as illustrated in Chapter 7. The term *system-level object* is sometimes used in this dissertation to avoid confusion with *language-level object*.

Processors and network connections may fail by crashing. As pointed out by [Bacon93], failure transparency is expensive and hence is not desirable to be supported in the low-level architecture. When a failure occurs, an exception is raised and passed to the client affected. It is up to the clients to react in the face of failures.

3.2 Object Management Layers

Object management in COMMOS is divided into two layers: the basic *virtual memory management layer* and the *persistent object management layer* (see figure 3.1). Above these layers, there may be other value-adding clients, such as persistent programming languages, database management systems, distributed file systems and persistent shared virtual memory.

The virtual memory management (VMM) layer manages the local memory and is further divided into machine-dependent and machine-independent parts.

The machine-dependent part is concerned with managing the memory management unit (MMU) hardware and catching all page faults. The machine-independent part is concerned with managing address maps, satisfying the page faults for zero-filled objects and interfacing the virtual memory management layer with the persistent object management layer.

The persistent object management layer manages the persistent objects and consists of three components: **persistent object managers** (POM), **coherence servers** (CoherSvr) and a **public interface**. The POM is in charge of the data movement between the main memory and the secondary storage. Each object type has only one corresponding POM on each node while a POM can manage one or more object types. The CoherSvr is concerned with the object coherency control and cooperates with the POM to maintain the object coherency in a distributed system. There is only one CoherSvr for each coherence protocol while a CoherSvr can manages one or more coherence protocols. The reason for the separation of the CoherSvr from the POM will be discussed in Section 3.4. Finally an interface is provided for its clients to manipulate objects.

3.3 Object Naming and Protection

A global naming scheme is desirable in order to make the persistent objects visible in any node of the system. A global name is globally unique and location independent, and can be used to refer to an object from anywhere in the distributed system. A naming scheme with fixed-length bit pattern identifiers, such as the one used in the *multi-service storage architecture* (MSSA) [Lo94] is of preference because it is convenient to manage.

In order to protect objects from unauthorised client accesses, the persistent object manager needs to know the access rights of clients to the objects it manages. This information can be stored as an access matrix [Lampson71], in which the rows represent the clients and the columns represent the objects. The access rights that a client holds for an object can be found at the intersection of the corresponding row and column. This matrix, however, is too large and sparse to store. An alternative way to represent the access matrix is to associate each object with a list containing all the clients that may access the object with their access rights. This list is called an **access control list** (ACL). The other way of slicing up the matrix is to associate each client with a list of objects that may be accessed with an indication of which operations

are permitted. This list is called a **capability list** and the individual items on it are called **capabilities**. The ACL method is chosen in this design but capabilities approach could also be used.

3.4 Concurrency and Coherency Control

Allowing objects to be mapped into process virtual address spaces on different network nodes at the same time introduces the object coherency problem. One way to ensure object coherency is to allow multiple readers to read a specific object fragment at the same time but writers must have exclusive access [Bacon93, Tanenbaum92]. This can be achieved by using locking.

If the system is concurrency transparent, clients need not be aware of the existence of other clients. When a process attempts to access a memory location, the system automatically acquires a lock for the corresponding memory unit on behalf of the process. However, if language compilers are not to be changed, at least some sort of preprocessors are needed to detect the memory accesses and to acquire locks on the clients' behalf. More importantly, the semantics of transparent locking is not well defined. By contrast, in an explicit locking scheme, clients are required to acquire locks when synchronisation is desired. By doing so, clients obtained better control over concurrent accesses of shared data. The explicit locking approach is adopted in this architecture framework.

Many protocols have been developed to tackle the system wide cache coherency problem both in distributed shared memory systems and in distributed file systems with client caching. Nevertheless, it is extremely difficult to have a single protocol which can perform well for different types of application. Two kinds of commonly used protocols, namely *write-invalidate* and *write-update*, are considered here. In the following discussion in this section, the page is assumed as the unit of coherency control. The list of nodes which have a valid cached copy of an object page for reading is called the *copy set* of the page. The node which has the most recent write access to an object page, and hence has the up-to-date page contents, is called the *owner* of the page. In write-invalidate protocols, when a process attempts to write to an object page it has to instruct the copy set of the page to invalidate their caches before it can go ahead. Any subsequent read accesses from nodes other than the owner will cause a copy of the page to be shipped from the owner. In the write-update protocols, each modification to an object page is written through to the copy set.

In the case that the copy set is relatively stable and the page is accessed many times by each node, the write-update protocols will perform better. For example, if the size of the copy set is n, for each writing operation, n RPC calls are required in the write-update protocols while $(2 \times n)$ RPC calls are needed in the write-invalidate protocols if the page is accessed again by all the nodes in the copy set after the updating. On the other hand, if the copy set changes dynamically, the write-invalidate protocols are expected to perform better since there is no need to ship data to those nodes which are in the copy set but which do not use the data in the near future.

The separation of the CoherSvr from the POM is proposed in this dissertation to overcome the problem of coherence mismatch. A CoherSvr can implement one or more coherence protocols while there is only one CoherSvr for each coherence protocol on each node. Working with the typed object principle, this coherence mechanism enables the clients to apply different protocols to different types of object or even to different individual objects so that they can choose the most suitable protocols for their applications. Dynamically adopting different coherence protocols to meet the change of the runtime circumstance is also possible. The design and implementation of this coherence mechanism are discussed in detail in Chapter 4 and Chapter 5.

3.5 Persistent Object Management

There is a data structure to represent each object which is currently in use. These data structures form an **object table** which resides in the kernel but may be mapped into the address spaces of POMs and CoherSvrs. The interaction between a user thread and the POM is carried out via a simple interface and a set of events.

The interface for the POMs supports a number of functions, such as mapping the free page list into the POM's address space, registering with the system about managing an object type, getting the relevant characteristics of an object, and getting relevant data from the network storage server to serve page faults and then returning the results.

A process invokes the POM by triggering an event and then suspends itself until the POM replies. POMs handle various events which occur as objects are accessed (e.g. a page fault) or as operations are invoked by the client via the public interface. These include flushing modifications to the secondary storage, writing out changes when persistent objects are unmapped, swapping

out pages when they are selected by the paging algorithm to be removed from the main memory and reclaiming resources from a persistent object after it has fallen out of use. The details of the VMM interface and events handled by POMs are shown in Appendix B.

3.6 Public Interface

This system is assumed to be used mainly by class library builders or high-level abstraction implementors. The public interface provides the facilities to map and unmap objects in an address space, to create and destroy objects, to flush modifications back to the secondary storage server and to grow or shrink objects. Multiple-reader/single-writer synchronisation is also supported. Interested readers are referred to Appendix A for the details of the public interface.

In order to support persistent programming and database applications, it is important that upon successful return of a request to write modifications to the backing store, the modifications have indeed been written. So the operations of flushing and unmapping an object should be synchronous. This means that the caller is blocked until the POM has completed the operation and indicates whether the operation is successful. The other important issue is that the clients should be allowed to choose whether to be blocked when acquiring a lock which can not be granted immediately.

3.7 COMMOS Architecture

The coherent object management system consists of well-known global **coherence managers** (CoherMgr) and per node based **coherence servers** (CoherSvr). For each coherence protocol, there is one corresponding CoherMgr and multiple CoherSvrs, one per node. One set of CoherMgr and CoherSvrs may manage one or more protocols. When a persistent object is mapped onto a network node for the first time, the CoherMgr is made the default owner of all the pages of the object. The CoherMgr, CoherSvrs, POMs and the network storage server work together to manage objects and their coherency in the system. This is illustrated in figure 3.2. A number of processes on each network node have object x mapped into their address spaces, possibly at different virtual addresses, and are accessing it. When page faults or other events

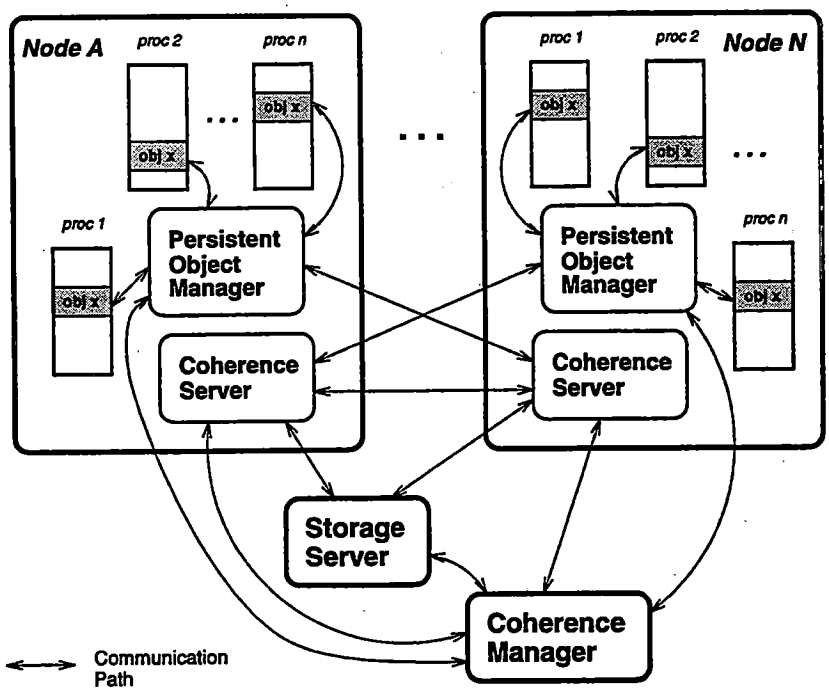


Figure 3.2: COMMOS Architecture

occur for object x, the POM, which manages all the objects of the persistent object type of x, is invoked. The POM communicates with the CoherMgr or the CoherSvr on another network node and the network storage server to move different parts of the object to and from the secondary storage and maintain the guarantee of object coherency if required.

3.8 Previous Work: Memory-Mapped Object Management on Wanda

This section presents a memory-mapped object management system in a locally developed microkernel which forms the starting point of this work.

Wanda [Dixon91, Bacon93] is an experimental microkernel operating system developed at the University of Cambridge. It is a vehicle for research into kernels for high-performance services. Wanda supports preemptive multi-threaded processes and multiprocessor thread scheduling. Memory management is based on paged segments, and page mapping and unmapping is used for efficient inter-process communication and for network I/O. Semaphores

are supported and are used both in the kernel and for user-level inter-thread communication within an address space. Wanda also employs an event mechanism by which processes are notified about various events.

An extension to the kernel virtual memory management supporting memory-mapped objects [Mapp91, Huang92] has been designed and implemented on Wanda Snap2. Objects are typed and managed by external *object managers*, which are in charge of the data movement between the main memory and the secondary storage.

In this system, a Wanda process consists of various objects namely: a *code* object, a *data* object, a *bss* (uninitialised data) object, a *stack* object, an *environ* object, and other objects created or mapped by the process. Information about where and how an object is mapped is contained in a **process map** which is mapped **read-only** into the address space of the process as part of its initialisation. Each entry in the process map, known as a `map_entry`, contains the name of the object, the type of the object, the starting address and the length of the object, the access rights of the user process, the index (called `vir_id`) of the `map_entry` in the process map and the number of threads accessing the object in the same address space. The creation of user processes is managed by a **process server** (ProcSvr) running in user mode.

Some ideas on which the Wanda memory-mapped object management was based have been used in the design of the COMMOS. These include the principle of typed object and the use of external object managers. Much of the source code is used in the COMMOS prototype implementation. However, the COMMOS differs from this previous work in the following key points. The operations to flush the modifications back to the secondary storage and to unmap an object were asynchronous. Although this provides better response time, it is potentially dangerous in persistent programming and database applications since data may be lost after the clients are told that they have been written back. Also, lock operations always block the caller in the previous design. This is not feasible because the high-level applications can not use these operations to construct their own sophisticated mechanisms like two-phase locking. More importantly, the previous design did not tackle the object coherency problem so it was not suitable to be used in a distributed computing environment.

3.9 Summary

A coherent memory-mapped object system architecture has been introduced in this chapter. The object model, which is based on the typed object principle is given and a two-layer structure for the COMMOS is chosen to meet the requirements discussed in the previous chapter. A broad outline of the architecture framework was then presented. Finally, the Wanda kernel and its memory-mapped object management were described.

The following chapters will present the design details of the coherence mechanism and a prototype implementation.

Chapter 4

Design of the Coherence Mechanism

4.1 Introduction

In order to support the memory-mapped object management system in distributed computing environments and to retain the same object sharing semantics as that in a centralised system, mechanisms to maintain the coherence between memory-mapped copies of the same object on different network nodes are desired. As discussed in Section 3.4, no single universal object coherence protocol can satisfy the needs of all kinds of applications. Also the support of typed objects in the memory management system provides a good opportunity to apply different protocols to different types of object. The key issue here is to design a well-defined and flexible interface to a variety of object coherence protocols by which applications can advise the system to apply the most suitable protocol for a specific type of object. Besides, there may be a need to apply different protocols to individual objects as well as dynamically changing the protocol being used according to the system load and the concurrency level being experienced.

This chapter explores the design of the COMMOS coherence mechanism to support multiple coherence protocols. Various major design issues are considered first and the public interface is then given. Third, the coherence protocols are presented. After that, the coherence manager and the coherence server are described. The coherency control using the COMMOS coherence mechanism is illustrated in Section 4.6. Readers who are not particularly interested in the inside details of the coherence mechanism may skip this section.

4.2 Design Considerations

4.2.1 Granularity

The granularity of memory, at which coherency is maintained, is an important design issue. The larger the granularity, the greater the contention will be [Coulouris94]. Memory contention occurs when two or more nodes attempt to access the same memory unit and at least one required access is a write. To reduce memory contention, a small granularity is desirable. In a typical network environment, due to the overhead of the software protocols, the transmission of large packets which may contain thousands of bytes is not much more expensive than the transmission of small ones. Therefore, large granularity is expected to improve the network throughput. In the emerging ATM network environments, network throughput improves while the packet size increases to a certain point [Dharanikota94, Kara94]. Beyond that point (8KB for IP in the environment used in [Dharanikota94]), the throughput drops because segmentation and reassembly takes more time. Meanwhile, a page represents the smallest memory unit on which protection can be enforced by the memory management hardware and the existing page fault schemes can be used. A page-based approach therefore seems natural.

Supporting Variable Length Granularity for Users

It is inappropriate for most distributed applications programmers to be required to be concerned about the details of page-level management. A friendlier interface supporting variable length units for synchronisation is desirable.

One way to provide variable length granularity is to allow fine-grained locking but updates are always directed to the process' own shadow copy. Modifications are merged to the master copy on release of a lock. The *copy-on-write* technique can be used to speed up page shadowing. A software approach always provides higher flexibility but is likely to be more complicated to implement and the performance will be poor due to software overhead.

An alternative approach is to translate variable length granularity in the user interface into the underlying page-based mechanism. This is relatively simple and straightforward and is adopted in this design. The major shortcoming is that memory contention is potentially high for fine-grained object sharing, and deadlock may be another problem (see Section 4.3.3 for details). Some object

clustering strategies to organise fine-grained objects which are unlikely to be accessed simultaneously into the same page may be desirable.

Some researchers have started exploring memory management techniques to support fine grained page size [Liedtke94b] and to accommodate mixed page sizes in virtual memory [Liedtke94b, Khalidi93b]. If these techniques become available in the future, hardware enforcement can be used for variable length granularity.

4.2.2 Remote Interprocess Communication

To build a coherent memory-mapped object system in a distributed environment, communication between processes on different network nodes is inevitable. Since nodes in distributed systems do not share physical memory, communication via shared memory is not applicable although a virtual shared memory abstraction can be created at a higher level. Remote interprocess communication can only be performed by exchanging messages. Different sets of primitives can be used but the basic one is *message passing*. Some other mechanisms, such as RPC and *transactions*, can be built on top of it.

Message passing between remote processes is an extension of interprocess communication for centralised systems. It appears more flexible than the other paradigms since a single message can result in zero, one, or many responses, and the responses need not come directly from the original message destination. Also, as in assembler programming, it is more convenient for programmers to tune their code for better performance. However, programming with message passing has been shown to be both time consuming and error prone [Hamilton84].

An alternative approach is based on the fundamental linguistic concept known as the procedure call. The general term *remote procedure call* means a type-checked mechanism that permits a language-level call on one computer to be automatically turned into a corresponding language-level call on another computer. Most languages are fundamentally procedural, so that using procedural interfaces for remote communication avoids an unnecessary conceptual change [Hamilton84]. RPC has been proven successful in easing the production of software and appears to be making its way slowly into commercial distributed and networked systems.

Transactions were originally developed for database management systems, to aid in maintaining arbitrary application-dependent consistency constraints on

stored data. The transaction mechanism can be built over message passing or RPC. This approach simplifies the construction of reliable systems since it provides uniform support for invoking and synchronising operations on shared data objects, assurance of serialisability of transactions with one another, and atomic behaviour and recovery in the face of network and node failures. The concept of a transaction is a very convenient representation of the integrity of communication and computation in distributed systems because interaction between processes may be a sequence of communications and computations [Goscinski91]. However, it is expensive and not needed in the low-level interprocess communication.

MultiRPC

When objects can be mapped into many user process address spaces on different nodes, *one-to-many* communication is likely to be involved in order to maintain the coherency between different memory-mapped copies of the same object. Using the traditional *one-to-one* RPC to carry out this kind of communication would introduce significant delay. This problem is exacerbated by the fact that an RPC to a dead or unreachable computer must time out before the connection is declared broken and the next computer tries. Each such node would cause a delay of many seconds, rather than the few tens of milliseconds typical of RPC roundtrip times for simple requests. It is felt that the potential delay in a scalable system would be unacceptable if one-to-one RPCs are used sequentially.

A simple broadcast is not advisable. With broadcast, all nodes in the system have to process each broadcast request, slowing down the computation on all computers and hence the performance is poor. Multicast communication is useful since it involves only the nodes in the group. This will get rid of both the problem of delay introduced by using one-to-one RPC and the problem of performance degradation by using broadcast. However, raw multicast is difficult to use and furthermore not all network hardware supports multicast.

A mechanism which retains one-to-one RPC semantics while overlapping the computation and communication overheads at each of the destinations, and which can make use of multicast if the underlying hardware has such a support, is desired. **MultiRPC** [Satyanarayanan90, Satyanarayanan91], which is an extension to RPC2 developed at Carnegie Mellon University and used in the Andrew and Coda file systems, meets such requirements. It enables a client to invoke multiple parallel remote servers with or without hardware

multicast support while retaining the control flow and delivery semantics of RPC (RPC2 and its MultiRPC extension will be referred to as MultiRPC hereafter). This mechanism has been adopted as the remote interprocess communication tool in COMMOS. It has been installed on the mips Ultrix in the University of Cambridge and has been ported onto Wanda.

4.2.3 Writing Modifications to Backing Store

The policy used to write modified pages to the backing store has a critical effect on the system's performance and reliability. The simplest policy is to write data through to the storage server as soon as it is placed in the main memory. The advantage of write-through is its reliability: little information would be lost when the machine crashes. However, this policy requires each write access to wait until the information is written to the disk, which results in poor write performance.

An alternative policy is to delay write-backs: pages are initially written only to the main memory and then written through to the storage server some time later. This policy has two advantages over write-through. First, since writes are to the main memory, write accesses complete much more quickly. Second, data may be deleted before it is written back, in which case it need not be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data will be lost whenever the machine crashes.

In persistent programming, one of the important issues is reliability. It is unacceptable that data may be lost after users think the changes have been made persistent by flushing or unmapping the object. Therefore, the policy of writing modified pages back to the storage server whenever an object is flushed or unmapped appears to be a good compromise. Besides, during the course of coherency maintenance, modified pages may have to be transferred from one node to the other. Since keeping track of the data modification between nodes would increase both the possibility of data loss and the complexity of implementation, modified pages are written back to the storage server before they are transferred to another node for further changes.

4.3 Public Interface and Related Issues

This section describes the public interface to the COMMOS coherence mechanism and discusses the related issues. Objects are identified by globally unique names as stated in Section 3.3. When an object is mapped at a network node, it is assigned a locally unique integer as the local name, which is referred to as the *object number* or `ObjNum`. After an object is mapped into a user address space, the index of the process map, which is called `vir_id`, is used as the object name local to that address space.

4.3.1 Public Interface

The following calls are provided to the clients of the COMMOS to assure the multiple-reader/single-writer constraint:

- `AcquireLock(vir_id, offset, length, rw, mode) -> {-1, 0}`

  Acquires a lock for an object fragment specified by parameters: `vir_id`, `offset` and `length`. Whether the lock required is `read` or `write` is indicated by the parameter `rw`. The parameter `mode` decides whether the caller is blocked if the lock cannot be granted immediately, i.e., there is already a write operation in progress or when a `write` lock is requested there are read operations in progress. This is for supporting higher level concurrency controls such as two-phase locking and transactions. The call returns 0 if the lock is granted or -1 if it is denied.

- `ReleaseLock(vir_id, offset, length, rw)`

  Releases a lock for an object fragment.

4.3.2 Page-Based Lock

As discussed in Section 4.2.1, the page is adopted as the the basic unit of object coherency control. A page-based locking mechanism is therefore provided for server and system use. The variable length object fragment locks are translated into page-based locks in the implementation.

