National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Distributed Persistent Objects: A Mach Approach

by
Michael L. Keith

**A thesis submitted to the Faculty of Graduate Studies
and Research in partial fulfillment of the requirements
for the degree of Master of Computer Science.**

**Ottawa-Carleton Institute of Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario.
August, 1993.**

**© Copyright 1993
Michael L. Keith.**

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-89819-4

Canada

Name MICHAEL KEITH

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0370
Slavic and East European 0372

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce-Business 0510
Finance 0511
History 0358
Labor 0346
Theory 0351
Folklore 0351
Geography 0346
Gerontology 0351
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0337
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
Criminology and Penology 0938
Demography 0631
Ethnic and Racial Studies 0628
Individual and Family Studies 0629
Industrial and Labor Relations 0630
Public and Social Welfare 0700
Social Structure and Development 0344
Theory and Methods 0709
Transportation 0999
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Immunology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0425
General 0996
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0460
Speech Pathology 0383
Toxicology 0386
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0609
Molecular 0610
Nuclear 0752
Optics 0756
Radiation 0611
Solid State 0463
Statistics 0346
Applied Sciences 0984
Applied Mechanics 0984
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Distributed Persistent Objects:

A Mach Approach

Submitted by Michael Keith, B.C.S.,
in partial fulfillment of the requirements for the degree of
Master of Computer Science.

Thesis Supervisor

Director, School of Computer Science
Carleton University

21 Sept 93

Date

ABSTRACT

The Mach operating system combines a micro-kernel with a suite of system and user servers that all subscribe to an object-based model. Of particular interest, is its memory object abstraction, and more specifically its external memory management interface that allows user-level servers the ability to manage their own external memory objects. This thesis proposes a design and implementation of a distributed persistent object server that was developed and executed in user space. It draws heavily on the external memory management subsystem, and maintains caching both at the server and client levels. It exports shareable distributed persistent objects to clients, and serves as a general purpose base for additional application servers or environments. Its performance is evaluated in the context of industrial applications with favourable results.

ACKNOWLEDGMENTS

Many thanks go out to all of those who played a part in not only the progression and completion of this thesis, but also the pleasant environment in which it was developed. This includes the administrative staff of Rosemary, Barbara, Marlene and others who no longer walk the hallowed halls.

I also wish to express my gratitude to Murali for his valued assistance, and to Peter, for all of his plugging and unplugging, as well as his unending support, without which I would not have had a Mach network to work on.

Very special thanks are owing to a dedicated supervisor, Dr. John Neilson, whose friendship, wise advice and support knew no bounds, even extending over vacations! He was instrumental in every stage of the work and preparation of this thesis, from the preliminary literature survey to the final hour of examination. For him, I insert a comma in this sentence!

Most of all, I thank my wife, Darleen, not only for the unending hours that should have been spent with her, but also for moral support above and beyond the call of duty. I am grateful to both her, and my children, Cierra and Ariana, for giving me a reason to complete this thesis. It is to them that I dedicate it, as they are the ones who paid the highest price for its completion.

Lastly I would truly be remiss if I did not thank the Natural Sciences and Engineering Research Council, as well as Dr. Murray Woodside, for their generous financial support of this research project.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Motivation	4
1.2	Why Mach?.....	7
1.3	Objectives.....	9
1.4	Organization of Thesis	11
2	MACH OVERVIEW	12
2.1	Execution Control	14
2.1.1	Tasks.....	14
2.1.2	Threads	15
2.2	Interprocess Communication.....	15
2.2.1	Ports and Messages	16
2.2.2	Object References.....	19
2.2.3	Port Sets	22
2.3	Virtual Memory Management	23
2.3.1	Virtual Address Spaces	24
2.3.2	Virtual Memory Characteristics	26
2.4	External Memory Management.....	27
2.4.1	Memory Objects and Memory Managers	28
2.4.2	Interfaces and Communication.....	31
2.4.2.1	Initialization and Termination	34
2.4.2.2	Reading and Writing	37
2.4.2.3	Consistency	38
3	DESIGN	40
3.1	Objects	40
3.1.1	Object Persistency	43
3.1.2	Distributed Objects	44

3.1.3	Object Naming	45
3.1.4	Object Sharing	47
3.1.5	Object Data Distribution.....	49
3.1.6	Object Characteristics	50
3.1.7	Object Operations.....	52
3.2	Persistent Object Server.....	53
3.2.1	Server Requests.....	54
3.2.1.1	Object Creation	54
3.2.1.2	Object Retrieval	55
3.2.2	Object Manager	57
3.2.2.1	Object Manager Clientele	57
3.2.2.2	Object Deletion.....	59
3.2.2.3	Object Flushing.....	60
3.2.3	Server Concurrency	62
3.3	Fault Tolerance.....	63
3.3.1	Object Reliability.....	64
3.3.2	Server Reliability	66
3.3.3	Client and Hardware Failures	67
3.4	Security	68
4	IMPLEMENTATION	70
4.1	Interfacing	70
4.1.1	Mach Interface Generator.....	70
4.1.2	Server and Object Interfaces.....	72
4.1.3	Input and Output Parameters	73
4.1.4	Internal Interfaces.....	74
4.2	Objects in the Server	75
4.2.1	Object State.....	76
4.2.2	Object Structure.....	77
4.2.3	Object Translation	79
4.2.4	Object Data Pages.....	79
4.2.5	Page Structure.....	83