- `AcquirePageLock(ObjNum, BlockNum, rw, mode) -> {-1, 0}`

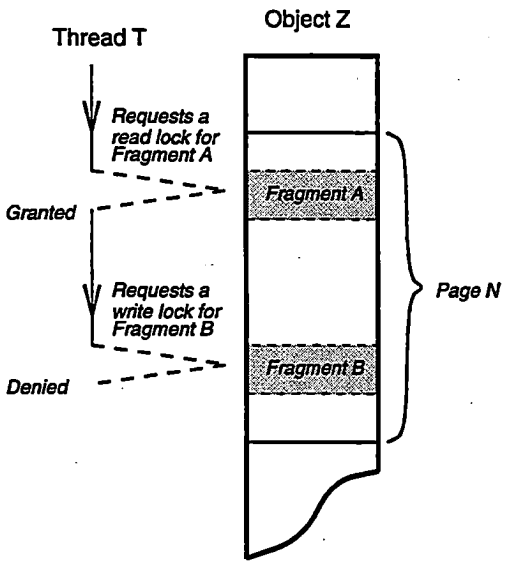


Figure 4.1: Self-Deadlock When Fine-Grained Locks Are Requested

Acquires a lock for an object page. `BlockNum` is the index of the page within the object. The parameter `rw` indicates a `read` or `write` lock is requested and `mode` decides whether the caller is blocked if the lock cannot be granted immediately. The call returns 0 if the lock is granted or -1 if it is denied.

- `ReleasePageLock(obj_num, block_num, rw)`

  Releases a page-based lock.

4.3.3 Deadlock Prevention

When a lock for an object fragment which consists of multiple pages is requested, the corresponding page locks are requested in incremental order so that no deadlock can occur. However, if locks are requested for small object fragments which are possibly located on the same page, deadlock may occur. There exist self-deadlock and inter-thread deadlock.

Self-Deadlock

The self-deadlock of a thread is illustrated in figure 4.1. After successfully obtaining a read lock for a small fragment A of object Z, a thread requests

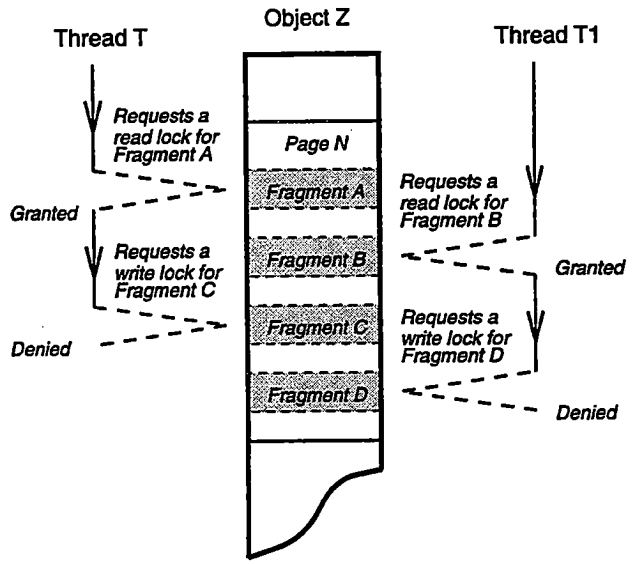


Figure 4.2: Inter-Thread Deadlock When Fine-Grained Locks Are Requested

a write lock for another fragment B which is in the same page as A. Since the underlying locking mechanism actually locks the whole page, this thread would never get the write lock for B before it releases the read lock for A. This also may occur with read and write the other way round or when both requested locks are write locks.

Self-deadlock can be prevented by keeping a list of threads which hold a lock in the page lock mechanism and checking before a lock request is denied. When a write lock for a page is requested, if the caller thread holds and is the only one who holds a read lock of the page, the write lock is granted. If a read lock for a page is requested and the calling thread holds the write lock for the page, the read lock is granted.

Inter-Thread Deadlock

Threads which request fine-grained locks on the same page can also result in a deadlock. Figure 4.2 shows an example. After successfully obtaining read locks for fragments A and B, thread T and T1 request write locks for fragments C and D. Because there are two holders of the read lock for page N, their requests can not be satisfied unless one of them releases the read lock. A deadlock occurs.

Suppose that a transaction mechanism based on two-phase locking is used at

the higher level, a simple way to prevent inter-thread deadlock is to use the unblocked mode of `AcquireLock` and spin-wait for certain amount of time. If the lock is still not granted abort the transaction.

4.4 Coherence Protocols

In this section, two groups of coherence protocols, *write-invalidate* and *write-update*, are described. They are derived from well known protocols and the modifications made in this work do not change their correctness.

A CoherSvr is the **owner** of an object page if one of the processes on that node has performed or is performing the most recent write operation to that page and hence owns the up-to-date contents of the page. This CoherSvr is said to hold the **ownership** of the page. The CoherMgr is set as the default owner of all objects' pages to which a protocol maintained by the CoherMgr is applied. This means that whenever a node has no idea about who the owner of an object page is, requests are made or forwarded to the CoherMgr.

All CoherSvrs which have an object page cached for read but are not the owner of the page form the **copy set** of that page.

4.4.1 Write-Invalidate Protocols

The write-invalidate protocols, namely the *centralised-control* protocol and the *distributed-control* protocol, are derived from those used in the IVY shared memory system [Li86] but differ in several ways. First, coherency control in COMMOS is integrated with high-level concurrency control (see Section 4.4.3 for more details) and the coherency control is only involved at synchronisation time. By contrast, the coherency control in IVY is not integrated with synchronisation and more messages are needed since the coherency control mechanism is invoked for each write instruction. Second, the COMMOS supports persistent data object sharing and the clients can choose different protocols for different applications while the IVY provides a flat shared virtual address space without persistence and its clients lack the flexibility needed to apply different protocols to different applications. Third, the COMMOS employs parallel MultiRPC for one-to-many communications while IVY uses only simple RPC.

Centralised-Control Protocol

In the centralised-control protocol, the *owner* and *copy set* information is maintained by the CoherMgr. When a page fault occurs, the POM on the faulting node makes a request to the CoherMgr. The owner of the page sends a copy of the page data to the faulting nodes. As long as a read copy exists, the page is not writable without an invalidation operation, which causes invalidation messages to be sent from the CoherMgr to all CoherSvrs in the copy set.

Because RPC is adopted as the remote interprocess communication tool, at most two RPCs are needed by the POM to serve a read fault: one from the POM to the CoherMgr and the other nested in the first one from the CoherMgr to the owner's CoherSvr. It is possible to let the CoherSvr on the owner of an object page send the data directly to the faulting node. However, this would require the faulting node to send a confirmation message to the CoherMgr after it receives the data [Li86]. This means that two additional RPCs are needed, thus increasing the network traffic.

Since the CoherMgr plays the role of helping all nodes locate and get object pages to serve all page faults occurring in the system, it is a potential traffic bottleneck. This will be worse as the number of the nodes in the system becomes large and there are many page faults.

Distributed-Control Protocol

In the distributed-control protocol, the *copy set* information is maintained by the owner of the page and the *owner* information recorded by all the CoherSvrs in the copy set. The owner information may not be correct but it has been proven in [Li86] that the true owner can always be reached eventually by forwarding requests. The owner information is therefore called *probable owner*. Instead of always making a request to the CoherMgr, when a page fault occurs, the POM on the faulting node sends a request directly to the probable owner of the page. The true owner sends a copy to nodes requesting the page. As long as a read copy exists, the page is not writable without an invalidation operation, which causes invalidation messages to be sent from the owner to all CoherSvrs in the copy set. The CoherMgr is included in the copy set if it is not the owner of the page and invalidation of an object page on the CoherMgr means updating the probable owner information.

The probable owner information is updated whenever an invalidation request is received, a page fault is served or a write/write-access fault request is

forwarded.

With the distributed-control protocol, a faulting node may need as few as one RPC call to serve a read fault. More importantly, it distributes the load of the CoherMgr to all the machines involved in the sharing.

4.4.2 Write-Update Protocol

In the write-update protocol, the *copy set* information is maintained by the owner of the page and the *owner* information is recorded by all CoherSvrs in the copy set.

When a page fault occurs, the POM on the faulting node makes a request to the CoherMgr. The CoherMgr forwards the request to the owner of the page. If it is a read fault, the owner includes the faulting node into the copy set and sends the faulting node a copy of the page data. If it is a write fault, the owner transfers the ownership and sends a copy of the page data with the copy set information to the faulting node. A node in the copy set which attempts to write to the page has to acquire the ownership from the owner before it can actually carry out the writing. During each write, after the owner updates the page, it sends out a MultiRPC call to the copy set to update all the cached copies. Any requests for a page or the ownership of the page to a node which is no longer the owner will be forwarded to the owner.

For each update, the owner of the page has to multicast the modification. When the CoherMgr or a CoherSvr receives an update request, it updates the local cache and the owner information. If the copy set is relatively stable, it will perform well. Otherwise the shipping of the up-to-date page data may be wasteful.

4.4.3 Integrated Coherency Control

The locking mechanism in COMMOS is integrated with the coherency control mechanism. In all the coherence protocols, when a copy of an object page is requested to satisfy a page fault, the owner CoherSvr has to obtain a read lock for the page before it can send a copy of the page data to the faulting node. In the write-update protocol, in addition, when the owner multicasts an update, the CoherSvr which receives the update has to wait for all the reading threads to complete their current reading sessions before it can actually update the

local cache. This assures the multiple-reader/single-writer constraint in distributed environments. Also in the write-update protocol, releasing a write lock to an object page on the owner node triggers the POM to multicast the up-to-date contents of the page to the copy set.

By integrating the high-level concurrency control with the low-level memory coherency control, the implementation of the coherence mechanism is simplified. The system performance is improved since the coherency control is invoked only at synchronisation time. Also, system wide concurrency control is achieved without a global locking mechanism.

4.5 Coherence Manager and Coherence Server

All three coherence protocols described in the last section are supported by a CoherMgr and a group of CoherSvrs. They work with POMs and the storage server to maintain the object coherency and to manage the object data.

The CoherMgr is a well-known server which acts as a coordinator for CoherSvrs. It provides POMs with facilities to open and close an object, to write modifications back to the storage server and to obtain page data or write permission. It also provides CoherSvrs facilities to update the owner information on the CoherMgr and to forward POMs' requests for page data or write permission to the CoherMgr. An **opened object table** is maintained to keep track of the activated objects managed by the CoherMgr. Interaction with the CoherMgr is via an RPC interface described in Appendix D.

The CoherSvrs are per node based servers, which work with the global CoherMgr, the POMs, and other CoherSvrs in the system to satisfy page faults and to maintain the object coherency. It provides POMs with facilities to obtain page data or write permission. It also provides the CoherMgr and other CoherSvrs with facilities to forward POMs' requests, to invalidate or update object pages. The calls to the CoherSvrs are again an RPC interface. It is detailed in Appendix E.

As stated in the last chapter, the COMMOS architecture does not support failure transparency. The error code `FAIL` will be returned and passed to the request POM whenever the RPC system detects that the remote server is not reachable. On receipt of this error code, the POM may raise an exception to the client program and it is the client's responsibility to handle the exception.

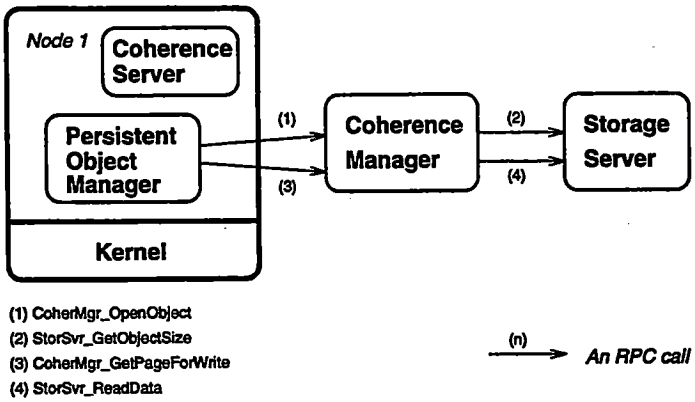


Figure 4.3: Opening An Object and the First Access (All Protocols)

4.6 Illustration of Coherency Control

In this section, some examples are given to illustrate how object coherency is maintained in COMMOS. The names of RPC calls are self-explanatory and detailed descriptions can be found in Appendix D and Appendix E.

4.6.1 Mapping An Object and First Access

When an application program wants to access a persistent object, `MapObject` is called. If it is the first access to the object from that node, the local POM sends a `CoherMgr_OpenObject` request to the CoherMgr which asks the storage server for the object size and sets up an entry in the opened object table before returning the object size for the request POM to set up the object page table. The access permission for all pages of the object is set to be read-only. When the program performs the first write operation, a write fault occurs and the POM makes a `CoherMgr_GetPageForWrite` request to the CoherMgr which loads the page from the storage server, records the requesting node as the owner and grants write permission and a copy of the page to the node. See figure 4.3.

4.6.2 Object Coherency

To demonstrate how the POMs, CoherSvrs and CoherMgr work together to maintain object coherency, after the object has been mapped and the first write access has completed on the first node (node 1), suppose a second node

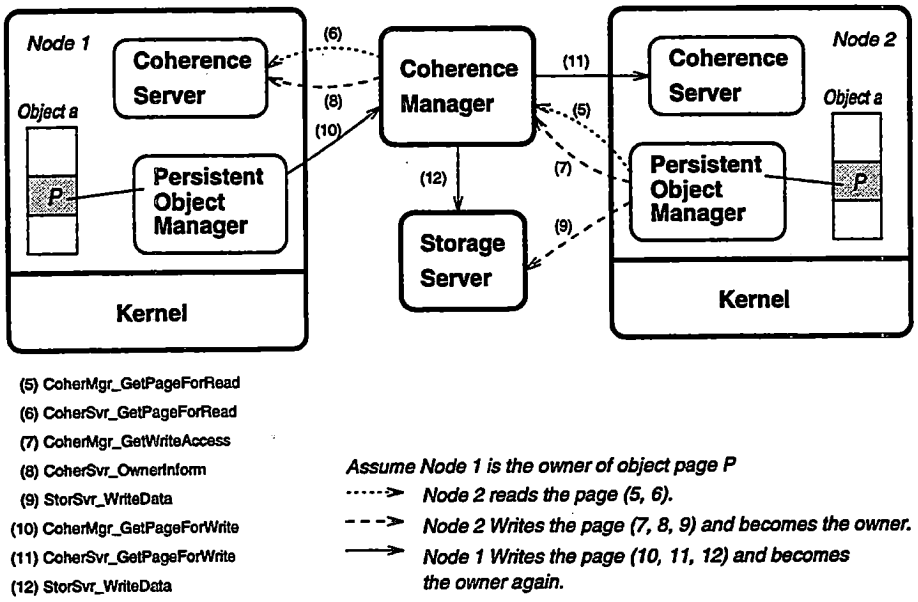


Figure 4.4: Illustration of the Centralised-Control Protocol

(node 2) reads from and then writes to the same object page before the first node (node 1) wants to write to it again. This procedure is examined separately for different coherence protocols.

Write-Invalidate Protocols

Centralised-Control Protocol As illustrated in figure 4.4, when node 2 wants to read the object page which has been fetched by node 1 for write, a read fault occurs and the local POM makes a `CoherMgr_GetPageForRead` request (*RPC 5*) to the CoherMgr. The CoherMgr looks up the owner from the opened object table and finds node 1. It calls `CoherSvr_GetPageForRead` on node 1 (*RPC 6*) to get a copy of the up-to-date page data, then includes node 2 into the copy set and returns the data to node 2 to satisfy the read fault.

Now, if node 2 wants to write to the same object page, a write access fault occurs. The local POM makes a `CoherMgr_GetWriteAccess` call (*RPC 7*) to the CoherMgr, which instructs all CoherSvrs on the owner and in the copy set except the faulting node (node 2) to invalidate their caches of the page (here only the owner node 1 is informed) (*RPC 8*), then clears the copy set information on the CoherMgr, records node 2 as the owner of the page and grants write permission to node 2. On the return of `CoherMgr_GetWriteAccess`, the POM of node 2 writes the page to backing store (*RPC 9*), sets the ownership

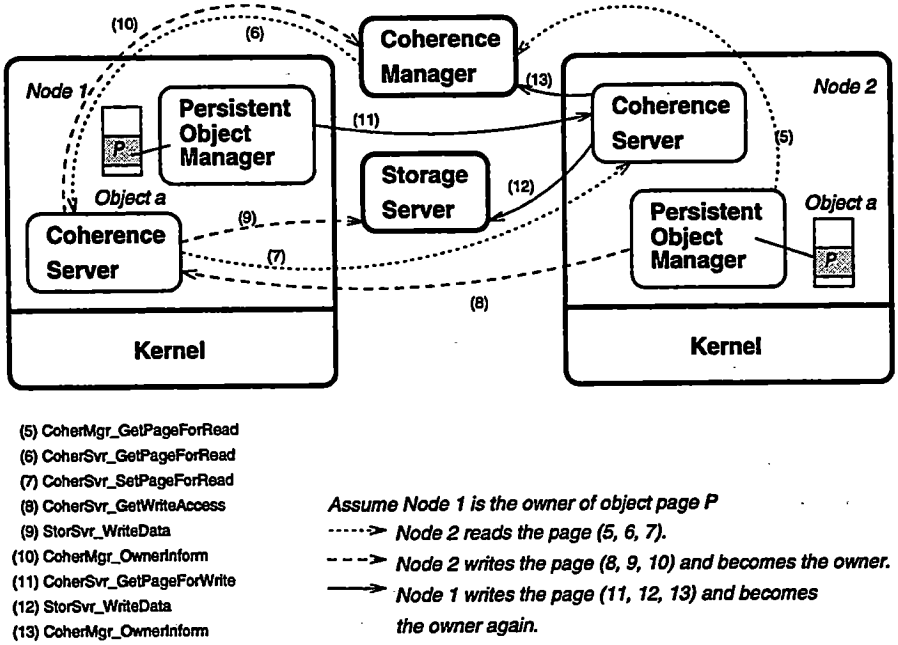


Figure 4.5: Illustration of the Distributed-Control Protocol

indicator of the page, and allows write access to the page.

At this point, if node 1 wants to write something to the same object page, a write fault occurs. The local POM calls `CoherMgr_GetPageForWrite` on the CoherMgr (*RPC 10*), which makes a `CoherSvr_GetPageForWrite` request (*RPC 11*) to the owner (node 2) to get write permission and the up-to-date page data and to invalidate the cache in the owner, then writes the data back to the storage server (*RPC 12*), invalidates the copy set (here no cache needs to be invalidated), clears the copy set information and records node 1 as the owner, then grants the write permission and the up-to-date page data to node 1. After return from `CoherMgr_GetPageForWrite`, the POM on node 1 sets the ownership indicator of the page and allows write access to the page.

Distributed-Control Protocol With the distributed-control protocol, the processing is shown in figure 4.5. A read fault occurs when node 2 reads the object page which has been loaded to node 1 for write access. Since node 2 has no idea who the owner of the page is, the local POM makes a `CoherMgr_GetPageForRead` request (*RPC 5*) to the CoherMgr. The CoherMgr looks up the opened object table for the owner and finds node 1. The request is forwarded by calling `CoherSvr_GetPageForRead` on node 1 (*RPC 6*), which includes node 2 in the copy set and sends data to node 2 by calling

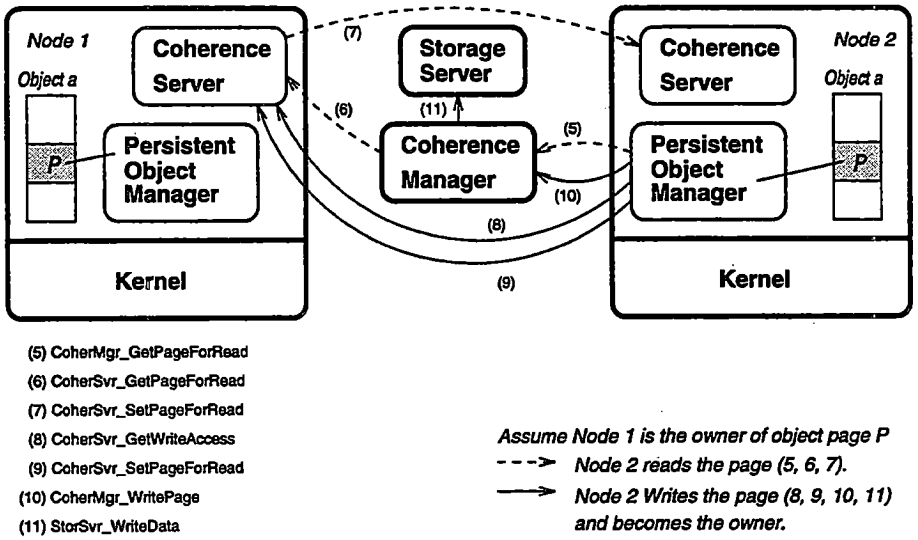


Figure 4.6: Illustration of the Write-Update Protocol

`CoherSvr_SetPageForRead` on node 2 (*RPC 7*).

If node 2 then attempts to write to the same page, a write access fault occurs and the POM makes a `CoherSvr_GetWriteAccess` call (*RPC 8*) to CoherSvr of the owner (node 1) which writes the page to the storage server (*RPC 9*), invalidates the local cache and the copy set except the faulting node (node 2) (only CoherMgr is instructed in this example) (*RPC 10*), clears the copy set information and the ownership indicator, then grants write permission to node 2. Upon return of `CoherSvr_GetWriteAccess`, the POM on node 2 satisfies the fault by setting the ownership indicator of the page and allowing write access to the page.

Suppose that node 1 writes to the same object page now, the local POM calls `CoherSvr_GetPageForWrite` on the CoherSvr of the owner (node 2) (*RPC 11*) which writes the page to the storage server (*RPC 12*), invalidates the local cache and the copy set (CoherMgr in this example) (*RPC 13*), clears the copy set information and the ownership indicator, then grants write permission and a copy of the page data to node 1. The POM on Node 1 then sets the ownership indicator of the page and allows write access to the page.

Write-Update Protocol

Assume the write-update protocol is used, the processing is illustrated in figure 4.6. A read fault occurs when node 2 reads the object page which has

been cached on node 1 for write. Since node 2 does not know who the owner of the page is, a `CoherMgr_GetPageForRead` request (*RPC 5*) is made by the local POM to the CoherMgr. The CoherMgr forwards the request to the owner node 1 (*RPC 6*), which includes the node 2 into the copy set and sends node 2 a copy of the page data via a `CoherSvr_SetPageForRead` call (*RPC 7*).

Suppose node 2 wants to write to the same page, a write access fault occurs and the POM makes a `CoherSvr_GetWriteAccess` call (*RPC 8*) to the owner node 1. Node 1 records node 2 as the owner of the page, sends node 2 the copy set information, and resets the ownership indicator and copy set information. After getting the copy set information, node 2 excludes itself from the copy set and includes node 1 into the copy set. After each writing session, node 2 releases the write lock for the page which triggers an event to invoke the POM to write the up-to-date copy of the page to the copy set (*RPC 9* and *RPC 10*, they are combined as a MultiRPC). For each update, the CoherMgr writes the up-to-date contents of the page to the storage server (*RPC 11*).

If node 1 attempts to write to the same object page now, the procedure is similar to that described for node 2 in the last paragraph.

4.6.3 Unmapping An Object

Finally, if the object is no longer required by an application program, `UnmapObject` is called. If it is the only thread on the node using the object, the local POM makes a `CoherMgr_CloseObject` to the CoherMgr which decreases the reference count and reclaims the ownership tokens held by the POM. When the reference count becomes 0 the CoherMgr destroys the entry in the opened object table.

4.7 Summary

In this chapter, the design issues of a coherence mechanism were discussed. Then the public interface and coherence protocols were presented. After that, the design of the CoherMgr and CoherSvr were described. The interactions between the CoherMgr, CoherSvrs and POMs to assure object coherency were also illustrated. This design shows that it is possible to provide a well-defined interface to a variety of object coherence protocols. In the next chapter, a prototype implementation will be described.

Chapter 5

Implementation

The architecture framework of COMMOS was outlined in Chapter 3 and the design of the coherence mechanism for COMMOS was presented in Chapter 4. A prototype implementation of COMMOS is detailed in this chapter. It is implemented in C and the code is organised in modular style so that data is placed in the same module as the routines manipulating it wherever possible and data and routines are kept local (static) to hide them from other modules whenever possible. This clean programming approach makes the code easy to understand, maintain and extend.

5.1 System Environment

The prototype COMMOS is implemented on Wanda, a locally developed micro-kernel operating system described in Section 3.8. The implementation was firstly done on MVME136 (MC68020) machines running Wanda 1.2 and then was ported to Wanda Snap3. There were three MVME136 machines when the implementation started but two of them malfunctioned and the support for the third one was finally withdrawn. The prototype system was hence ported to the MVME147 (MC68030) architecture.

MultiRPC has been ported onto Wanda and is running over the UDP/IP communication protocols. In Wanda 1.2, UDP/IP was implemented entirely as a run-time library [Gould91, Roe92]. Only one process on each node can use this protocol suite. It was then implemented on Wanda Snap3 using a user space process called the **internet server** [Richardson93] to overcome this problem.

The CoherMgr is implemented on a DEC 3100 workstation running Ultrix 4.3

and a **storage server** (StorSvr) emulator, which uses NFS as its backing store, is implemented to provide the persistent storage for the prototype system. They communicate with servers on Wanda using MultiRPC over Ethernet.

5.2 Object Management

In the prototype implementation, objects are named using NFS path names. As described in Chapter 3, the POMs and the CoherSvrs interact with users via an object table. The data structure used to represent an object contains a number of fields:

`ObjName:`	the global object name;
`ObjNum:`	the local object name;
`TypeName:`	the name of the object type;
`TypeNum:`	the system designated type number;
`Handler:`	the process Id of the POM responsible for the object;
`PageTable:`	the pointer to the object page table;
`ObjOwner:`	the Id of the process that creates this object;
`ProcList:`	a list of processes into which the object is mapped;
`PageAlgor:`	the paging algorithm to be used on this object;
`CoherProt:`	the coherence protocol to be applied on this object;
`PageBitMap:`	an array containing page specific information, which includes the page state, the page lock list, the coherence information and the list of threads waiting for the page fault to be satisfied;
`Monitor:`	a monitor structure (see [Mapp91] for details) used to synchronise access to the object data structures.

The data structure which describes the coherence information consists of the following components:

`access:`	the current access, i.e., nil, read or write, to this page on this node;
`ownership:`	whether this node is the owner of the page;
`owner:`	who the owner or the probable owner of this page is;
`copyset:`	a list of nodes who are in the copy set of this page.

The page lock is represented by a data structure which is made up of the current lock type of the page, a list of threads holding the lock as well as a list of threads waiting for the lock.

5.3 Persistent Object Manager

The prototype POM supports three coherence choices: no coherence, centralised-control write-invalidate or distributed-control write-invalidate. The write-update protocol is not implemented in the current prototype due to time constraints. More protocols may be added in the future.

The **object table** is mapped into the address spaces of POMs so that it can be used to interact with the user processes. The Wanda event mechanism is used to notify a POM of events occurring on an object which require its attention. The POM does an `Investigate` call to examine the state of the object and get the necessary information to deal with the event. It then returns to user space. After serving the event, it makes a `ReturnResult` call to return the results and unblocks threads waiting on the event.

When a POM is invoked to serve a page fault, it checks the field `CoherProt` first to see what coherence protocol is to be applied. If this field is not nil, the POM makes a call `ExchCoInfo` to get the coherence information of the object page and acts accordingly. After the page data or write permission is obtained, it does a call `ExchCoInfo` again to set the new coherence information, such as the access type, the owner and the copy set of the object page.

5.4 Coherence Manager

The CoherMgr can be implemented on the same node as the StorSvr or on any ordinary node. In the prototype implementation it is implemented as lightweight threads on Ultrix sharing a process address space with the StorSvr. The *light-weight process* (LWP) package provided with RPC2 [Satyanarayanan91] is used to support multiple lightweight threads. The prototype coherence manager consists of the following modules: initialisation, opened object table maintenance, server identifier manipulation, connection table maintenance, and object coherence management.

5.4.1 Server Identifier

Servers in the prototype implementation are represented by *server identifier* (`SvrIdent`) data structures. The basic components of a `SvrIdent` structure are:

`Host:` the host IP address;
`Portal:` the port number of the server;
`Subsys:` the subsystem Id of the server.

Other fields include `SecurityLevel`, `EncryptionType`, `UniqueIdent` and `SessionKey`, which may be used by the RPC system to establish secure communications (see [Satyanarayanan91] for details).

The server identifier manipulation module provides operations to `SvrIdent` structures, such as initialising the server's own identifier, comparing and copying `SvrIdent` structures, including a `SvrIdent` into a set, excluding a `SvrIdent` from a set, converting to and from the network order, and getting the `SvrIdent` from an RPC connection.

5.4.2 Object Management

There is an **opened object table** which is organised as a hash table on the CoherMgr to manage all of the opened persistent objects. The data structure for the opened object table is made up of the following fields:

`Name:` the global name of the object;
`ObjSize:` the size of the object in bytes;
`ObjClickSize:` the size of the object in pages;
`UseCount:` number of nodes having this object mapped;
`CoherProt:` the coherence protocol to be applied to this object;
`PageInfo:` an array of information for all the pages of the object.

The `PageInfo` data structure includes the following components for maintaining object coherency:

`ownership:`	whether the CoherMgr is the owner of the page;
`owner:`	the owner or probable owner of the page;
`copyset:`	the copy set of the page;
`lock:`	a lock for exclusively access to this data structure.

When `CoherMgr_OpenObject` or `CoherMgr_FlushObject` is called, the CoherMgr checks the opened object table to see if there exists an entry for the object. If not, a new entry is created. The `UseCount` is increased by 1 each time that `CoherMgr_OpenObject` is called.

When a page fault occurs in the system and the CoherMgr is invoked, the `PageInfo` of the involved object page is set up and used to maintain the object coherency.

Finally, if `CoherMgr_CloseObject` is invoked, the `UseCount` is decreased by 1 and, if it becomes 0, the corresponding entry is removed from the opened object table.

The opened object table maintenance module provides the following operations: looking up the table, creating a new entry, removing an entry and extending the `PageInfo` for an object.

5.4.3 Communication Management

As discussed before, MultiRPC is used as the communication tool in this system. Before an RPC is made, a connection between the source and the destination nodes has to be established. In order to reuse and locate the existing connections efficiently, another hash table, called **connection table** is employed to record them. The data structure for this table consists of three fields:

`svr_ident:`	the identifier of the destination server;
`conn_id:`	the Id of the RPC connection to the destination server;
`conn_timer:`	the timer information for maintaining the hash table.

When communication to a remote CoherSvr is required, the CoherMgr looks up the connection table, if there is no entry for the destination server, it makes a new connection and creates a new entry. This connection, once established, can be used for other RPCs to the same server.

A timer mechanism is needed to maintain the RPC connections. If a connection has not been used for a certain amount of time, it is closed and the associated entry is removed.

The connection table maintenance module provides operations to look up the table and to make a connection to a remote server and then create an entry for it.

5.4.4 Other Modules

The initialisation module is responsible for initialising the LWP and the RPC runtime system while the coherence management module implements the interface presented in Appendix D.

5.5 Coherence Server

The prototype CoherSvr is implemented as lightweight threads sharing an address space with the prototype POM. However there is no conceptual constraint in the architecture which would prevent the CoherSvr from being implemented as a separate user-level process. In fact, if there exist multiple POMs for different object types which share the services provided by a CoherSvr, the CoherSvr is better implemented as a separate process. The CoherSvr is made up of five modules: initialisation, server identifier manipulation, connection table maintenance, coherence management, and housekeeping.

As described in section 5.2, the CoherSvr interacts with user processes via the object table which is mapped into its address space. Like the POM, the CoherSvr accesses coherence information which resides in the kernel using the `ExchCoInfo` call. The initialisation module initialises the RPC runtime system and creates initial threads for the CoherSvr. The coherence management module implements the interface given in Appendix E.

In order to avoid multi-level nested RPCs, in the distributed-control write-invalidate protocol, when a CoherSvr is serving a remote request and finds that the faulting node is not the calling node, it invokes the housekeeping module. The housekeeping module is implemented as one or more threads, to send data or grant write permission to the faulting node, or forward the request to another node.

The other two modules, namely server identifier manipulation and connection table maintenance, are similar to those in the CoherMgr.

5.6 Storage Server Emulator

In the prototype system, the StorSvr is implemented on a DEC 3100 workstation running Ultrix 4.3. NFS is used as the backing store and persistent objects are stored as files. In the future, the *flat file custode* (FFC) in the locally developed MSSA [Lo94] can be used to provide better performance and support a more sophisticated naming scheme. The StorSvr provides facilities to store and retrieve object pages. It also supports calls to get size of an object or different segments of an executable file. The detailed interface is presented in Appendix F.

5.7 Summary

This chapter described a prototype implementation of the COMMOS architecture. It shows that the design is practicable and feasible. The next chapter will present the performance of this prototype system.

Chapter 6

Performance

6.1 Introduction

As described in the previous chapter, the prototype COMMOS is implemented on MVME147 (MC68030) machines running the Wanda microkernel operating system. There were four MVME147 machines, namely *lamprey, lumpfish, piranha* and *shark*. Each of them contains a main board MC68030 CPU with 4MB of RAM. The CPU speed of *piranha* and *shark* is 25MHz while the speed of the other two is 20MHz. They are connected by a non-dedicated Ethernet. *Piranha* has a 4MB RAM card configured, and each of *lamprey* and *lumpfish* has an 8MB RAM card configured. Unfortunately, *shark* malfunctioned during the course of the measurements. Most of the results were therefore taken on the three remaining machines. This configuration is illustrated in figure 6.1. The CoherMgr and the StorSvr run as lightweight processes sharing one Ultrix 4.3 process on a DEC 3100 workstation. There are three CoherMgr threads and one StorSvr thread running concurrently. The Wanda COMMOS software, which includes the extended Wanda kernel, an internet server, a ProcSvr, a CoherSvr and a POM, run on MVME147 machines. The CoherSvr and the POM share the same Wanda process.

In this chapter the basic performance of the RPC system is presented and the use of MultiRPC is justified. Then the performance of memory-mapping and non memory-mapping approaches on Wanda are compared. Finally the performance of the COMMOS prototype is presented. Due to the limitation of the number of processors available and the time constraints, it is not possible in this work to measure and compare the performance of different coherence protocols in a large scale system. The measurements are taken in the small

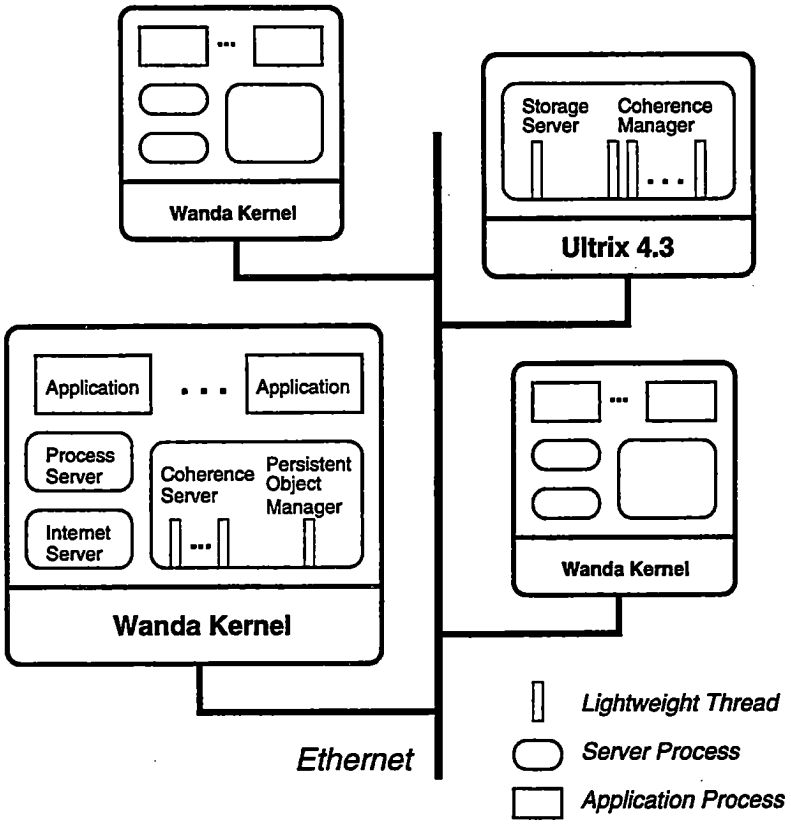


Figure 6.1: Prototype Configuration

system described above and the results are presented to show how this prototype performs. All the results given in this chapter are averages of more than 10,000 iterations of the operation. The worst figures (i.e. the maximum time) are shown wherever the mixing of machines with different CPU speeds is unavoidable. Interested readers are referred to Appendix G for more detailed measurement results.

6.2 Performance of the RPC System

The basic performance of the RPC2 system is presented in this section so that the communication overheads can be known and the use of MultiRPC can be justified. All the results described here are taken from measurements of null RPC calls. A **null RPC** is an RPC without parameters, that executes a null procedure and returns no values. Null RPC performance is important because it measures a fixed overhead [Coulouris94].

RPC Type	*Time (ms)*	
	20MHZ CPU	*25MHZ CPU*
Local Simple RPC	11.0 ms	9.1 ms
Local MultiRPC	11.6 ms	9.6 ms
Remote Simple RPC	16.8 ms	13.1 ms
Remote MultiRPC	17.2 ms	13.7 ms

Table 6.1: Simple RPC vs MultiRPC for One Recipient

Destination Number	*MultiRPC Time (ms)*	*Average Time (ms)*
1	17.2	17.2
2	20.9	10.5
3	24.3	8.1

Table 6.2: Performance of MultiRPC on Wanda

6.2.1 Simple RPC vs MultiRPC

When the source and the destination are located on the same network node, the communication is referred to as *local communication*. Otherwise it is referred to as *remote communication*. To compare the complexity of simple RPC and MultiRPC, local communication and remote communication to a single destination using simple RPC and MultiRPC are measured. The results are shown in table 6.1. It takes more time to communicate with a single destination using MultiRPC than using simple RPC. This is because a MultiRPC has extra complexity compared to a simple RPC. However the difference is small.

6.2.2 MultiRPC Speed-Up

The time taken to make a MultiRPC to up to 3 remote destinations between Wanda machines is given in table 6.2. The average time taken per destination is illustrated in figure 6.2. From the figure, it can be seen that the average time taken per destination has reduced significantly compared to using simple RPCs iteratively.

In order to get a more general idea of how far the speed-up can be taken,

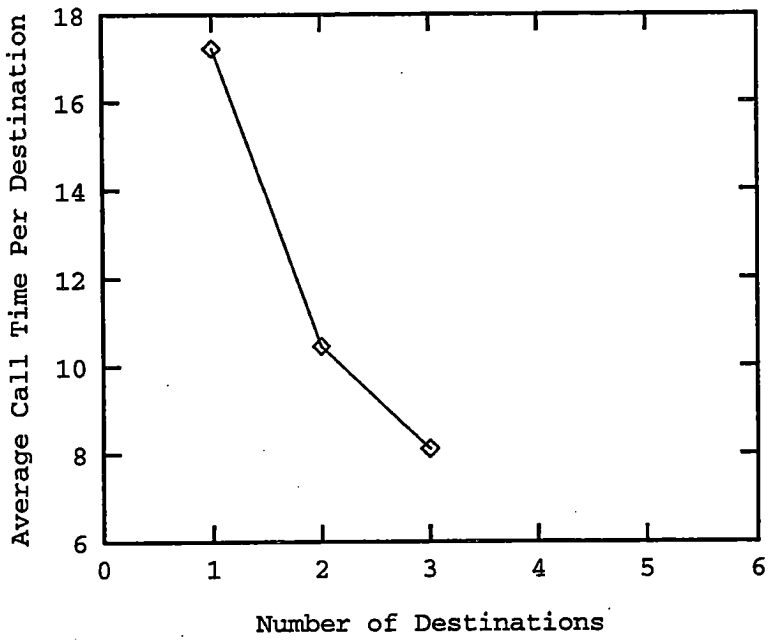


Figure 6.2: Speed-up for One-to-Many Communications on Wanda

Destination Number	*MultiRPC Time (ms)*	*Average Time (ms)*
1	6.4	6.4
2	7.3	3.7
3	8.6	2.9
4	9.4	2.4
5	11.2	2.2
6	13.0	2.2
7	14.3	2.0
8	16.3	2.0
9	18.2	2.0
10	19.8	2.0

Table 6.3: Performance of MultiRPC on Ultrix

similar measurements for up to 10 remote destinations were taken on DEC 3100 workstations running the Ultrix 4.3 operating system. They are shown in table 6.3. The average time taken per destination is given in figure 6.3. It decreases dramatically when the number of destinations is small and then approaches a constant.

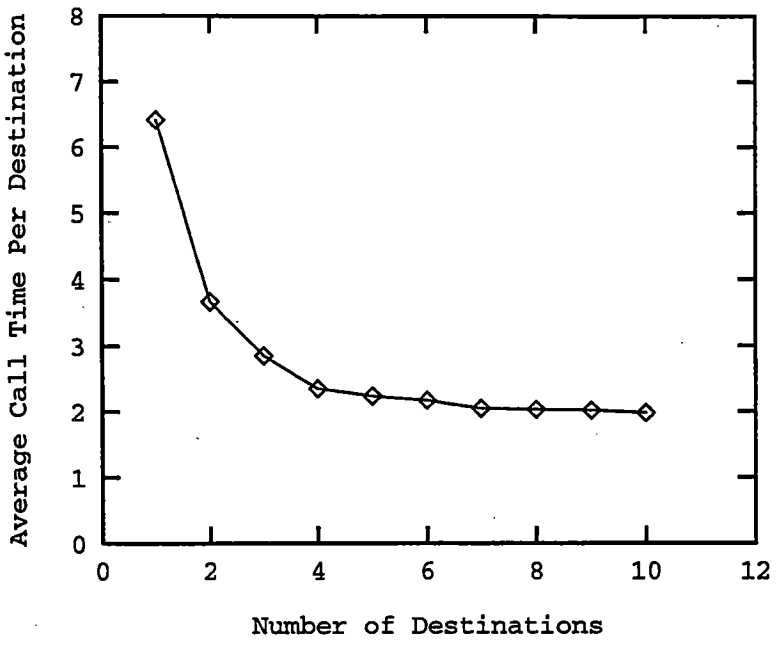


Figure 6.3: Speed-up for One-to-Many Communications on Ultrix

6.2.3 Summary

From the measurement results, it is concluded that the use of MultiRPC is important to speed up the one-to-many communications involved in the object coherency control mechanism.

6.3 Memory-Mapping vs Non Memory-Mapping on Wanda

It is beyond the scope of this work to build a file system without memory-mapping on Wanda to compare the performance with the memory-mapped object system. However, the performance comparison can still be illustrated and reasoned. As a starting point, assume that a two-level store file system is implemented in the model shown in figure 2.2 (b). There is a file system agent on each network node and client caching is supported. In order to access a file from a user process, at least two interprocess communications are required: one for the user to make a request and the other for the agent to return the result. The cost of maintenance of cache coherency in the file system agent is assumed to be the same as that incurred in the POM. The time taken for a

CPU Speed	*Roundtrip IPC Time (ms)*
20MHZ	1.67
25MHZ	1.33

Table 6.4: Performance of A Local Roundtrip IPC

CPU Speed	*Page Fault (ms)*	*Page Invalidation (ms)*	*Sum (ms)*
20MHZ	0.76	0.42	1.18
25MHZ	0.60	0.35	0.95

Table 6.5: Time to Serve A Page Fault and Invalidate A Page

roundtrip IPC on the same machine is shown in table 6.4. Here a roundtrip IPC is when a client thread sends an IPC message to the server process and a thread in the server process sends a reply to the client. There is only one thread in each of the client and server processes in the experiment. On the other hand, the file system agent has to copy the requested data to the IPC buffer and the client has to copy the data from the IPC buffer to the location where the data will be used in practice. These extra overheads for data copying are not counted in the figures shown in the table.

In a memory-mapped object system, after a page fault is satisfied, there is no extra cost for accessing the persistent object. However, servicing page faults and invalidating pages takes time. These are shown in table 6.5.

It can be seen that on Wanda the total time for servicing a page fault and invalidating a page is smaller than the time for a roundtrip IPC even if the overhead for data copying in the non memory-mapping model is not counted. This means that even in the worst situation where a page is fetched for only one access before it is invalidated, the performance is still better than an access in the two-level store model.

This conclusion might not be valid on other platforms because performance varies from implementation to implementation. For example, [Liedtke93] discusses various strategies and shows a significant improvement in IPC performance. Although the performance for page fault processing and page invalidation might also be improved, it is possible that the time taken for a roundtrip IPC in some system is less than that taken for a page fault and a page invali-

Faulting Node	*Time (ms)*
20MHz CPU	31.5
25MHz CPU	30.1

Table 6.6: Fetching a Page from the Storage Server

Owner	*Number of RPCs*	*Time (ms)*
CoherMgr	1	32.9
Wanda	2	40.6

Table 6.7: Fetching a Page for Read in the Centralised-Control Protocol

dation. However, if the working set of a user process can be kept in the main memory for sufficient time that each page is accessed for many times before it is invalidated, performance is still expected to be better than making the same number of accesses to a file system agent using IPC.

6.4 Performance of the Prototype COMMOS

This section presents the results of measuring the time taken by the POM to fetch a page from a remote server to serve a page fault when different coherence protocols are applied.

6.4.1 No Coherency

The time taken to fetch a page (1K bytes) from the StorSvr in order to serve a page fault without any coherency guarantee is shown in table 6.6.

6.4.2 Centralised-Control Protocol

Table 6.7 shows the time taken to fetch a page to serve a read fault in the centralised-control protocol. The first column indicates the owner of the page. Comparing with table 6.6, it can be seen that the overhead for executing the coherence protocol code is not very large. When the owner is another Wanda

Owner	*Copy Set*	*Number of RPCs*	*Time (ms)*
CoherMgr	–	1	32.8
CoherMgr	1	2	56.3
CoherMgr	2	2	57.5
Wanda	–	2	54.6
Wanda	1	3	123.6

Table 6.8: Fetching a Page for Write in the Centralised-Control Protocol

node, a nested RPC, including one from the faulting node to the CoherMgr and the other to the owner, is used to carry out the page fetching. Such a nested RPC is counted as 2 RPCs in this section.

The time taken to fetch a page to serve a write fault in the centralised-control protocol is shown in table 6.8. It can be seen that the time taken to serve a write fault when there are one or two nodes in the copy set is almost the same. This provides further evidence that the use of MultiRPC improves the performance of one-to-many communications significantly. When the owner of the requested page is another Wanda machine, an operation is invoked to write the modified page back to the secondary store. This explains why the time to serve a write fault is longer than the time to serve a read fault in the same situation. Because the CoherMgr runs on a Unix system there is a degree of unpredictability in any timings which involve it. Efforts have been made to reduce the unpredictability: first by ensuring that all the measurements are taken in the middle of the night and that there are no other user processes running on the machine thus reducing unpredictability of process scheduling; second by ensuring that the StorSvr closes the file at the end of every page fetch in order to eliminate the unpredictability of file buffer management. However, there were still system daemon processes running when the measurements were taken. The last row of table 6.8 and the third row of table 6.10 almost certainly represent Unix process rescheduling effects.

6.4.3 Distributed-Control Protocol

Table 6.9 illustrates the time taken to fetch a page to serve a read fault in the distributed-control protocol. When the CoherMgr is named as a relay node, it means that the faulting node has no idea who the owner is and makes its request to the default owner — CoherMgr.

Owner	*Relay Node*	*Number of RPCs*	*Time (ms)*
CoherMgr	–	1	32.4
Wanda	CoherMgr	3	55.3
Wanda	–	1	31.8

Table 6.9: Fetching a Page for Read in the Distributed-Control Protocol

Owner	*Relay Node*	*Copy Set*	*Number of RPCs*	*Time (ms)*
CoherMgr	–	–	1	32.6
Wanda	CoherMgr	–	4	99.0
Wanda	CoherMgr	1	4	165.7
Wanda	–	–	3	85.4
Wanda	–	1	3	99.9

Table 6.10: Fetching a Page for Write in the Distributed-Control Protocol

The time taken to fetch a page to serve a write fault in the distributed-control protocol is shown in table 6.10.

6.5 Summary

In this chapter, the performance of the RPC system was firstly reported and the use of MultiRPC was shown to be an important feature for improving the one-to-many communications for object coherency control. The performance of the memory-mapping and non memory-mapping approaches on Wanda were measured and compared. The results showed that access to persistent data can be improved by using the memory-mapping technique. Finally, the basic performance of the COMMOS prototype system was also reported. The results showed that the system is not fast. There are several reasons for this. First, the internet server, the POM and the CoherSvr all run in user space. This means more context switches are involved than in monolithic kernel systems where all the system services run in the kernel. Second, the general RPC mechanism is used to serve page faults and to maintain object coherency. The marshalling and unmarshalling of arguments for remote invocation are time consuming. Third, while the architecture is aiming for a future high speed and wide address space environment, the prototype is implemented on slow

machines connected by a slow network. It should also be noticed that the prototype has not been specially tuned for optimal performance. Since there are few performance reports available for similar systems and the implementation environments are different from system to system, it is difficult to compare the performance to other systems.

Chapter 7

Supporting Distributed Persistent Programming

7.1 Introduction

Research into persistent object systems and persistent programming languages, which aim to overcome the disadvantages of two-level store systems, is active. Most persistent object systems to date, however, are implemented above a conventional operating system on a single workstation. Some of them, such as CONCERT [Blott94] and Flask [Munro94], use the Unix `mmap` interface to manage memory-mapped paged objects. One recent move is to distribute these systems. For example, Distributed Galileo [Mainetto94] consists of a single object store shared by clients running on different network nodes; Thor [Liskov93] is made up of a set of object stores in a distributed system; and BMX [Ferreira94] is a distributed persistent object system implemented above Unix using the memory-mapping technique. Unlike all of these systems, COMMOS provides comprehensive support for object persistency and distribution from the operating system level. One of its prospective applications is distributed persistent programming.

PS-algol [Atkinson83] is one of the earliest persistent programming languages. It is implemented as a number of functional extensions to S-algol. Persistent data are managed as databases. Each database is associated with an index table which contains a set of name-value pairs and is the root from which preserved data are identified by transitive closure of reachability. Persistent data are managed in a heap in the main memory and the persistent object management system is concerned with the movement of data between the

main memory heap and the backing store. Data may be created on the heap during a transaction or it may migrate there as a copy of some persistent data. When a transaction is committed, all the data on the heap that is reachable from the persistent objects used during the transaction are transferred back to the disk.

Another approach to persistent programming is to provide a persistent class library for an object-oriented programming language. Arjuna [Group92] is an example. The state of each persistent object is stored as a file. The state management mechanism implemented by `StateManager` provides support for the use of persistent and recoverable objects. The users of the Arjuna system have to be concerned about defining appropriate `save_state` and `restore_state` operations for the class to translate data formats between main memory and backing store. They also have to implement the `type` operation indicating where the object should be saved in the object store. Distribution is based on the client-server model. When client code invokes an operation on a remote object, the operation is actually invoked on the client stub object. This stub object packs up the parameters to the operation with sufficient information for the correct operation to be invoked at the server side and transmits the request via RPC to the remote server stub object. The server stub object unpacks the data received and invokes the specified object operation. When the operation has completed the stub objects are responsible for packing and transmitting the reply to the client program.

Some other persistent programming language systems extend existing languages or design new languages to support persistent classes. C** on Comandos [Cahill93] is an example of the former. Distribution and persistence are provided through extensions to the C++ language while concurrency and transactions are provided through library classes. New keywords are added. The keyword `global` is used to distinguish distributed objects while `permclass` is used to declare persistent classes and `volclass` is used to define non-persistent classes. New operators and operations are introduced to support them. The Comandos Object-Oriented Language [Cahill93] is an example of new languages supporting persistent classes.

It is beyond the scope of this dissertation to design a complete persistent programming system. As will be shown in this chapter, however, supporting distributed persistent programming on COMMOS is easy and straightforward. C++ is an object-oriented programming language widely used in industrial and academic environments. It is chosen to demonstrate how distributed persistent programming can be supported. What should be done is to define or overload C++ standard operators and build a class library using the object

mapping and locking mechanisms provided by the COMMOS. A GNU g++ cross compiler has been configured and installed on mips Ultrix to generate executable code for Wanda. All the code fragments given in this chapter have been tested.

7.2 Defining C++ Class Operators

In C++ [Stroustrup91], an object can be created by the operator `new` and destroyed by the operator `delete`.

The `new` operator attempts to create an object of the object type to which it is applied and returns a pointer to the object created. It will call the function `operator new()` to obtain storage. The first argument must be of type `size_t`, an implementation-dependent integral type defined in the standard header `<stddef.h>`.

An object created by the `new` operator exists until it is explicitly destroyed by the `delete` operator. The `delete` operator may be applied only to a pointer returned by `new` or to zero.

It is possible to take over memory management for a class by defining `operator new()` and `operator delete()`. This remains possible and is even more useful for a class that is the base for many derived classes. This feature makes supporting distributed persistent programming on COMMOS very straightforward. What should be done is to define a base class for persistent object classes and implement the operators `new` and `delete` to map persistent object states. The following code fragment illustrates one of the possible implementations.

```
enum operation {OBJ_CREATE, OBJ_MAPPING};
     // OBJ_CREATE:  create a system-level object;
     // OBJ_MAPPING: map a system-level object.

extern struct objmap* myprocmap; // the process map.

class PersistClass {
 public:
  ...
  void operator new(size_t size, char* name,
                    char* type, operation op);
  void operator delete(void* ptr);
```

```
  ~PersistClass();
};
...

void* PersistClass::operator
new(size_t size, char* name, char* type, operation op)
{
  long vir_id;
  void* ptr;

  switch (op) {
  case OBJ_CREATE:
    vir_id = CreateObject(type, name, ...);
    if (vir_id < 0) {
      printf("PersistClass::new: failed to create object %s\n",
             name);
      exit(1);
    }
    break;
  case OBJ_MAPPING:
    vir_id = MapObject(type, name, ...);
    if (vir_id < 0) {
      printf("PersistClass::new: failed to map object %s\n",
             name);
      exit(1);
    }
    break;
  default:
    printf("PersistClass::new: unknown operation %d\n", op);
    exit(1);
  }
  ptr = (void*) myprocmap[vir_id].ob_addr;
  return ptr;
}

void PersistClass::operator delete(void* ptr)
{
  long vir_id = -1, i, rc;
  char* name;

  for (i = 0; i < N_OBJECTS; i++) {
    if (ptr == (void*) myprocmap[i].ob_addr) {
      vir_id = i;
```

```
        break;
      }
    }
    if (vir_id < 0) {
      printf("PersistClass::delete: failed to locate the object\n");
      exit(1);
    }

    rc = UnmapObject(vir_id, -1);
    if (rc != 0) {
      name = myprocmap[vir_id].ob_name;
      printf("PersistClass::delete: failed to unmap object %s\n",
             name);
      exit(1);
    }
  }
  ...
```

After defining the operators new and delete for the base class of all persistent object classes, application programmers can define their own persistent classes as derived classes of the base class and use operators new and delete to create and destroy objects. For example:

```
class BookEntry: public PersistClass {
  char author[MAX_NAME];
  char title[MAX_TITLE];
  char publisher[MAX_PUBLISHER];
  int  year;
 public:
  void Display();
  ...
  ~BookEntry() {};
};

main()
{
  BookEntry* entry;

  entry = new ("/homes/abc/book1", "persist", OBJ_CREATE) BookEntry;
  entry->Display();
  ...
  delete entry;
}
```

The implementation described above does not support class constructors. According to the C++ Reference Manual [Stroustrup91], class constructors should work with the user defined class operators. If class constructors are defined, however, the code does not compile using GNU g++ or SunOS CC. Therefore, application program have to initialise objects in hand.

7.3 Overloading C++ Operator

Initialising objects in hand is error prone and inelegant. Fortunately, class constructors work fine with an overloaded global new operator. In C++, several different function declarations can be specified for a single name in the same scope and the name is said to be *overloaded*. When that name is used, the correct function is selected by comparing the types of the actual arguments with the types of the formal arguments. The unique best matching function is then called. Using overloading, the implementation is as follows:

```
enum operation {OBJ_CREATE, OBJ_MAPPING};
     // OBJ_CREATE:  create a system-level object;
     // OBJ_MAPPING: map a system-level object.

extern struct objmap* myprocmap; // the process map.

class PersistClass {
 public:
  PersistClass();
  ...
  void operator delete(void* ptr);
  ~PersistClass();
};
...

void* operator
new(size_t size, char* name, char* type, operation op)
{
  long vir_id;
  void* ptr;

  switch (op) {
  case OBJ_CREATE:
    vir_id = CreateObject(type, name, ...);
```

```
    if (vir_id < 0) {
      printf("new: failed to create object %s\n", name);
      exit(1);
    }
    break;
  case OBJ_MAPPING:
    vir_id = MapObject(type, name, ...);
    if (vir_id < 0) {
      printf("new: failed to map object %s in\n", name);
      exit(1);
    }
    break;
  default:
    printf("new: unknown operation %d\n", op);
    exit(1);
  }
  ptr = (void*) myprocmap[vir_id].ob_addr;
  return ptr;
}

void PersistClass::operator delete(void* ptr)
{
  long vir_id = -1, i, rc;
  char* name;

  for (i = 0; i < N_OBJECTS; i++) {
    if (ptr == (void*) myprocmap[i].ob_addr) {
      vir_id = i;
      break;
    }
  }
  if (vir_id < 0) {
    printf("PersistClass::delete: failed to locate the object\n");
    exit(1);
  }

  rc = UnmapObject(vir_id, -1);
  if (rc != 0) {
    name = myprocmap[vir_id].ob_name;

    printf("PersistClass::delete: failed to unmap object %s\n",
           name);
    exit(1);
```

```
  }
}
...
```

7.4 Supporting Fine Grained Objects

The implementations discussed in Sections 7.2 and 7.3 are simple. The state of each persistent object is stored as a system-level object. This should be sufficient to support medium and coarse grained objects. However, in some applications, supporting fine grained persistent objects might be desired. In this case, it is desirable to cluster language-level fine grained objects into system-level objects. By clustering together objects that are frequently referenced together, locality is increased, main memory is used more efficiently and fewer pages need to be transferred in order to access the objects [Gourhant92]. It is important for performance purpose. This section illustrates how to implement object clustering but it does not attempt to address object clustering algorithms. Readers who are interested in object clustering algorithms are referred to research results on object-oriented systems and database systems, such as [Gourhant92] and [Chang89].

A possible approach for object clustering is to have all the objects of the same class stored in a system-level object. In this case, the operators `new` and `delete` must be implemented for each class to allocate and deallocate memory from a system-level object.

A more flexible approach is to define a persistent object container class and define the operators `new` and `delete` of the persistent object class to allocate memory for objects from different containers. In this scheme, whenever an application creates a new persistent object, it can specify the container in which that object should be created. The following code fragment demonstrates how this can be done.

```
enum operation {OBJ_CREATE, OBJ_MAPPING};
     // OBJ_CREATE:  create a system-level object;
     // OBJ_MAPPING: map a system-level object.

extern struct objmap* myprocmap; // the process map.

class Container {
 public:
```

```
  virtual void* alloc(size_t) = 0;
  virtual void free(void*) = 0;
};

class PersistContainer: public Container {
  char name[MAX_OBJ_NAME];
  char type[MAX_TYPE_NAME];
  long vir_id;
  void* addr;
  ...
 public:
  PersistContainer(char* nm, char* tp, long sz);
                                    // create a new container.
  PersistContainer(char* nm, char* tp); // map in a container.
  void* alloc(size_t);
  void free(void*);
  ~PersistContainer();
};

PersistContainer::PersistContainer(char* nm, char* tp, long sz)
{
  vir_id = CreateObject(tp, nm, ...);
  if (vir_id < 0) {
    printf("PersistContainer: failed to create object %s\n", nm);
    exit(1);
  }
  strcpy(name, nm);
  strcpy(type, tp);
  addr = (void*) myprocmap[vir_id].ob_addr;
}

PersistContainer::PersistContainer(char* nm, char* tp)
{
  vir_id = MapObject(tp, nm, ...);
  if (vir_id < 0) {
    printf("PersistContainer: failed to map object %s\n", nm);
    exit(1);
  }
  strcpy(name, nm);
  strcpy(type, tp);
  addr = (void*) myprocmap[vir_id].ob_addr;
}
```

```
PersistContainer::~PersistContainer()
{
  int rc;

  rc = UnmapObject(vir_id, -1);
  if (rc != 0) {
    printf("~PersistContainer: failed to close the object %s\n",
           myprocmap[vir_id].ob_name);
    exit(1);
  }
}
...

class PersistClass {
 public:
  void* operator new(size_t size, Container* ct);
  ...
  ~PersistClass();
};

void* PersistClass::operator new(size_t size, Container* ct) {
  return ct->alloc(size);
}
...
```

Now a BookEntry object can be allocated from a persistent object container but the destructor must be called explicitly when the object is no longer needed.

```
extern Container* percon;

entry = new (percon) BookEntry;
...
entry->BookEntry::~BookEntry();
percon->free(entry);
```

7.5 Other Issues

Persistence support is discussed in the previous sections. In order to provide distributed programming, distribution and concurrency control are also required. Since object coherency is guaranteed by COMMOS, no extra mech-

anism is necessary for these. The locking mechanism provided by the underlying system makes two-phase locking possible and hence it is not difficult for the class library implementor to build transaction support.

7.5.1 Working with Pointers

Another issue which needs to be considered is pointers. This raises the question of how pointers are stored and used. The most flexible approach is to encoded pointers as *surrogates* defined in some external naming context and to translate them at reference time. The *forwarder* in [Bliot90] is one of the examples. This approach could be used but the cost of dereferencing is high.

Pointer swizzling has been proposed to reduce the cost of interpreting pointers by caching the result of a translation, overwriting the surrogate in place with the virtual address. [Wilson92] describes a technique via which all objects in a given page are swizzled at the time the page is brought into memory. This technique can utilise the virtual memory hardware to detect accesses to non-swizzled objects. While these techniques do work, they still incur considerable overheads. For example, Wilson's scheme requires that all pages be scanned as they are brought into memory and there are considerable costs during page discard. Also, in COMMOS there is at most one copy of objects on each node possibly shared by multiple processes. This further complicates the pointer swizzling scheme. An object address allocation mechanism is desirable to ensure that an object is mapped at the same address in all processes on the same node.

A frequently used alternative approach is to map objects at fixed, non-overlapping virtual addresses. In this approach, objects occupy permanently a piece of virtual memory and pointers always point to the correct place. Therefore, pointers can be stored in secondary storage in their in-core format and pointer swizzling is avoided. Monads [Rosenberg92] and Opal [Chase92] are some examples which use this approach. The major problem with this approach are that a virtual address space might not be big enough to accommodate unlimited numbers of objects.

Another approach is the *persistent context space* model proposed by [Amaral92]. In this model, objects are bound to addresses and are organised into *containers*. Direct references between data in different containers are not allowed (between different containers objects reference each other through surrogates). When mapped into an address space, each container

forms a *complete memory space* in which there are no unresolved references. A *persistent context* is a context that has one or more containers mapped into its address space. A *persistent context space* is made up of a set of distributed persistent contexts, which are able to share a single uniform address space where one or more containers can be mapped. A persistent application is executed in parallel by a number of processors that have contexts in one or more sites with a common global address space of persistent memory. Within a persistent context space, the uniqueness of address allocation for shared persistent objects is guaranteed. With relocation, the binding between a container and a range of addresses may be changed. This model allows direct object pointers at system-level and retains the multiple address space environment model at the same time. This approach seems more suitable for the COMMOS architecture.

7.6 Summary

Supporting distributed persistent programming above COMMOS has been explored in this chapter. No modification to compilers is required and the class library implementors and the end users need not be concerned about the location of objects and data movement between main memory and backing store. Although no application has been built due to time constraints, the discussion in this chapter has shown that the COMMOS architecture is flexible and easy to use.

Chapter 8

Related Work

The use of memory-mapping techniques to support data persistency in distributed computing environments has become increasingly popular in recent years and there exist many research projects and systems around the world which employ it. This chapter does not attempt to exhaustively survey all of them. Instead, it briefly reviews the projects and systems closely related to COMMOS and then compares their important aspects in the last section.

8.1 Apollo Domain

Apollo Domain [Leach83], developed at the Apollo Computer Inc in USA, was one of the earliest systems to assure coherency of shared memory-mapped objects in a local area network of personal workstations and data servers.

In Apollo Domain, objects were typed, protected, abstract information containers addressed by *unique identifiers* (UID). An **object** was in fact an uninterpreted byte sequence with arbitrary length. Associated with each object was the UID of a type descriptor, the UID of an access control list object, a disk storage descriptor and some other attributes. Programs access all objects by presenting their UIDs and asking for them to be mapped into the program's address space. Subsequently, they are accessed with ordinary machine instructions, utilising virtual memory demand paging. On a page fault, the *object storage server* locates and reads the object from the corresponding disk.

A two-level approach was used to assure coherency of the replicated copies of an object. In the lower level, a timestamp corresponding to the time that the object was last modified is used to detect concurrency violations. Every

node remembers the timestamp for all remote objects whose pages it has encached in its main memory. Every time an object's page is read from another node, its timestamp is returned with it. If it is the only page of the object encached in this node, its timestamp is remembered. Otherwise, the returned timestamp must match the remembered timestamp for the object. If not, a read concurrency violation has occurred. Every time a page of an object is written back to its home node, the current timestamp is sent with the write request and an updated timestamp is returned. The home node will only accept the page if it comes from the current version of the object. Otherwise a write concurrency violation has occurred. A page write updates both the home node's and the requesting node's timestamp for the object. The system also provides primitives to discard stale pages of a cached object, to inquire about the current timestamp of an object, and to send back modified pages of a cached object. The higher level provides an object locking mechanism. Multiple-reader/single-writer locks are supported. Lock and unlock requests for remote objects are always sent to the home node. A lock request that is granted returns the current timestamp of the object, which is used to remove stale pages from the requesting node's main memory. The unlock operation forces modified pages back to the home node before the lock is released. Lock requests are not enqueued. If a lock is currently in use requests for the lock are denied and the requesters have to retry later.

Performance of reading a page from the local disk, remote disk and remote main memory was reported.

8.2 Mach

Mach [Accetta86] is a portable, multiprocessor operating system developed at Carnegie Mellon University, USA. One of the major goals of Mach is to move more and more functionality out of the kernel, until everything is done by user mode tasks communicating via the kernel. There are five main kernel abstractions:

- A **task** is an execution environment in which threads may run. It is the unit of resource allocation. A task includes a paged virtual address space and protected access to system resources.
- A **thread** is the unit of CPU utilisation. A thread belongs to one and only one task that defines its virtual address space. All threads within a task share access to all task resources.

- A **port** is a communication channel, accessible only via send/receive capabilities. Every system entity except virtual memory ranges is named by a port.

- A **message** is a typed collection of data objects used in communication between threads.

- A **memory object** is the internal unit of memory management. It is a collection of data provided and managed by a server which can be mapped into the address space of a task.

The Mach virtual memory system [Rashid88] is clearly layered into the machine dependent and the machine independent part. The machine dependent part is concerned with managing the MMU hardware. It provides a simple interface for validating, invalidating and setting the access rights for pages of virtual memory. The machine independent part provides support for virtual address spaces, memory ranges within an address space, and the interface to the backing store for these ranges via the external management interface.

Memory objects can be created and serviced by a user-level *memory manager* (also called *external pager*). The memory manager is entirely responsible for the initial values of the memory object and the permanent storage if necessary. The interface between memory managers and the kernel consists of three parts: calls made by an application program to cause a memory object to be mapped into its address space; calls made by the kernel to the memory manager to initialise a memory object and to carry out page-in/page-out operations; and calls made by the memory manager to the kernel to control the use of memory objects.

When a memory object is mapped by tasks on multiple network nodes, a global shared memory manager called the *fault scheduler* is employed to manage strict coherency between multiple copies of memory pages using centralised coherence algorithms. A *distributed paging server* [Forin88] has also been developed to support distributed coherence algorithms. Coherency is at the page level and the algorithms adopted are write-invalidate protocols similar to those used in the IVY system [Li86]. Some performance measurements are reported but no explicit figure about servicing remote page faults is given.

8.3 Chorus

Chorus [Rozier88] is a distributed, scalable operating system developed by Chorus Systems, France. A Chorus system is composed of a minimal *Nucleus* and a set of independent system servers that rely on the basic, generic services provided by the Nucleus. The most important abstractions in Chorus are:

- An **actor** is a protected virtual address space that can be either in user mode or in supervisor mode. Any given actor is tied to a node and a given node can support many simultaneous actors.
- A **thread** is the unit of execution. A thread is always tied to one and only one actor. One or more **threads** can run simultaneously within an actor. These threads share the resources of that actor and can communicate using the shared memory provided by the address space of the actor.
- A **message** is a contiguous byte string used for interprocess communication.
- A **port** is an address, designated with an unique identifier, to which messages can be sent, and a queue holding the messages received but not yet consumed by the threads. A port can only be attached to a single actor at a time, but can be successively attached to different actors, effectively migrating the port from one actor to another. Ports may also be grouped together dynamically to form port groups.

The Chorus virtual memory system [Abrossimov90] provides support for separate address spaces, efficient and versatile mechanisms for data transfer between address spaces, and between secondary storage and an address space. Secondary storage objects, called **segments**, are managed outside of the memory manager subsystem by external pagers called *segment mappers*. Segments are named by capabilities, containing the mapper's port name and a key. The key is opaque to the system, and used by the mapper to manage and protect segment access. The mapper is always invoked using the Chorus RPC mechanism.

A segment can be mapped into many actor address spaces on many nodes at the same time. It can also be accessed by explicit operations by any number of threads. When a segment is shared among different nodes, the segment mapper is in charge of maintaining the segment coherency. A centralised mapper was developed to support a centralised coherence algorithm as described in the

IVY system. Another set of mappers [Abrossimov92], which includes a global mapper and a per node local mapper, have been implemented to support a dynamic decentralised coherence algorithm used in the IVY system.

8.4 The V System

The V system [Cheriton88a] is an operating system developed at Stanford University as part of a research project to explore issues in distributed systems. The system is structured as a relatively small distributed kernel and a set of service modules. The V kernel provides a network transparent abstraction of address spaces, lightweight processes and interprocess communication.

In the V kernel memory management system [Cheriton88b], an address space consists of ranges of addresses, called **regions**, bound to some portions of open files. A reference to a memory cell of a region is semantically a reference to the corresponding data in the open file bound to this region. A simple ownership protocol is used in conjunction with a lock manager at the backing server to implement region coherency.

V++, the extended V system, supports *external page-cache management* [Harty92], which provides applications with a page frame cache abstraction to monitor and control the amount of physical memory available for execution, the contents of this memory and the scheduling and nature of page-in/page-out. The mechanism consists of a number of *segment managers* and a *system page cache manager* (SPCM); they run in user space. The segment manager is similar to external pagers in Mach and Chorus except in maintaining free page frames allocated by the SPCM. The SPCM allocates from the global memory pool among the segment managers and enables monitoring and controlling of the physical memory availability for different applications. Performance is measured but no figure about servicing remote page faults is given.

8.5 Clouds

Clouds [Dasgupta91] is a distributed object-based operating system developed at the Georgia Institute of Technology, USA. The Clouds system is composed of persistent named address spaces called **objects**. Clouds objects encapsulate code and data and provide data storage, data manipulation, data sharing,

concurrency control, and synchronisation. Control flow is achieved by **threads** invoking objects. A thread is not associated with a single address space and may span machine boundaries during the course of its execution. Data flow is achieved by parameter passing. The Clouds implementation has three levels: the minimal kernel *Ra* which provides the mechanisms for managing basic resources, namely processor and memory; a set of system objects which are trusted software modules providing essential system services; and finally some user objects which provide non critical services such as naming and spooling.

A virtual address space in Clouds is partitioned into three distinct regions: *O* space for the object, *P* space for the thread, and *K* space for the kernel. Storage in the system is represented by **segments**. A segment is an arbitrary length uninterpreted sequence of bytes which can be mapped into virtual address spaces. Associated with a segment are static storage attributes such as copy-on-write, persistent, and non-persistent, as well as static access control information such as read-only and read-write.

Clouds supports persistent distributed shared memory with memory coherency being maintained by a lock-based multiple-reader/single-writer protocol at the segment level [Ananthanarayanan92b]. Performance measurements are taken on a configuration of Sun 3/60s (MC68020, 16 MHz clock) connected by a 10Mbps Ethernet. Getting a 8-Kbyte page from a remote data server takes 15.50 milliseconds without forwarding and 18.50 milliseconds with forwarding. Further research in the Clouds project concludes that no single coherence protocol can perform well for all types of application [Ananthanarayanan92a]. They proposes a solution which requires the application programmers to use the primitives provided by the system to maintain the memory coherency by themselves. The users have to be completely aware of the underlying DSM mechanisms and have to explicitly activate the coherence operations.

8.6 Choices

Choices [Campbell93] is an operating system for distributed and shared memory multiprocessor systems developed at the University of Illinois at Urbana-Champaign, USA. It uses class hierarchies and object-oriented programming to facilitate the construction of the operating system.

A virtual address space is referred to as a **Domain**. One or more lightweight **Processes** can execute within one domain. The entity for sending and receiv-

ing messages is the **MessageContainer**. It is named and can have multiple senders and multiple receivers.

In the Choices virtual memory [Russo89], a **MemoryObject** is a logical collection of data, which may be a process stack, code, heap, or data of a program. Subclasses of MemoryObject, such as **PersistentStore**, which represents various kinds of disks and files, may be used. A MemoryObject is made accessible to a process by its **MemoryObjectCache**. After being cached, the contents of the MemoryObject may be referenced by virtual memory addresses. A MemoryObject can be mapped into multiple Domains to provide shared memory.

DistributedMemoryObjectCache and **PageRecord** were added to support distributed virtual memory [Sane90] in Choices. An instance of the DistributedMemoryObjectCache provides a local physical memory cache for the copy of the shared data on a network node. It is responsible for locating and retrieving pages to satisfy virtual memory faults generated by the processes on its node. Memory coherence protocols are implemented in the PageRecord and memory coherency is maintained at the page level using a write-invalidate multiple-reader/single-writer protocol. Performance is evaluated by comparing the timing for applications in DSM and in shared memory multiprocessor systems.

8.7 Spring

Spring [Hamilton93] is a distributed, multi-threaded operating system developed at Sun Microsystems Laboratories, USA. System resources are represented as **objects**. A Spring object is an abstraction that contains state and provides a set of methods to manipulate the state.

A Spring **domain** is an address space with a collection of **threads**. A cross-domain call chain, which is a series of application visible threads, is referred to as a **shuttle** and is an entity schedulable by the kernel. A **door** represents an entry point for a cross-domain call associated with both an entry point program counter (PC) and an integer datum that can be used to identify a particular object in the target domain. The Spring kernel supports basic cross domain invocations, threads, and low-level machine-dependent interrupt handling and provides basic virtual memory support for memory mapping and physical memory management. Other operating system functions run as user-mode servers.

There are two sets of agents that cooperate to provide virtual memory in the Spring operating system [Khalidi93a]. A per-node *virtual memory manager* is responsible for handling mapping, sharing, and caching of local memory. The virtual memory manager depends on external pagers for accessing backing storage and maintaining inter-machine coherency.

The main components in the Spring virtual memory system are **address spaces** and **memory objects**. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of memory, which may have corresponding secondary storage, that can be mapped into address spaces. The main operations on address space objects are to map and unmap memory objects into selected address ranges of the address space. Operations are also provided to allocate new *zero-filled* memory. A memory object can be mapped into more than one address space at the same time on more than one network node.

The virtual memory architecture defines two other types of objects: the **pager** object and the **cache** object. The pager object is implemented by external pagers and provides operations to page in and out memory blocks and is used by the virtual memory manager to populate a local memory cache. The cache object is implemented by the virtual memory manager and is used to build a two-way communication channel between the virtual memory manager and the pager of a cache object. Cache coherency is at the page level and a centralised control multiple-reader/single-writer protocol is implemented [Nelson93].

8.8 Comandos

Comandos [Cahill93] is an ESPRIT project which aims to provide an integrated environment for the construction of distributed applications.

The Comandos *virtual machine* provides management of persistent storage, control of distributed computations, network communications and transaction management. At the upper level of the virtual machine is the *generic runtime* which provides a language independent layer implementing distributed object invocation. The lower level of the virtual machine is the *kernel* layer, including those components which must be implemented in a protected way, which interfaces directly with the underlying host environment.

From the virtual machine's point of view, Comandos **objects** are simply con-

tiguous blocks of memory. A supported language may bind class code to objects through its *language specific run-time* library. The objects existing in a Comandos system are divided into a set of non-overlapping **extents.** Every object belongs to exactly one extent. At any time some of the objects belonging to an extent may be stored in secondary storage while others may be mapped into virtual memory at various nodes. A **context** is a dynamically varying collection of objects located at the same node and may contain objects from one extent only. There may be only one context for a given extent at any given node. All objects belonging to an extent that are in use at a node are mapped into the single context for that extent at the node. A **cluster** contains a set of persistent objects which are mapped into one contiguous region of virtual memory. **Jobs** and **activities** are the units of distributed processing of objects. An activity is a sequence of synchronous invocations on one or more objects possibly on different nodes and a job is a set of one or more activities. Jobs and activities communicate by means of shared objects. *Virtual object memory* (VOM) implements the distributed object space manipulated by jobs and activities.

8.8.1 Amadeus

The Amadeus platform [Cahill93] is the reference implementation of the Comandos virtual machine developed at Trinity College, Dublin, Ireland. The Amadeus *kernel* is implemented as a guest layer on Unix. Each extent is defined by a *text image* which includes all of the classes in the extent. Each context is implemented as a Unix process. Activities are represented by threads in each visited context. At any time, a cluster can be mapped into at most one context anywhere in the system. In order to access a shared object, activities running on nodes other than the one which has the object mapped have to make cross-context invocations via RPC. The Amadeus platform has been ported to run on Mach 3.0 Unix servers.

8.8.2 COOL

Chorus Object-Oriented Layer (COOL) [Lea93] is the Chorus implementation of the Comandos virtual machine developed in Chorus Systems, France. The *COOL-base* extends the Chorus microkernel to support distributed persistent virtual memory by the *context space* model. A context space is a collection of distinct address spaces on one or more nodes. Any cluster belonging to a

context space is mapped into all contexts of that context space at the same range of addresses. Coherency between clusters in a context space is managed by the distributed virtual memory mapper described in section 8.3.

8.8.3 Guide-2

Guide-2 [Balter93] is a native implementation of the Comandos system on top of the Mach 3.0 microkernel operating system developed at the Unite Mixte Bull-IMAG/Systems, France. Any context within a job is represented by a Mach task. A job is hence a collection of Mach tasks. A major feature of Guide-2 is the extensive use of memory management facilities offered by Mach. Clusters are managed by a set of Mach memory managers as described in section 8.2. The memory manager which manages a cluster controls the coherency of the shared data. The performance of reading a page from and writing a page to the secondary storage is reported but no figure about the coherency mechanism is given.

8.9 Casper

Casper [Vaughan92] is a distributed persistent store architecture developed at the University of Adelaide in Australia. It aims to support the persistent programming language Napier88 using facilities, such as memory mapping and the external pager, provided by the Mach operating system.

The system consists of a *Stable Store Server* (SSS) and a number of clients. The main functions of the SSS are managing the supply of pages on demand to clients, ensuring that coherent versions of the pages are supplied and maintaining the integrity of the stable store. The SSS interface to the clients is provided by the *Server Request Handler* (SRH). Each client has an interface called the *Client Request Handler* (CRH) for communicating with the SSS and other clients.

The cache coherence protocol used in Casper is a central directory, multiple-reader/single-writer protocol. All read and write requests are made directly to the SSS. The SSS maintains all the information concerning the distribution and modification of pages. If a page has been modified since the last stabilisation and a current copy is not available in the store, read requests for the page are forwarded to a client with an up-to-date copy of that page. The

server only services requests itself when it holds a valid copy of the page. If a client wishes to modify a page, it must already have read access to that page and send the SSS a modification request. The SSS next instructs all clients which have read access to invalidate the page. These clients must reply with an acknowledgement to the SSS on completion of the invalidation. Once all acknowledgements are received the SSS sends a *Write Acknowledge* signal to the originally requesting client and the client can go ahead to modify the page.

A client consists of three main threads: the user program, the CRH and the external pager. The external pager handles any page faults or protection faults caused by the user program. The CRH and the external pager jointly implement the client's part of the cache coherence protocol.

8.10 Opal

Opal [Chase92] is an operating system for a distributed environment developed at the University of Washington, USA. It defines a single address space that maps all data in the system including persistent data. The global address space is extended across the network by placing servers on each node that maintain a partitioning of the address space, both to ensure global uniqueness and to allow data to be located from its virtual address.

The units of execution are **threads**. A **protection domain** provides the execution context for threads, restricting their access to a specific set of segments at a particular instant in time. All the protection domains share a single virtual address space. The units of storage allocation and protection are **segments**, which are virtually contiguous extents of pages and potentially persistent. The virtual address of a segment is a permanent attribute, fixed by the system at allocation time. Once created, a segment can be explicitly attached to protection domains, permitting threads within those domains to access the segment. Conversely, segments can be explicitly detached to deny access. Threads, protection domains and segments are named by **capabilities**. Threads in different protection domains can communicate using shared memory if their segment privileges overlap. In addition, control can be transferred between domains. A **portal** is an entry point to a domain, uniquely identified by a 64-bit value. Any thread that knows the value of a portal name can make a system call that transfers control into the protection domain named by the portal.

An Opal prototype is implemented on top of the Mach microkernel operating

system [Chase93]. Within the single address space, Opal protection domains are implemented as Mach tasks. Opal segments are implemented as Mach memory objects. Memory coherency has not been supported in the prototype implementation.

8.11 Pegasus

Pegasus [Leslie93] is a joint project of the University of Cambridge in the UK and the University of Twente in the Netherlands. The major goal is to design and implement an operating system that allows capture, rendering, storage, and interactive processing of multimedia data by user-level applications, while keeping all of the desirable properties of distributed systems, such as resource sharing, data sharing, security, and fault tolerance.

It is proposed that a shared address space could be used by local groups of mutually trusted machines that share the same data representation. A Pegasus address space is long lived, that is, the address space survives processes and processor crashes. The operating system manipulates memory objects called **segments**. Most segments are linked to typed external objects referred to as files. A segment occupies a contiguous range of virtual addresses and may be paged in and out.

8.12 Comparison

A great deal of effort has been made to support data persistency in distributed systems using memory-mapping techniques as shown by the related projects and systems reviewed in this chapter.

Apollo Domain had the same goal as COMMOS, to provide a coherent integrated virtual memory in distributed systems, but it was not a microkernel system. Also, although it had the notion of *type* similar to that in COMMOS it did not make use of this property to provide clients more flexibility to choose the way objects are managed.

Mach, Chorus and the V system share many common points which are used in COMMOS. They are microkernel operating systems. Lightweight processes and efficient message passing are supported and memory-mapped secondary storage objects are managed by external pagers. V++ takes a step further

to allow applications to control the usage of physical memory. Unlike COMMOS, however, memory coherency in distributed environments is maintained by external pagers at either segment or page level and each external pager usually supports only one coherence protocol.

Clouds, Choices and Spring are organised using an object-oriented approach. All components of the operating system are treated as objects. In Clouds and Spring, there is no explicit interprocess communication mechanism and all communications are carried out via object invocations. Spring borrows many ideas from Mach and Chorus, especially in virtual memory management. It develops the notion of the external pager by separating the memory object representing memory from the pager object that provides the methods to page-in and page-out the memory. This separation allows the memory object and the pager object to be placed in different domains. The performance of fetching a page from a remote server in Clouds is faster than that in COMMOS. The main reason is that low-level message passing is used and no disk access is involved in the measurements. Researchers in the Clouds project have noticed that no single coherence protocol can perform well for all types of applications. However, the solution they proposed to support multiple coherence protocols requires the applications to explicitly invoke the coherence operations.

The major goals of Comandos and Casper are to build a platform to support persistent programming languages on top of existing operating systems. There is no coherency problem to tackle in the Amadeus implementation since mapping an object simultaneously into multiple address spaces on different nodes is forbidden. The COOL-base simply makes use of the underlying distributed virtual memory mapper to maintain object coherency while Guide-2 developed their own coherence mechanism in external pagers.

Finally, Opal and Pegasus are both trying to explore the use of 64-bit wide address spaces. As demonstrated in the Opal prototype implementation, these abstractions can easily be constructed on top of a system like COMMOS.

To sum up the discussion, COMMOS exploits the advantages of the microkernel approach, the memory-mapping technique and the typed object principle to provide a flexible and well-defined interface for coherent memory-mapped object management in distributed computing environments. One important feature of COMMOS is the separation of the coherence server from the external pager so that multiple coherence protocols can be supported via a generic interface. This allows different coherence protocols to be applied not only to different types of object but also to different individual system-level objects. The dynamic adoption of a coherence protocol can also be supported.

Chapter 9

Conclusions

In this chapter, the research on the COMMOS architecture is summarised and the results are highlighted. Some suggestions for further work are then given. It is concluded that COMMOS provides a flexible approach to future system design.

9.1 Conclusions

A **coherent memory-mapped object system** (COMMOS) architecture has been proposed to restructure virtual memory to support distributed computing environments. The requirements for the integration of virtual memory with secondary storage mechanisms and local memory with remote memory were deduced from an analysis of existing virtual memory and storage management and the observation of their limitations, particularly in the emerging microkernel architectures and distributed computing environments.

Application programmers are often distracted by different views of volatile data and persistent data in traditional two-level store systems. This impairs the quality and productivity of software development. Besides, the two-level store system is not efficient because of mandatary data copying and user-kernel boundary crossing. This is exacerbated in a microkernel architecture where most of the user-kernel boundary crossings become context switches. Due to double paging, resources are not used efficiently and the double paging anomaly may occur in database systems implemented on top of two-level store systems. These lead to the desirability of integrating main memory with secondary storage. With careful design, the memory-mapping technique can

be used for this purpose.

In distributed environments, especially in the wide address space architectures, DSM is becoming increasingly attractive since it is easier to program than the message-passing abstraction. Coherency control, however, is expensive. Also, existing DSM systems typically provide only one coherence protocol. There exists a potential mismatch between the supplied coherence protocol and some applications' requirements. Current distributed file systems rely on client caching to achieve high performance. The cache coherency problem exists here too. It is desirable that some uniform and flexible approach could be employed to integrate local memory with remote memory. The flexible coherence mechanism proposed in this dissertation can be used to reduce the coherency overhead and to avoid coherence protocol mismatch.

An outline of the architecture framework of COMMOS was presented. In particular, the mismatch between cache coherence protocols and application requirements was discussed. It was proposed that coherence servers are separated from external pagers to support a flexible coherence mechanism for multiple coherence protocols. This, combined with the typed object model, allows clients to choose the most suitable protocols for their applications and hence solves the problem of protocol mismatch.

The dissertation then focussed on the design of a flexible coherence mechanism. Various design issues, such as granularity, remote communication tools, and the policy used for writing the modifications back to secondary store, were discussed. Several modified coherence protocols, namely centralised-control and distributed-control write-invalidate protocols and a write-update protocol were described. The integration of low-level coherency control with high-level concurrency control realises system-wide object coherency and synchronisation without severely impacting the system performance.

A prototype implementation of COMMOS which realises both write-invalidate protocols was given and performance measurements were presented. The prototype proves that the proposed COMMOS architecture is practicable.

The application of COMMOS was demonstrated by the exploration of how to build a C++ class library to support distributed persistent programming. This showed that the use of COMMOS is indeed flexible and straightforward.

Some related systems and projects, which share the same goal to support data persistency in distributed systems using the memory-mapping technique, were reviewed and compared with COMMOS.

9.2 Further Work

The investigation of coherent memory-mapped object system architecture in this dissertation has centred on the support of a flexible coherence mechanism for multiple coherence protocols and the exploitation of the typed object principle to enable clients to choose the most suitable protocols for their applications. This section suggests some further work that can be done. Perhaps the easiest and most obvious extension of this work is to provide more flexible coherence schemes. Further research on supporting heterogeneous architectures is interesting. In addition, exploiting COMMOS functionality to support persistent programming languages, distributed database systems and computer-supported cooperative work are ambitious and useful projects.

9.2.1 More Flexible Coherence Schemes

The current design of COMMOS enables clients to advise the system to apply different coherence protocols to different types of object. More flexible schemes may be desirable and are easy to be added to the current implementation.

Individual Object Coherency COMMOS is based on sharing typed data objects rather than a flat virtual address space. It is possible to apply different coherence protocols to different individual objects. One way of realising this is to build a mini-database for storing and retrieving information about which coherence protocol should be applied to a specific object. This information would then be used to override the type description about the coherence protocol when the object is created or mapped.

Adaptive Coherency Control The runtime situation, such as the work load of the system, the concurrency level of a particular object and the access pattern to an object may change dynamically. In order to optimise system performance in a changing world, it may be required that different coherence protocols are used when circumstances change. Some mechanism which monitors the system might be built and used to choose dynamically the coherence protocol according to the runtime condition.

9.2.2 Support for Heterogeneous Architectures

COMMOS currently supports only homogeneous systems. It is worth exploring whether it is feasible to support a coherent memory-mapped object system architecture in heterogeneous systems.

Multiple Page Sizes In a heterogeneous system or a homogeneous system in which different machines use different page sizes, support for multiple page sizes in the coherence mechanism is desirable. A *server page size* can be adopted by the coherence mechanism for its own use. For requests which are smaller than the server page size, the request may be rounded up. For requests which are larger than the server page size, the request is satisfied by multiple server pages.

Data Representation Translation Interfacing heterogeneous architectures raises not only the problem of potentially different page sizes, but also the problem of different data representations. A server for translation between different data representations may be built and the translations may be carried out when pages are required. The other problem is that it may require close integration with programming language compilation in order to identify which data items need to be converted.

9.2.3 Exploitation of COMMOS Functionality

Another area for further work is in the exploitation of COMMOS functionality to support high-level applications. Some of them are *persistent programming languages*, *distributed databases* and *computer-supported cooperative work* (CSCW).

Persistent Programming Languages The ease of supporting distributed persistent programming has been illustrated in Chapter 7. A complete class library may be built to exploit the COMMOS architecture. It is particularly interesting to explore how to support interworking between different programming languages.

Distributed Database Systems The COMMOS architecture eliminates double paging and provides support for coherent persistent objects in dis-

tributed environments. As discussed earlier, the locking mechanism in COMMOS can be used to implement sophisticated concurrency control facilities such as two-phase locking and transactions. In order to support database systems, persistent objects should be recoverable. This can be realised by using the write ahead log or shadow paging techniques.

Computer-Supported Cooperative Work Another potential application is computer-supported cooperative work (CSCW) [Reinhard94]. CSCW software systems can be divided into four classes of features or devices: input, output, application and data. Each of them can be distributed or centralised. Centralising the data guarantees consistency but requires high traffic to the application or I/O devices. Distributed data requires less traffic for updates and is hence more desirable than the centralised scheme. COMMOS could be used to support coherent distributed data management in CSCW.

9.3 Final Word

The major contribution of this dissertation is exploring the architectural support for a flexible coherence mechanism for memory-mapped object systems. Through the COMMOS architecture, main memory has been integrated with secondary storage and local memory has been integrated with remote memory. Virtual memory is hence restructured to support distributed computing environments. Meanwhile, the system is kept open to meet different requirements for different applications without sacrificing system performance. The prototype implementation and performance measurements have shown that the architecture is practicable and feasible.

Bibliography

[Abrossimov90] V Abrossimov, M Rozier, and M Gien. **Virtual Memory Management in Chorus.** In W. Schroder-Preikschat and W. Zimmer, editors, *Progress in Distributed Operating Systems and Distributed System management: European Workshop, Berlin, FRG, April 18/19, 1989 Proceedings*, volume 433 of *Lecture Notes in Computer Science*, pages 45 – 59. Springer-Verlag, 1990. *(cited on page 80)*

[Abrossimov92] V Abrossimov, F Armand, and M I Ortega. **A Distributed Consistency Server for the CHORUS System.** In *Proceedings of SEDMS III, Symposium on Experiences with Distributed and Multiprocessor Systems*, Newport Beach, CA, March 26 – 27 1992. *(cited on page 81)*

[Accetta86] M Accetta, R Baron, D Golub, R Rashid, A Tevanian, and M Young. **Mach: A New Kernel Foundation for Unix Development.** In *Proceedings of the Summer 1986 USENIX Conference*, pages 93 – 112, July 1986. *(cited on pages 6, 21, 78)*

[Amaral92] P Amaral, C Jacquemot, and R Lea. **A Model for Persistent Shared Memory Addressing in Distributed Systems.** Technical Report CS/TR-92-52, Chorus Systems, 6 Avenue Gustave Eiffel, F-78182, Saint-Quentin-en-Yvelines, France, September 1992. *(cited on page 75)*

[Ananthanarayanan92a] R Ananthanarayanan, M Ahamad, and R J LeBlanc. **Application Specific Coherence Control for High Performance Distributed Shared Memory.** In *Proceedings of SEDMS III, Symposium on Experiences with Distributed and Multiprocessor Systems*, Newport Beach, CA, March 26 – 27 1992. *(cited on page 82)*

[Ananthanarayanan92b] R Ananthanarayanan, S Menon, A Mohindra, and U Ramachandran. **Experience in Integrating Distributed Shared Memory with Virtual Memory Management.** *ACM Operating System Review*, **26**(3):4 – 26, July 1992. *(cited on page 82)*

[Atkinson83] M P Atkinson, P J Bailey, K J Chisholm, P W Cockshott, and R Morrison. **An Approach to Persistent Programming.** *The Computer Journal*, **26**(4):360 – 365, 1983. *(cited on pages 11, 12, 65)*

[Bacon93] Jean Bacon. **Concurrent Systems: An Integrated Approach To Operating Systems, Database, And Distributed Systems.** International Computer Science Series. Wokingham: Addison-Wesley, 1993. *(cited on pages 4, 5, 9, 20, 23, 25, 28)*

[Bal93] H E Bal and M F Kaashoek. **Object Distribution in Orca Using Compile-Time and Run-Time Techniques.** In *Proceedings of 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*, pages 162 – 177, Washington, DC, USA, 26 Sep – 1 Oct 1993. ACM Press. *(cited on page 13)*

[Balter93] R Balter, P Y Chevalier, A Freyssinet, D Hagimont, S Lacourte, and X Rousset de Pina. **Is the Microkernel Technology Well Suited for the Support of Object-Oriented Operating Systems: the Guide Experience.** In *Workshop on Microkernel and Other Kernel Architectures*, September 1993. *(cited on page 86)*

[Birrell84] A D Birrell and B J Nelson. **Implementing Remote Procedure Calls.** *ACM Transactions on Computer Systems*, **2**(1):39 – 59, January 1984. *(cited on page 12)*

[Bisiani88] R Bisiani and A Forin. **Multilanguage Parallel Programming of Heterogeneous Machines.** *IEEE Transactions on Computers*, **37**(8):930 – 945, August 1988. *(cited on page 12)*

[Bliot90] J Bliot and B Moss. **Design of the Mneme Persistent Object Store.** *ACM Transactions on Information Systems*, **8**(2):103 – 139, April 1990. *(cited on page 75)*

[Blott94] S Blott, H Kaufmann, L Relly, and H J Schek. **Buffering Long Externally-Defined Objects.** In Malcolm Atkinson, Veronique Benzaken, and David Maier, editors, *Proceedings of Sixth International Workshop on Persistent Object Systems*, pages 40 – 53, Tarascon, France, 5 – 9 September 1994. *(cited on page 65)*

[Bricker91] A Bricker, M Gien, M Guillemont, J Lipkis, D Orr, and M Rozier. **A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility.** Technical Report CS/TR-91-7, Chorus systèmes, February 1991. *(cited on page 6)*

[Cahill93] V Cahill, R Balter, N Harris, and X R de Pina. **The Comandos Distributed Application Platform.** Technical report, ESPRIT, January 1993. *(cited on pages 66, 84, 85)*

[Campbell93] R H Campbell, N Islam, D Raila, and P Madany. **Designing and Implementing Choices: An Object-Oriented System in C++.** *Communications of the ACM*, **36**(9):117 – 126, September 1993. *(cited on page 82)*

[Carriero89] N Carriero and D Gelernter. **Linda in Context.** *Communications of the ACM*, **32**(4):444 – 458, April 1989. *(cited on page 13)*

[Chang89] Ellis E Chang. **Effective Clustering and Buffering in an Object-Oriented DBMS.** PhD Thesis, Dept of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720, 1989. Also available as Technical Report UCB/CSD 89/515. *(cited on page 72)*

[Chase92] J S Chase, H M Levy, E D Lazowska, and M Baker-Harvey. **Lightweight Shared Objects in a 64-Bit Operating System.** In Andreas Paepcke, editor, *Proceedings of 7th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92)*, pages 397 – 413, Vancouver, British Columbia, Canada, 18 – 22 Oct 1992 1992. ACM Press. *(cited on pages 75, 87)*

[Chase93] J S Chase, H M Levy, M J Feeley, and E D Lazowska. **Sharing and Protection in a Single Address Space Operating System.** Technical Report 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, April 1993. *(cited on pages 20, 88)*

[Cheriton88a] D Cheriton. **The V Distributed System.** *Communications of the ACM*, **31**(3):314 – 333, March 1988. *(cited on pages 6, 21, 81)*

[Cheriton88b] D R Cheriton. **The Unified Management of Memory in the V Distributed System.** Technical Report STAN-CS-88-1192, Department of Computer Science, Stanford University, Stanford, California 94305, USA, August 1988. *(cited on page 81)*

[Chew92] K-M Chew and A Silberschatz. **Toward Operating System Support for Recoverable-Persistent Main Memory Database Systems.** Technical Report TR-92-05, Department of Computer Science, University of Texas at Austin, Austin, TX 78712, February 1992. *(cited on page 9)*

[Coulouris94] George Coulouris, Jean Dollimore, and Tim Kindberg. **Distributed Systems Concepts and Design.** Addison-Wesley Publishing Company, Second edition, 1994. *(cited on pages 5, 6, 12, 15, 16, 32, 56)*

[Dasgupta91] P Dasgupta, R J LeBlanc Jr, M Ahamad, and U Ramachandran. **The Clouds Distributed Operating System.** *IEEE Computer*, **24**(11):34 – 44, November 1991. *(cited on page 81)*

[Dean92] R W Dean and F Armand. **Data Movement in Kernelized Systems.** School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, 1992. *(cited on page 9)*

[Dharanikota94] S Dharanikota, K Maly, and C M Overstreet. **Performance Evaluation of TCP(UDP)/IP over ATM Networks.** Technical Reports TR-94-23, Computer Science Department, Old Dominion University, Norfolk VA 23529-0162, 1994. *(cited on page 32)*

[Dixon91] Michael Joseph Dixon. **System Support for Multi-Service Traffic.** PhD Thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, September 1991. Also available as University of Cambridge Computer Laboratory Technical Report No. 245. *(cited on page 28)*

[Ferreira94] P Ferreira and M Shapiro. **Garbage Collection of Persistent Objects in Distributed Shared Memory.** In Malcolm Atkinson, Veronique Benzaken, and David Maier, editors, *Proceedings of Sixth International Workshop on Persistent Object Systems*, pages 176 – 191, Tarascon, France, 5 – 9 September 1994. *(cited on page 65)*

[Forin88] A Forin, J Barrera, M Young, and R Rashid. **Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.** Technical Report CMU-CS-88-165, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, August 1988. *(cited on pages 19, 79)*

[Fotheringham61] J Fotheringham. **Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store.** *Communications of the ACM*, 4:435 – 436, October 1961. *(cited on page 7)*

[Gien91] M Gien. **Next Generation Operating Systems Architecture.** In A Karshmer and J Nehmer, editors, *Operating Systems of the 90s and Beyond – International Workshop*, number 563 in Lecture Notes in Computer Science, pages 227 – 232. Springer-Verlag, Dagstuhl Castle, Germany, July 1991. *(cited on page 5)*

[Goldberg74] R P Goldberg and R Hassinger. **The Double Paging Anomaly.** In *Proceedings of AFIPS National Computer Conference*, volume 43, pages 195 – 199, Chicago, Illinois, May 6 – 11 1974. AFIPS Press. *(cited on page 10)*

[Goscinski91] Andrzej Goscinski. **Distributed Operating Systems — The Logical Design.** Addison-Wesley Publishing Company, 1991. *(cited on page 34)*

[Gould91] A J Gould. **Implementation of the IP and UDP Communications Protocols on the Experimental Operating System Wanda.** Computer Science Tripos, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, 1991. *(cited on page 48)*

[Gourhant92] Y Gourhant, S Louboutin, V Cahill, A Condon, G Starovic, and B Tangney. **Dynamic Clustering in an Object-Oriented Distributed System.** In *Proceeding of OLDA-II Workshop (Objects in Large Distributed Applications)*, Ottawa, Canada, 18 October 1992. *(cited on page 72)*

[Group92] Arjuna Research Group. **The Arjuna System Programmer's Guide.** Technical report, Department of Computer Science, Computing Laboratory, University of Newcastle Upon Tyne, NE1 7RU, UK, July 1992. *(cited on page 66)*

[Hamilton84] Kenneth Graham Hamilton. **A Remote Procedure Call System.** PhD Thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, December 1984. Also available as University of Cambridge Computer Laboratory Technical Report No. 70. *(cited on page 33)*

[Hamilton93] G Hamilton and P Kougiouris. **The Spring Nucleus: A Microkernel for Objects.** In *Proceedings of USENIX Summer 1993 Technical Conference*, pages 147 – 159, Cincinnati, Ohio, USA, 21 – 25 June 1993. *(cited on pages 6, 83)*

[Harty92] K Harty and D R Cheriton. **Application-Controlled Physical Memory using External Page-Cache Management.** In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187 – 197, Boston, Massachusetts, 12 – 15 October 1992. Also as *ACM SIGPLAN Notices 27(9)*. *(cited on page 81)*

[Hayter93] Mark David Hayter. **A Workstation Architecture to Support Multimedia.** PhD Thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, November 1993. Also available as University of Cambridge Computer Laboratory Technical Report No. 319. *(cited on page 4)*

[Hemmendinger92] D Hemmendinger and C J Fleckenstein. **Architectural Support for Distributed Shared Memory.** In M C Yovits, editor, *Advances in Computers*, volume 35, pages 270 – 285. Academic Press Inc., 1992. *(cited on page 12)*

[Herlihy82] M Herlihy and B Liskov. **A Value Transmission Method for Abstract Data Types.** *ACM Transactions on Programming Languages and Systems*, 4(4):527 – 551, October 1982. *(cited on page 12)*

[Huang92] F Huang. **Current Status of Wanda Memory-Mapped Object Management.** Internal document, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, September 1992. *(cited on page 29)*

[Kane92] Gerry Kane and Joe Heinrich. **MIPS RISC Architecture: MIPS RISC Processors, Reference for the R2000, R3000, R6000, and the New R4000 Reduced Instruction Set Computer Architecture.** Prentice Hall, 1992. *(cited on page 4)*

[Kara94] M Kara, R Drew, N D Hunter, J Jackson, and P M Dew. **An Empirical Study of Video Application Performance over ATM and Ethernet Networks.** Research Report 94-17, School of Computer Studies, University of Leeds, May 1994. *(cited on page 32)*

[Khalidi93a] Y A Khalidi and M N Nelson. **The Spring Virtual Memory System.** Technical Report SMLITR-93-09, Sun Microsystems Laboratories, Inc, MTV29-112, 2550 Garcia Ave, Mountain View, CA 94043, USA, February 1993. *(cited on page 84)*

[Khalidi93b] Y A Khalidi, M Talluri, and M N Nelson. **Virtual Memory Support for Multiple Page Sizes.** Technical Report SMLITR-93-17, Sun Microsystems Laboratories, Inc, MTV29-112, 2550 Garcia Ave, Mountain View, CA 94043, USA, September 1993. *(cited on page 33)*

[Korth91] Henry F Korth and Abraham Silberschatz. **Database System Concepts.** Computer Science Series. McGraw-Hill, Inc., Second edition, 1991. *(cited on pages 9, 20)*

[Krueger93] K Krueger, D Loftesness, A Vahdat, and T Anderson. **Tools for the Development of Application-Specific Virtual Memory Management.** Technical Report UCB/CSD 93/740, University of California Computer Science Division, Berkeley, CA 94720, April 1993. *(cited on page 8)*

[Lampson71] B W Lampson. **Protection.** In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437 – 443. Princeton University, March 1971. Reprinted in Operating Systems Review, 8(1), January 1974, pp. 18 – 24. *(cited on page 24)*

[Lampson79] B W Lampson and R F Sproull. **An Open Operating System for a Single User Machine.** In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 98 – 105, 1979. *(cited on page 6)*

[Lea93] R Lea, C Jacquemot, and E Pillevesse. **COOL: System Support for Distributed Programming.** *Communications of the ACM*, **36**(9):37 – 46, September 1993. *(cited on page 85)*

[Leach83] P J Leach, P H Levine, B P Douros, J A Hamilton, D L Nelson, and B L Stumpf. **The Architecture of an Integrated Local Network.** *IEEE Journal on Selected Areas in Communications*, **1**(5):842 – 857, November 1983. *(cited on page 77)*

[Lee89] R B Lee. **Precision Architecture.** *IEEE Computer*, **22**(1):78 – 91, January 1989. *(cited on page 4)*

[Leslie93] I M Leslie, D McAuley, and S J Mullender. **Pegasus — Operating System Support for Distributed Multimedia Systems.** *ACM Operating Systems Review*, **27**(1):69 – 78, January 1993. *(cited on page 88)*

[Li86] Kai Li. **Shared Virtual Memory on Loosely Coupled Multiprocessors.** PhD Thesis, Yale University Department of Computer Science, New Haven, CT 06520, September 1986. Also available as Technical Report YALEU/DCS/RR-492. *(cited on pages 12, 13, 39, 40, 79)*

[Liedtke93] J Liedtke. **Improving IPC by Kernel Design.** In *Proceedings of 14th ACM Symposium on Operating System Principles*, pages 175 – 188, Asheville, North Carolina, 5 – 8 December 1993. *(cited on page 60)*

[Liedtke94a] J Liedtke. **Address Space Sparsity and Fine Granularity.** In *Proceedings of the Sixth ACM SIGOPS European Workshop*, Dagstuhl Castle, Wadern, Germany, 12 – 14 September 1994. *(cited on page 20)*

[Liedtke94b] J Liedtke. **Page Table Structures For Fine-Grain Virtual Memory.** Technical Report 872, German National Research Center for Computer Science (GMD), GMD I5.RS, 53754 Sankt Augustin, Germany, October 1994. *(cited on page 33)*

[Liskov93] B Liskov, M Day, and L Shrira. **Distributed Object Management in Thor.** In M Tamer Ozsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 79 – 91. Morgan Kaufmann Publishers, Inc, 1993. *(cited on page 65)*

[Lo94] Sai-Lai Lo. **A Modular and Extensible Network Storage Architecture.** PhD Thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, January 1994. Also available as University of Cambridge Computer Laboratory Technical Report No. 326. *(cited on pages 24, 54)*

[Mainetto94] G Mainetto, M Di Giacomo, and L Vinciotti. **Distributed Galileo: a Persistent Programming Language with Transactions.** In Malcolm Atkinson, Veronique Benzaken, and David Maier, editors, *Proceedings of Sixth International Workshop on Persistent Object Systems*, pages 487 – 509, Tarascon, France, 5 – 9 September 1994. *(cited on page 65)*

[Mapp91] Glenford Ezra Mapp. **An Object-Oriented Approach To Virtual Memory Management.** PhD Thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, September 1991. Also available as University of Cambridge Computer Laboratory Technical Report No. 242. *(cited on pages 21, 29, 49)*

[Morrison90] R Morrison and M P Atkinson. **Persistent Languages and Architectures.** In J Rosenberg and J L Keedy, editors, *Security and Persistence — Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 9 – 28, Bremen, West Germany, 8 – 11 May 1990. Springer-Verlag. *(cited on pages 11, 19)*

[Mullender90] S J Mullender, G van Rossum, A S Tanenbaum, R van Renesse, and J M van Staveren. **A Distributed Operating System for the 1990s.** *IEEE Computer*, **23**(5):44 – 53, May 1990. *(cited on page 6)*

[Munro94] D S Munro, R C H Connor, R Morrison, S Scheuerl, and D W Stemple. **Concurrent Shadow Paging in the Flask Architecture.** In Malcolm Atkinson, Veronique Benzaken, and David Maier, editors, *Proceedings of Sixth International Workshop on Persistent Object Systems*, pages 16 – 37, Tarascon, France, 5 – 9 September 1994. *(cited on page 65)*

[Needham91] R M Needham. **What Next? Some Speculations.** In A Karshmer and J Nehmer, editors, *Operating Systems of the 90s and Beyond – International Workshop*, number 563 in Lecture Notes in Computer Science, pages 220 – 223. Springer-Verlag, Dagstuhl Castle, Germany, July 1991. *(cited on pages 4, 5)*

[Nelson88] M N Nelson, B B Welch, and J K Ousterhout. **Caching in the Sprite Network File System.** *ACM Transactions on Computer Systems*, **6**(1):134 – 154, February 1988. *(cited on pages 15, 16)*

[Nelson93] M N Nelson, Y A Kalidi, and P W Madany. **The Spring File System.** Technical Report SMLITR-93-10, Sun Microsystems Laboratories, Inc, MTV29-112, 2550 Garcia Ave, Mountain View, CA 94043, USA, February 1993. *(cited on page 84)*

[Nitzberg91] B Nitzberg and Virginia Lo. **Distributed Shared Memory: A Survey of Issues and Algorithms.** *IEEE Computer*, **24**(8):52 – 60, August 1991. *(cited on page 12)*

[Organick72] E I Organick. **The Multics System: An Examination of Its Structure.** The Massachusetts Institute of Technology, 1972. *(cited on page 17)*

[Ozsu91] M Tamer Ozsu and Patrick Valduriez. **Principles of Distributed Database Systems.** Prentice-Hall International Inc, 1991. *(cited on page 9)*

[Rashid88] R Rashid, A Tevanian Jr, M Young, D Golub, R Baron, D Black, W J Bolosky, and J Chew. **Machine-Independent Virtual Memory Management For Paged Uniprocessor And Multiprocessor Architectures.** *IEEE Transactions on Computers*, **37**(8):896 – 908, August 1988. *(cited on page 79)*

[Reinhard94] W Reinhard, J Schweitzer, G Volksen, and M Weber. **CSCW Tools: Concepts and Architectures.** *IEEE Computer*, **27**(5):28 – 36, May 1994. *(cited on page 94)*

[Richardson93] T Richardson. **TCP & UDP/IP on Wanda.** In *ATM Document Collection 2*, pages 9–1 – 9–7. University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, February 1993. *(cited on page 48)*

[Roe92] M Roe. **IP over MSNL.** In *ATM Document Collection*. University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, February 1992. *(cited on page 48)*

[Rosenberg92] J Rosenberg. **Architectural and Operating System Support for Orthogonal Persistence.** *Computing Systems*, 5(3):305 – 335, November 1992. *(cited on page 75)*

[Rozier88] M Rozier, V Abrossimov, F Armand, I Boule, M Gien, M Guillemont, F Herrmann, C Kaiser, S Langlois, P Leonard, and W Neuhauser. **Chorus Distributed Operating Systems.** *Computing Systems*, 1(4):305 – 367, 1988. *(cited on pages 6, 21, 80)*

[Russo89] V F Russo and R H Campbell. **Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques.** In N Meyrowitz, editor, *OOPSLA'89: Proceedings of Object-Oriented Programming: Systems, Languages and Applications*, pages 267 – 278, New Orleans, Louisiana, USA, October 1 – 6 1989. ACM Press. *(cited on page 83)*

[Sane90] A Sane, K MacGregor, and R Campbell. **Distributed Virtual Memory Consistency Protocols: Design and Performance.** In *Proceedings of the Second IEEE Workshop in Experimental Distributed Systems*, pages 91 – 96, Huntsville, Alabama, USA, October 1990. *(cited on page 83)*

[Satyanarayanan90] M Satyanarayanan and E H Siegel. **Parallel Communication in a Large Distributed Environment.** *IEEE Transactions on Computers*, 39(3):328 – 348, March 1990. *(cited on page 34)*

[Satyanarayanan91] M Satyanarayanan. **RPC2 User Guide and Reference Manual.** School of Computer Science, Carnegie Mellon University, October 1991. *(cited on pages 34, 50, 51)*

[Satyanarayanan93] M Satyanarayanan. **Chapter 14: Distributed File Systems.** In Sape Mullender, editor, *Distributed Systems*, ACM Press Frontier Series, pages 353 – 383. ACM Press, Second edition, 1993. *(cited on pages 15, 16)*

[Shekita91] E Shekita and M Zwilling. **Cricket: A Mapped, Persistent Object Store.** In *Implementing Persistent Object Bases — Principles and Practice, The Fourth International Workshop on Persistent Object Systems*, pages 89 – 102, Massachusetts, USA, 1991. Morgan Kaufmann Publishers, Inc. *(cited on page 19)*

[Sites92] Richard L Sites. **Alpha Architecture Reference Manual.** Digital Press, 1992. *(cited on page 4)*

[Stonebraker81] M Stonebraker. **Operating System Support for Database Management.** *Communications of the ACM*, **24**(7):412 – 418, July 1981. *(cited on pages 9, 20)*

[Stroustrup91] Bjarne Stroustrup. **The C++ Programming Language.** Addison-Wesley Publishing Company, Second edition, 1991. *(cited on pages 67, 70)*

[Tanenbaum92] Andrew S Tanenbaum. **Modern Operating Systems.** Prentice-Hall International Editions, 1992. *(cited on pages 5, 15, 19, 25)*

[Traiger82] I L Traiger. **Virtual Memory Management for Database Systems.** *Operating Systems Review*, **16**(4):26 – 48, October 1982. *(cited on pages 9, 20)*

[Vaughan92] F Vaughan, T L Basso, A Dearle, C Marlin, and C Barter. **Casper: a Cached Architecture Supporting Persistence.** *Computing Systems*, **5**(3):337 – 359, November 1992. *(cited on page 86)*

[Welch91] B Welch. **The File System Belongs in the Kernel.** In *Proceedings of the 2nd USENIX Mach Symposium*, pages 233 – 250, November 20 – 22 1991. *(cited on page 8)*

[Wilson92] P R Wilson and S V Kakkad. **Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware.** In *Proceedings of 1992 International Workshop on Object Orientation in Operating Systems*, pages 364 – 377, Paris, France, 24 – 25 September 1992. IEEE Press. *(cited on page 75)*

[Zahorjan91] J Zahorjan. **Operating Systems of the 90s and Beyond (with Emphasis on Multiprocessing).** In A Karshmer and J Nehmer, editors, *Operating Systems of the 90s and Beyond – International Workshop*, number 563 in Lecture Notes in Computer Science, pages 53 – 55. Springer-Verlag, Dagstuhl Castle, Germany, July 1991. *(cited on page 6)*

Appendix A

Public Interface

This appendix describes the public interface of the COMMOS. Each object is identified by a global name, which is a NFS path name in the prototype implementation. When an object is mapped at a network node, it is assigned a unique integer as the local name of the object, which is referred to as the *object number* or `ObjNum`. After an object is mapped into a user address space, the index of the **process map** (see 3.8 for details), which is called `vir_id`, is used as the object name local to that address space.

- `GetMyProcMap() -> virtual address`

  Maps the process map for the calling process as read-only and returns its location.

- `CreateObject(TypeName, ObjName, MapAttr, Size) -> vir_id`

  Creates a new object and returns its index in the process map. The parameter `MapAttr` indicates how the object is mapped. Table A.1 shows the possible mapping attributes.

- `DestroyObject(vir_id) -> {-1, 0}`

  Destroys an object that has been created by the process. Returns 0 if successful and -1 otherwise.

- `MapObject(TypeName, ObjName, MapAttr) -> vir_id`

  Maps an object into the caller's address space and returns the index in the process map.

- `MapFragmentObject(TypeName, ObjName, Asid, Size, MapAttr, StartAddr, StorObjOffset, DataSeg) -> vir_id`

Attribute	*Description*
MAP_TYPETEXT MAP_TYPEDATA	Specifies the type of the object being mapped as code or data.
MAP_GROWUP MAP_GROWDOWN MAP_GROWNOT	Indicates whether the object is allowed to grow and the direction in which the object is expected to grow.
MAP_READONLY MAP_READWRITE	Specifies the type of access, read only or read/write, which is desired.
MAP_ZEROFILL	Indicates that the object consists of a zero-filled region.
MAP_SAVE	Indicates that the object should be saved when it is no longer in use.
MAP_DISCARD	Indicates that changes made to this object must not be flushed to disk.
MAP_INPLACE	Indicates that an object must be mapped into an address space at a specified virtual address.

Table A.1: Characteristics for Mapping Objects

Maps part of a secondary storage object into an address space at a specific virtual address. This is used when objects are stored in a composite manner such as an executable file which contains both code and data persistent objects. `Asid` indicates the address space into which the object is mapped, `StartAddr` specifies the starting address for the object in the address space, `StorObjOffset` suggests where the object starts in the corresponding secondary storage object. The `DataSeg` parameter allows the caller to associate another memory object from which data can be loaded so it is possible to load data from another object.

- `UnmapObject(vir_id, length) -> {-1, 0}`

  Unmaps an object from the caller's address space. Returns 0 if successful and -1 otherwise.

- `FlushObject(vir_id, length) -> {-1, 0}`

  Flushes the modifications to the object back to the secondary storage. Returns 0 if everything goes well, otherwise returns -1.

- `DiscardObject(vir_id) -> {-1, 0}`

  Prevents all modifications to an object made on the local machine from being written to the backing store. Returns 0 if successful or -1 otherwise.

- `ExtendObject(vir_id, new_size) -> new size of the object`

Increases or shrinks the size of an object and returns the new size of the object.

- `LockInMem(vir_id) -> {-1, 0 }`

  Disables paging on an object. Returns 0 if successful and -1 otherwise.

- `UnlockInMem(vir_id) -> {-1, 0}`

  Enables paging on an object. Returns 0 if successful and -1 otherwise.

- `AcquireLock(vir_id, offset, length, rw, mode) -> {-1, 0}`

  Acquires a lock for an object fragment. The parameter `rw` indicates a *read* or *write* lock is required and `mode` specifies whether the caller is blocked if the lock is not available immediately. Return value 0 indicates that the lock is granted and -1 means that the lock acquirement is denied.

- `ReleaseLock(vir_id, offset, length, rw) -> {-1, 0}`

  Releases a lock for an object fragment. Returns 0 if successful and -1 otherwise.

- `GetObjTypeNum(TypeName) -> object type number`

  Gets the system assigned object type number.

Appendix B

VMM and POM Interaction

POMs interact with users via an **object table**, which resides in the kernel but may be mapped into the address spaces of POMs. Activated objects of a given type are also linked together in a **type list**. POMs communicate with the virtual memory management layer through a VMM interface while the user processes invoke the POMs by triggering events.

VMM Interface

- `MapFreePageTable() -> virtual address`

  Maps the free page table into the address space of the POM or the CoherSvr and returns the location.

- `RegObjHandler(TypeInfo) -> system type number`

  Registers with the system that the calling POM is willing to be the object manager of the object type. The parameter `TypeInfo` is the type information which includes the type name, the POM's process id, the paging algorithm and the coherence protocol will be used. Returns the system assigned type number.

- `ConstructPageTable(ObjNum, Size) -> {-1, 0}`

  Constructs a page table for an object. This is done after the POM has obtained the size of the object from the network storage server. Returns 0 if everything goes well, -1 otherwise.

- `Investigate(ObjNum, ObjState)`

Obtains information about an object. The `ObjState` parameter points to a structure that contains the required information for the POM to handle the page fault from its own address space.

- `ReturnResult(ObjNum, ObjState) -> {-1, 0, 1}`

  Invoked by the POM after an event on an object has been processed. It updates the object state and wakes up threads waiting for the event.

Events Handled by POMs

`PAGETABLE_PENDING:`	This event occurs when an object is first mapped into main memory. Responding to it, the POM constructs the page table for the object.
`PAGEFAULT_PENDING:`	Signals that a page fault has occurred on an object.
`WRITEACCESS_PENDING:`	Signals that the faulting thread attempts to write to a read only page. It is used by the POM to get the ownership for the page.
`WRITETHROUGH_PENDING:`	Signals that a modified page has to be written through to the copy set.
`SWAP_PENDING:`	Pages related to the object mapping have been removed from the resident paging set and need to be placed on a swap device.
`FLUSH_PENDING:`	Indicates that modified pages of an object must be written out to the secondary storage.
`WRITEOUT_PENDING:`	All the relevant user processes have finished using the object and the modifications need to be written back to the secondary storage.
`REMOVAL_PENDING:`	All operations on an object have been completed and thus the object should be removed from the main memory.

Interaction Protocol

A process invokes the POM by posting an event and then suspends itself until the POM replies. Figure B.1 shows the state machine of this interaction protocol.

Suppose that the call `MapObject` is invoked by a user. The object type list

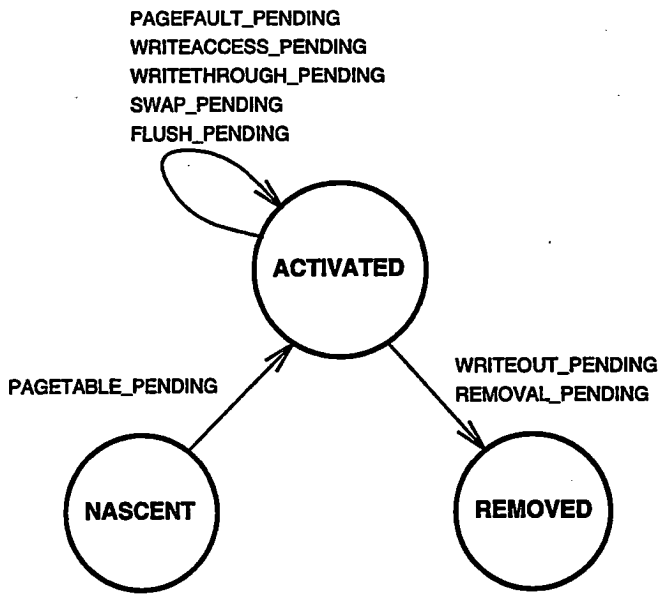


Figure B.1: VMM and POM Interaction Protocol

is searched first to see if there exists a corresponding entry. If not, a new object data structure is created with its initial state set to **NASCENT**. A `PAGETABLE_PENDING` event is posted to the POM which gets the object size from the network storage server and constructs the page table for the object.

The object state is changed into **ACTIVATED** after the page table has been constructed and the object can be mapped into user process address spaces. Certain events such as `PAGEFAULT_PENDING`, `WRITEACCESS_PENDING`, `WRITETHROUGH_PENDING`, `SWAP_PENDING` and `FLUSH_PENDING` do not cause a state change.

When the user has finished using the object, `UnmapObject` is called. If no other thread in that user's address space is using the object, it is unmapped from the user's address space. If no other address space has the object mapped, and it is mapped read/write with no discard indicator being set, a `WRITEOUT_PENDING` event occurs and the POM writes the modifications back to the storage server.

If the discard bit on an object is set, or the object is mapped read only, a `REMOVAL_PENDING` event is set. This causes the POM to remove the object.

Appendix C

Coherence Protocols

C.1 Write-Invalidate Protocols

C.1.1 Centralised-Control Protocol

Persistent Object Manager

```
Read Fault Handler:
  BEGIN
    locate the fault: x.p (page p of object x);
    ask the CoherMgr for read access and a copy of x.p;
  END;

Write Fault Handler:
  BEGIN
    locate the fault: x.p;
    ask the CoherMgr for write permission and a copy of x.p;
    set the ownership indicator of x.p;
  END;

Write Access Fault Handler:
  BEGIN
    locate the fault: x.p;
    ask the CoherMgr for write permission to x.p;
    set the ownership indicator of x.p;
    write x.p to storage server;
```

```
  END;
```

Coherence Manager

```
Read Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
    ELSE
      ask the owner for a copy of x.p;
    END;
    include the request node to the copy set of x.p;
    send a copy of x.p to the request node;
  END;

Write Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
    ELSE
      ask the owner for write permission and a copy of x.p;
      write x.p to the storage server;
    END;
    invalidate the copy set;
    clear the copy set information and the ownership indicator;
    set owner as the request node;
    grant write permission and a copy of x.p to the request node;
END;

Write Access Fault Manager:
  BEGIN
    IF the request node is in the copy set THEN
      invalidate the owner and the copy set except the request node;
      clear the copy set information;
      set owner as the request node;
      grant write permission of x.p to the request node;
    ELSE
      the request is rejected;
    END;
  END;
```

Coherence Server

```
Read Fault Server:
  BEGIN
    IF I am the owner THEN
      send a copy of x.p to the CoherMgr;
    ELSE
      the request is rejected;
    END;
  END;

Write Fault Server:
  BEGIN
    IF I am the owner THEN
      clear ownership indicator of x.p;
      send a copy of x.p to the CoherMgr;
      invalidate the local cache of x.p;
    ELSE
      the request is rejected;
    END;
  END;
```

There is no *write access server*. This is because when there exists a read copy in the request node, the owner of the page should not be in the middle of a write operation. The coherence manager can hence grant the write permission to the request node after invalidating the caches on the owner and the copy set except the request node.

C.1.2 Distributed-Control Protocol

Persistent Object Manager

```
Read Fault Handler:
  BEGIN
    locate the fault: x.p (page p of object x);
    IF I have probable owner information THEN
      ask the probable owner for read access and a copy of x.p;
    ELSE
      ask the CoherMgr for read access and a copy of x.p;
```

```
    END;
    wait until x.p has been arrived;
    record the responder as the probable owner of x.p;
  END;

Write Fault Handler:
  BEGIN
    locate the fault: x.p;
    IF I have probable owner information THEN
      ask the probable owner for write permission and a copy of x.p;
    ELSE
      ask the CoherMgr for write permission and a copy of x.p;
    END;
    wait until write permission and a copy of x.p has been granted;
    set the ownership indicator of x.p;
  END;

Write Access Fault Handler:
  BEGIN
    locate the fault x.p;
    IF I am the owner THEN
      invalidate the copy set;
      clear the copy set;
    ELSE
      ask the probable owner for write permission to x.p;
      wait until write permission has bee granted;
      set the ownership indicator of x.p;
    END;
  END;
```

Coherence Manager

```
Read Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
      include the request node to the copy set of x.p;
      send the copy of x.p to the request node;
    ELSE
      forward the request to the probable owner;
```

```
    END;
  END;

Write Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
      invalidate the copy set;
      clear the copy set information and the ownership indicator;
      grant the write permission and a copy of x.p to the request node;
    ELSE
      forward the request to the probable owner;
    END;
    set probable owner as the request node;
  END;

Write Access Fault Manager:
  BEGIN
    IF I am the owner THEN
      invalidate the copy set except the request node;
      clear the copy set information and the ownership indicator;
      set probable owner as the request node;
      grant the write permission of x.p to the request node;
    ELSE
      the request is rejected;
    END;
  END;
```

Coherence Server

```
Read Fault Server:
  BEGIN
    IF I am the owner THEN
      include the request node into the copy set of x.p;
      send a copy of x.p to the request node;
    ELSE
      forward the request to the probable owner;
    END;
  END;
```

```
Write Fault Server:
  BEGIN
    IF I am the owner THEN
      write x.p to the storage server;
      invalidate the copy set and the local copy
      clear the copy set information and the ownership indicator;
      grant write permission and a copy of x.p to the request node;
    ELSE
      forward the request to the probable owner;
    END;
    set probable owner as the request node;
  END;

Write Access Fault Server:
  BEGIN
    IF I am the owner THEN
      write x.p to the storage server;
      invalidate the local copy of x.p
          and the copy set except the request node;
      clear the copy set information and ownership indicator;
      set probable owner as the request node;
      grant write permission to the request node;
    ELSE
      the request is rejected;
    END;
  END;
```

C.2 Write-Update Protocol

Persistent Object Manager

```
Read Fault Handler:
  BEGIN
    locate the fault x.p (page p of the object x);
    ask the CoherMgr for a copy of x.p;
    wait until a copy of x.p has arrived;
    record the responder as the owner of x.p;
  END;
```

```
Write Fault Handler:
  BEGIN
    locate the fault x.p;
    ask the CoherMgr for write permission and a copy of x.p;
    wait until the copy set and a copy of x.p has arrived;
    set the ownership indicator of x.p;
    include the responder into the copy set;
  END;

Write Access Fault Handler:
  BEGIN
    locate the fault x.p;
    ask the owner for write permission of x.p;
    wait until the copy set has arrived;
    set the ownership indicator of x.p;
    include the responder into the copy set;
    exclude myself from the copy set;
  END;
```

Coherence Manager

```
Read Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
      include the request node to the copy set of x.p;
      send the copy of x.p to the request node;
    ELSE
      forward the request to the owner of x.p;
    END;
  END;

Write Fault Manager:
  BEGIN
    IF I am the owner THEN
      get a copy of x.p from the storage server;
      record the request node as the owner of x.p;
      send the copy set information and the copy of x.p to
           the request node;
      reset the ownership indicator and copy set information;
```

```
    ELSE
      forward the request to the owner of x.p;
    END;
  END;

Write Access Fault Manager:
  BEGIN
    IF I am the owner THEN
      record the request node as the owner of x.p;
      send the copy set information to the request node;
      reset the ownership indicator and copy set information;
    ELSE
      forward the request to the owner of x.p;
    END;
  END;
```

Coherence Server

```
Read Fault Server:
  BEGIN
    IF I am the owner THEN
      include the request node to the copy set of x.p;
      send a copy of x.p to the request node;
    ELSE
      forward the request to the owner of x.p;
    END;
  END;

Write Fault Server:
  BEGIN
    IF I am the owner THEN
      record the request node as the owner of x.p;
      send the copy set information and a copy of x.p to
           the request node;
      reset the ownership indicator and copy set information;
    ELSE
      forward the request to the owner of x.p;
    END;
  END;
```

```
Write Access Fault Server:
  BEGIN
    IF I am the owner THEN
      record the request node as the owner of x.p;
      send the copy set information to the request node;
      reset the ownership indicator and copy set information;
    ELSE
      forward the request to the owner of x.p;
    END;
  END;
```

Each time when the CoherMgr or a CoherSvr receives an update, the owner information of the page is updated if it has been changed.

Appendix D

Coherence Manager

The CoherMgr is a well-known server and acts as a coordinator for CoherSvrs. It maintains an **opened object table** to keep track of the activated objects managed by the CoherMgr. This appendix describes the RPC interface to the CoherMgr. In the following description, `IN` indicates an input parameter, `OUT` indicates an output parameter and `INOUT` indicates an input/output parameter. Except where explicitly specified otherwise, the return code `SUCCESS` indicates that everything goes well and `FAIL` implies that the CoherMgr is not reachable (crashing or network partitioning). The return code `FORWARDED` indicates that the request has been forwarded to the (probable) owner of the object page. When a POM thread on the request node gets this return code, it blocks itself and will be woken up with the object page available or time out.

- `CoherMgr_OpenObject(IN ObjName, IN CoherProt, OUT ObjSize) -> {SUCCESS, ERR_PROT, NOT_FOUND, FAIL}`

  This is called by a POM which attempts to map an object. If the object has not been opened, it asks the storage server for the object size and creates a new entry in the opened object table. Otherwise, it increases the reference count of the object. It then returns the object size. The first node which opens the object sets the coherence protocol to be used for the object. The return code `ERR_PROT` implies that the object has been opened with a weaker coherence protocol. The return code `NOT_FOUND` indicates that the object is not found.

- `CoherMgr_FlushObject(IN ObjName, IN CoherProt, IN ObjSize) -> {SUCCESS, FAIL}`

  This is called by a POM when a non-persistent object is made persistent or the modifications to a persistent object need to be written back to the

backing store. If there is no entry in the opened object table for this object, which means a non-persistent object is being made persistent, it creates one. In any case, it updates the object size.

- `CoherMgr_CloseObject(IN ObjName) -> {SUCCESS, FAIL}`

  This is called by a POM when an object has fallen out of use on that node. It reclaims the ownership of the object pages if any, removes the caller's CoherSvr from the copy set of the object, and decreases the reference count of the object. When the reference count becomes 0, it removes the entry from the opened object table. The return code `FAIL` implies that the CoherMgr is not reachable or the object has not been opened.

- `CoherMgr_OwnerInform(IN ObjName, IN BlockNum, IN NewOwner) -> {SUCCESS, FAIL}`

  Updates the owner information of an object page.

- `CoherMgr_GetPageForRead(IN ReqNode, IN ObjName, IN BlockNum, OUT data, OUT size) -> {SATISFIED, FORWARDED, FAIL}`

  Gets an object page for a read fault.

  If the CoherMgr is the owner, for all protocols, it includes the `ReqNode` into the copy set and returns the data to the `ReqNode`.

  If not the owner, in the case of the write-invalidate protocols, with centralised-control, it includes the `ReqNode` into the copy set and gets the data from the owner. If it is distributed-control, the request is forwarded to the probable owner. In the write-update protocol, the request is redirected to the owner.

  The return value `SATISFIED` means that the page data has been returned in the `data` parameter. `FORWARDED` indicates that the request has been forwarded to the (probable) owner.

- `CoherMgr_GetPageForWrite(IN ReqNode, IN ObjName, IN BlockNum, OUT data, OUT size, OUT CopySet) -> {SATISFIED, FORWARDED, FAIL}`

  Gets an object page for a write fault.

  In the write-invalidate protocols, if the CoherMgr is the owner of the object page, it informs the CoherSvrs in the copy set to invalidate their caches, clears the copy set information and the ownership indicator, sets the (probable) owner as the `ReqNode` and then grants write permission and a copy of the page data to the `ReqNode`. Otherwise, if the protocol is centralised-control, it contacts the owner to revoke the ownership and get the up-to-date contents of the page then writes the page data to the

backing store before other actions; if the protocol is distributed-control, the request is forwarded to the probable owner and the probable owner information is changed as the ReqNode.

In the write-update protocol, if the CoherMgr was the owner, the owner of the page is changed to be the ReqNode and a copy of the page and the copy set information are returned to the ReqNode, the ownership indicator and copy set information in the CoherMgr is reset. Otherwise, the request is forwarded to the owner.

- `CoherMgr_GetWriteAccess(IN ReqNode, IN ObjName, IN BlockNum, OUT CopySet) -> {SATISFIED, REJECTED, FORWARDED, FAIL}`

  Gets write access permission to an object page.

  In the write-invalidate protocols, with centralised-control, if the ReqNode is in the copy set, informs the owner and all the CoherSvrs in the copy set except the ReqNode to invalidate their caches, clears the copy set information and set the owner as the ReqNode, then the write permission is granted to the ReqNode. Otherwise, the request is rejected. If the protocol is distributed-control, in the case that CoherMgr is the owner of the object page, it invalidates the caches in the copy set except the ReqNode, clears the copy set information and the ownership indicator, set the probable owner as the ReqNode and grants write permission to the ReqNode. Otherwise it rejects the request. The return value REJECTED indicates that the owner has changed and hence the request node has to generate a write fault to get the up-to-date contents of the page.

  In the write-update protocol, if the CoherMgr is the owner, it records the ReqNode as the owner, returns the copy set information to the ReqNode, resets the ownership indicator and the copy set information. Otherwise, the request is forwarded to the owner.

- `CoherMgr_WritePage(IN ObjName, IN BlockNum, IN data, IN length, OUT written_size) -> {WRITTEN, NOT_OWNER, FAIL}`

  Writes an object page back to the storage server. It checks whether the caller is the owner of the page before writing. The return value WRITTEN indicates that the data has been successfully written. NOT_OWNER means that the caller is not the owner of the page, hence nothing has been written.

Appendix E

Coherence Server

The interface to the CoherSvr is presented in this appendix. In the following description, `IN` indicates an input parameter, `OUT` indicates an output parameter and `INOUT` indicates an input/output parameter. Except where explicitly specified otherwise, the return code `SUCCESS` indicates that everything goes well and `FAIL` implies that the server is not reachable (crashing or network partitioning). The return code `INPROGRESS` indicates that the remote server is processing the request and a call back will be issued later to satisfy the request. The calling POM thread will block itself and will be waken up with the object page available or time out. The return code `FORWARDED` indicates that the request has been forwarded to the (probable) owner of the object page. When a POM thread on the request node gets this return code, it blocks itself and will be woken up with the object page available or time out.

- `CoherSvr_OwnerInform(IN ObjName, IN BlockNum, IN NewOwner) -> {SUCCESS, FAIL}`

  Invalidates the local cache of an object page. If the coherence protocol is distributed-control write-invalidate, updates the probable owner information in the object table.

- `CoherSvr_GetPageForRead(IN ReqNode, IN ObjName, IN BlockNum, OUT data, OUT size) -> {SATISFIED, INPROGRESS, FORWARDED, FAIL}`

  Gets an object page for a read fault.

  In the write-invalidate protocols, with centralised-control, sends the page data to the CoherMgr. In the case of distributed-control, if the local CoherSvr is the owner of the object page, includes the `ReqNode` into the copy set of the page, if the caller is the `ReqNode`, returns a copy of the

page data, otherwise, sends a copy of the page data to the ReqNode by calling CoherSvr_SetPageForRead on the ReqNode. If the local CoherSvr is not the owner, the request is forwarded to the probable owner.

In the write-update protocol, if the CoherSvr is the owner, includes the ReqNode into the copy set, returns the page data if the caller is the ReqNode or otherwise invokes the CoherSvr_SetPageForRead on the ReqNode to satisfy the fault. If the CoherSvr is not the owner, the request is forwarded to the owner.

The return value SATISFIED means that the data has been returned as the data parameter. INPROGRESS indicates that the CoherSvr_SetPageForRead on the request CoherSvr is going to be invoked. FORWARDED suggests that the request is forwarded to the (probable) owner.

- CoherSvr_GetPageForWrite(IN ReqNode, IN ObjName, IN BlockNum, OUT data, OUT size, OUT CopySet) -> {SATISFIED, INPROGRESS, FORWARDED, FAIL}

  Gets an object page for a write fault.

  In write-invalidate protocols, if it is centralised-control, clears the ownership indicator, then sends a copy of the page data to the CoherMgr and invalidates the local cache. In the case of distributed-control, if the local CoherSvr is the owner, writes the up-to-date contents of the page back to the storage server, informs the CoherSvrs in the copy set to invalidate their caches, invalidates the local cache, clears the copy set information and ownership indicator, then sets the probable owner as the ReqNode, if the caller is the ReqNode, returns a copy of the data directly, otherwise grants the write permission and a copy of the data to the ReqNode by calling its CoherSvr_SetPageForWrite. If the local CoherSvr is not the owner, the request is forwarded to the probable owner and the ReqNode is recorded as the new probable owner.

  In the write-update protocol, if the CoherSvr is the owner, records the ReqNode as the owner, returns the copy set information and a copy of the page data directly if the the caller is the ReqNode, otherwise, invokes the CoherSvr_SetPageForWrite on the ReqNode to satisfy the fault, and finally resets the local ownership indicator and copy set information. If the CoherSvr is not the owner, the request is forwarded to the owner.

- CoherSvr_GetWriteAccess(IN ReqNode, IN ObjName, IN BlockNum, OUT CopySet) -> {SATISFIED, INPROGRESS, REJECTED, FORWARDED, FAIL}

  Transfers the ownership of an object page to the ReqNode.

In the distributed-control write-invalidate protocol, if the CoherSvr is the owner, writes the up-to-date contents of the page to the storage server, informs the CoherSvrs in the copy set except the ReqNode to invalidate their caches, invalidates the local cache, clears the copy set information and the ownership indicator, and sets the probable owner as the ReqNode. Otherwise, the request is rejected.

In write-update protocol, if the CoherSvr is the owner, records the ReqNode as the owner, returns the copy set information directly if the caller is the ReqNode or otherwise invokes the CoherSvr_SetWriteAccess on the ReqNode to satisfy the fault, resets the local ownership indicator and the copy set information. If the CoherSvr is not the owner, the request is forwarded to the owner.

- `CoherSvr_SetPageForRead(IN ObjName, IN BlockNum, IN data, IN length) -> {SUCCESS, NOT_CACHED, FAIL}`

  Invoked by CoherSvr of the object page owner to set the up-to-date contents of an object page. The value NOT_CACHED is returned in the write-update protocol when the object is no longer in use in the callee node. On receipt of this value, the caller excludes the callee from the copy set of the page.

- `CoherSvr_SetPageForWrite(IN ObjName, IN BlockNum, IN data, IN length, IN CopySet) -> {SUCCESS, FAIL}`

  Invoked by the CoherSvr of the object page owner to set an object page and the copy set in order to satisfy a write fault.

- `CoherSvr_SetWriteAccess(IN ObjName, IN ObjNum, IN CopySet) -> {SUCCESS, FAIL}`

  Invoked by the CoherSvr of the object page owner to satisfy a write access fault.

Appendix F

Storage Server Emulator

The StorSvr manages the secondary storage for the COMMOS. It provides facilities to store and retrieve object pages. In the prototype implementation, the following RPC interface is provided for the CoherMgr, CoherSvrs, POMs and ProcSvrs.

- `StorSvr_GetSegSize(IN ObjName, OUT TextSize, OUT DataSize, OUT BssSize)`

  Gets the text, data and bss size of an executable file. Used by the ProcSvrs when a new process is being created.

- `StorSvr_GetObjSize(IN ObjName, OUT ObjSize)`

  Gets the size of an object.

- `StorSvr_ReadData(IN ObjName, IN Offset, OUT Data, OUT ReadSize)`

  Reads a block of data from a backing store object, starting at the `Offset`. The return value `ReadSize` indicates how many bytes have actually been read. Currently, the block size is 1K bytes, the same as the main memory page size.

- `StorSvr_AppendData(IN ObjName, IN Data, IN WriteLength, OUT WrittenSize)`

  Appends data to the end of a backing store object. The argument `WriteLength` indicates how many bytes are intended to be written and `WrittenSize` returns how many bytes have been written.

- `StorSvr_OffWriteData(IN ObjName, IN Offset, IN Data, IN WriteLength, OUT WrittenSize)`

Overwrites data to a backing store object, starting at the `Offset`. The argument `WriteLength` indicates how many bytes are intended to be written and `WrittenSize` returns how many bytes have been written.

Appendix G

Performance Measurements

This appendix gives the results of the performance measurements. A summary and discussions of these results were presented in Chapter 6.

The hardware configuration for the performance measurements is shown in table G.1. The first column gives the machine names, the second column

Machine	*CPU Speed (MHz)*	*Memory (MB)*	*Off-Board Memory (MB)*
lamprey	20	4	8
lumpfish	20	4	8
piranha	25	4	4
shark	25	4	–

Table G.1: Hardware Configuration for Performance Measurements

shows the CPU speed and the third one describes size of the on board memory while the fourth column exhibits the configured off-board memory. During the course of the performance measurements, *shark* malfunctioned and hence appears only in the first section.

G.1 RPC Performance

This section presents the various aspects of the performance of the RPC2 system. All the RPC calls measured are null calls.

Machine Type	*Simple RPC (ms)*	*MultiRPC (ms)*
20MHz+12MB	11.0	11.6
25MHz+8MB	9.1	9.6
25MHz+4MB	7.8	–

Table G.2: Simple RPC vs MultiRPC for One Local Recipient

Source	*Destination*	*Simple RPC (ms)*	*MultiRPC (ms)*
20MHz+12MB	20MHz+12MB	16.8	17.2
20MHz+12MB	25MHz+8MB	15.7	16.2
20MHz+12MB	25MHz+4MB	14.3	15.0
25MHz+8MB	20MHz+12MB	15.7	16.0
25MHz+8MB	25MHz+4MB	13.1	13.6
25MHz+4MB	20MHz+12MB	14.4	15.0
25MHz+4MB	25MHz+8MB	12.9	13.7

Table G.3: Simple RPC vs MultiRPC for One Remote Recipient

RPC and MultiRPC on Wanda

Performance of the RPC2 system on Wanda machines are given first. Table G.2 compares the performance of an RPC call on the same machine using simple RPC and MultiRPC. The real remote RPC calls to single destination using simple RPC and MultiRPC are compared in table G.3. Remote communications to two and three destinations using MultiRPC are given in table G.4 and table G.5.

Cross-Architecture RPC and MultiRPC

Since the CoherMgr and the StorSvr in the prototype implementation run on a DEC 3100 workstation. The performance of RPC calls cross these two architectures are also measured. The results about simple RPC are shown in table G.6 and those about MultiRPC are given in table G.7.

Source	*Destinations*	*MultiRPC (ms)*	*Average (ms)*
20MHz+12MB	20MHz+12MB, 25MHz+8MB	20.9	10.5
20MHz+12MB	20MHz+12MB, 25MHz+4MB	20.4	10.2
20MHz+12MB	25MHz+8MB, 25MHz+4MB	20.0	10.0
25MHz+8MB	20MHz+12MB, 20MHz+12MB	19.9	10.0
25MHz+8MB	20MHz+12MB, 25MHz+4MB	18.7	9.4
25MHz+4MB	20MHz+12MB, 20MHz+12MB	17.7	8.9
25MHz+4MB	20MHz+12MB, 25MHz+8MB	17.2	8.6

Table G.4: MultiRPC to Two Remote Recipients

Source	*Destinations*	*MultiRPC (ms)*	*Average (ms)*
20MHz+12MB	20MHz+12MB, 25MHz+8MB, 25MHz+4MB	24.7	8.2
25MHz+8MB	20MHz+12MB, 20MHz+12MB, 25MHz+4MB	22.0	7.3
25MHz+4MB	20MHz+12MB, 20MHz+12MB, 25MHz+8MB	19.8	6.6

Table G.5: MultiRPC to Three Remote Recipients

Source	*Destination*	*RPC Time (ms)*
20MHz+12MB	DECStation	13.7
25MHz+8MB	DECStation	12.3
25MHz+4MB	DECStation	11.4
DECStation	20MHz+12MB	12.1
DECStation	25MHz+8MB	14.6
DECStation	25MHz+4MB	13.5

Table G.6: Simple RPC between Wanda and Ultrix on a DECStation

Source	*Destinations*	*MultiRPC (ms)*	*Average (ms)*
DECStation	20MHz+12MB	13.0	13.0
DECStation	25MHz+8MB	13.7	13.7
DECStation	25MHz+4MB	14.3	14.3
DECStation	20MHz+12MB, 20MHz+12MB	15.2	7.6
DECStation	20MHz+12MB, 25MHz+8MB	15.7	7.9
DECStation	20MHz+12MB, 20MHz+12MB, 25MHz+8MB	17.5	5.8

Table G.7: MultiRPC from Ultrix on DECStation to Wanda

G.2 Page Fault, Invalidation and Roundtrip IPC

Table G.8 shows some measurement results for comparison between memory-

Machine Type	*Page Fault* (ms)	*Invalidation* (ms)	*IPC* (ms)	*System Call* (ms)
20MHz+12MB	0.76	0.42	1.67	0.016
25MHz+8MB	0.60	0.35	1.33	0.013

Table G.8: Page Fault, Invalidation, Roundtrip IPC and System Call

mapping and non memory-mapping on Wanda. The second column gives overhead for a user process to process a page fault, the third one shows the time taken for the CoherSvr to invalidate a page, and the fourth presents the time for a null roundtrip IPC. The time for a null system call is also exhibited.

Faulting Node	*Time (ms)*
20MHz+12MB	31.5
25MHz+8MB	30.1

Table G.9: Fetching a Page from the StorSvr

Faulting Node	*Owner*	*Time (ms)*
20MHz+12MB	CoherMgr	32.9
25MHz+8MB	CoherMgr	31.2
20MHz+12MB	20MHz+12MB	40.6
20MHz+12MB	25MHz+8MB	38.7
25MHz+8MB	20MHz+12MB	38.8

Table G.10: Fetching a Page for Read in the Centralised-Control Protocol

G.3 Performance of the COMMOS Prototype

The performance of the COMMOS prototype is presented in three sub sections according to the coherence protocol used.

No Coherency

Table G.9 gives the time taken for the POM to fetch a page directly from the StorSvr without coherence maintenance.

Centralised-Control Protocol

Table G.10 presents the time taken for the POM to get a page to satisfied a read fault. The measurement results for the POM to get a page to satisfied a write fault is shown in table G.11.

Distributed-Control Protocol

Table G.12 gives the performance for the POM to fetch a page to serve a read

Faulting Node	*Owner*	*Copy Set*	*Time (ms)*
20MHz+12MB	CoherMgr	–	32.8
25MHz+8MB	CoherMgr	–	30.8
20MHz+12MB	CoherMgr	20MHz+12MB	56.3
20MHz+12MB	CoherMgr	25MHz+8MB	55.0
25MHz+8MB	CoherMgr	20MHz+12MB	55.5
20MHz+12MB	CoherMgr	20MHz+12MB, 25MHz+8MB	57.4
25MHz+8MB	CoherMgr	20MHz+12MB, 20MHz+12MB	56.8
20MHz+12MB	20MHz+12MB	–	54.6
20MHz+12MB	25MHz+8MB	–	52.6
25MHz+8MB	20MHz+12MB	–	53.2
20MHz+12MB	20MHz+12MB	25MHz+8MB	123.6
20MHz+12MB	25MHz+8MB	20MHz+12MB	122.9
25MHz+8MB	20MHz+12MB	20MHz+12MB	122.8

Table G.11: Fetching a Page for Write in the Centralised-Control Protocol

Faulting Node	*Owner*	*Relay Node*	*Time (ms)*
20MHz+12MB	CoherMgr	–	32.4
25MHz+8MB	CoherMgr	–	30.8
20MHz+12MB	20MHz+12MB	CoherMgr	55.3
20MHz+12MB	25MHz+8MB	CoherMgr	53.2
25MHz+8MB	20MHz+12MB	CoherMgr	53.7
20MHz+12MB	20MHz+12MB	–	31.8
20MHz+12MB	25MHz+8MB	–	30.4
25MHz+8MB	20MHz+12MB	–	30.6

Table G.12: Fetching a Page for Read in the Distributed-Control Protocol

fault in distributed-control protocol. The column *relay node* indicates the node known by the faulting node as the probable owner but it is not the real owner hence requests are forwarded. Table G.13 gives the measurement results for the POM to fetch a page to serve a write fault.

Faulting Node	*Owner*	*Relay Node*	*Copy Set*	*Time (ms)*
20MHz+12MB	CoherMgr	–	–	32.6
25MHz+8MB	CoherMgr	–	–	30.9
20MHz+12MB	20MHz+12MB	CoherMgr	–	99.0
20MHz+12MB	25MHz+8MB	CoherMgr	–	94.8
25MHz+8MB	20MHz+12MB	CoherMgr	–	96.8
20MHz+12MB	20MHz+12MB	CoherMgr	25MHz+8MB	165.7
20MHz+12MB	25MHz+8MB	CoherMgr	20MHz+12MB	163.8
25MHz+8MB	20MHz+12MB	CoherMgr	20MHz+12MB	165.8
20MHz+12MB	20MHz+12MB	–	–	85.0
20MHz+12MB	25MHz+8MB	–	–	81.1
25MHz+8MB	20MHz+12MB	–	–	84.7
20MHz+12MB	20MHz+12MB	–	25MHz+8MB	99.9
20MHz+12MB	25MHz+8MB	–	20MHz+12MB	95.0
25MHz+8MB	20MHz+12MB	–	20MHz+12MB	98.9

Table G.13: Fetching a Page for Write in the Distributed-Control Protocol