4.3	Server Concurrency	85
4.3.1	Server Port Set.....	85
4.3.2	Number of Threads.....	86
4.3.3	Message Order	87
4.3.4	Deadlock.....	88
4.3.4.1	Remote Procedure Calls	89
4.3.4.2	Locking.....	90
4.4	Primary Operations.....	92
4.4.1	Creation.....	94
4.4.1.1	Object Port	94
4.4.1.2	Object Registration	95
4.4.2	Deletion	97
4.4.2.1	Starting Deletion.....	97
4.4.2.2	Page Availability.....	98
4.4.2.3	Patient Reclamation	99
4.4.3	Flushing	100
4.5	External Memory Management.....	103
4.5.1	Memory_object_init.....	104
4.5.2	Memory_object_terminate	105
4.5.3	Memory_object_data_request	106
4.5.4	Memory_object_data_unlock.....	107
4.5.5	Memory_object_data_return.....	108
4.5.6	Memory_object_lock_completed	109
4.5.7	Control Right Disposal.....	111
4.6	Secondary Storage.....	112
4.6.1	Super Block.....	112
4.6.2	Hnode Table.....	114
4.6.3	Hnode Structure	116
5	APPLICATIONS	118
5.1	Test Application.....	118
5.1.1	Interface.....	119
5.1.2	Opening a File.....	119
5.1.3	File I/O.....	120

5.1.4	Closing a File	122
5.1.5	Extensions	122
5.2	Memory-Mapped Files	123
5.3	Distributed Databases	124
5.4	Object-Oriented Environments	125
5.5	Distributed Persistent Shared Memory	127
6	PERFORMANCE EVALUATION	129
6.1	Testbed Environment	130
6.2	Creation/Deletion	131
6.3	Sequential I/O	132
6.4	Random Access	136
6.5	Multiple Objects	138
6.6	Concurrent Write Sharing	140
7	CONCLUSIONS	144
7.1	Mach Concerns	145
7.2	Future Work	147
7.3	Summary	148
	REFERENCES	149

LIST OF TABLES

Table 2.1	Subset of task interface.....	14
Table 2.2	Subset of thread interface	16
Table 2.3	Subset of port operations	20
Table 2.4	Identification traps	21
Table 2.5	Port set specific operations	23
Table 2.6	Virtual memory operations.....	26
Table 2.7	Call to map a memory object	29
Table 2.8	Cache object interface	32
Table 2.9	Server interface.....	33
Table 3.1	Client-Object interface.....	53
Table 3.2	Client-Server interface.....	54
Table 4.1	Object states	76
Table 4.2	Object structure.....	78
Table 4.3	Page states	82
Table 4.4	Page structure	83
Table 4.5	Super block structure.....	113
Table 4.6	Hnode structure	116
Table 6.1	File/object creation delay	132
Table 6.2	File/object deletion delay	132
Table 6.3	Sequential local read access throughput.....	133
Table 6.4	Sequential local write access throughput.....	133
Table 6.5	Sequential remote read access throughput	134
Table 6.6	Sequential remote write access throughput	135
Table 6.7	Local random access throughput.....	137
Table 6.8	Remote random access throughput	137
Table 6.9	File latencies.....	139
Table 6.10	Throughput with remote contention	141
Table 6.11	Loss of throughput	142

LIST OF FIGURES

Figure 2.1	Port and port right relationships.....	19
Figure 2.2	Remote message send	21
Figure 2.3	Task virtual address space snapshot.....	25
Figure 2.4	Object causal interaction sequence	30
Figure 2.5	Initialization/termination message sequence	36
Figure 2.6	Client read/write fault handling sequence	37
Figure 3.1	Boiled egg model of an object.....	41
Figure 3.2	Distributed object page caching	49
Figure 3.3	High-level client-server-kernel interactions.....	57
Figure 3.4	Multi-threaded request dispatch	63
Figure 4.1	Synchronous client-server messaging using MIG	73
Figure 4.2	Object finite state machine	75
Figure 4.3	Page finite state machine	81
Figure 4.4	Multi-threaded message reordering	87
Figure 4.5	Locking hierarchy.....	91
Figure 4.6	Object structure allocation.....	93
Figure 4.7	Persistent object disk partition.....	115

Chapter 1

INTRODUCTION

The emergence of Unix [Rit74] ushered in a new era of operating systems, one that extolled simplicity and portability as virtues of the timesharing operating system movement. As a revolutionary school of thought it professed that small and simple were beautiful, in direct contrast with the ever-swelling numbers of specialized and machine-specific systems of the day that seemed to be regularly and uncontrollably expanding in size and complexity. The benefits of an understandable and portable system were apparent and its popularity soared, most noticeably in academic environments, although it was not long before industry realized its worth.

When a system captures the attention of the computing community in as swift and intense a manner as did Unix, vendors want to share in the profits. In addition, with what some considered an ideal philosophy was an imperfect system, lacking many features that the growing number of vendors wanted included. Thus it began, first the factioning, and then the polluting. As performance was tantamount, the Unix kernel became a dumping ground for both standard and non-standard features alike, resulting in divergent and sometimes unrecognizable versions of what was once a small, well-defined system kernel. The resurgence of the "small is beautiful" way of thinking initiated a new flurry of activity

amongst operating system researchers, and the quest for the modular micro-kernel was reinstated.

Another focus caused the operating systems of the late 80's and 90's to follow the lead of technological advances in processor hardware, as they assumed that networks would grow in size and power. This increase would be due to faster processors that could be purchased at lower costs, allowing the number of processors to be multiplied either as interconnected computers or loosely coupled multiprocessors, sometimes called multicomputers.

Some operating systems such as Amoeba [Mul90] and Plan 9 [Pik90] adopted purist distributed strategies to harness pools of dedicated group processors, or compute servers available to all system users. In these systems a user logs on to the system from any connected terminal and is logged on to the entire system entity, not just on a single node in a network. The task of providing computational resources, secondary storage, and in some cases even volatile memory is left to the system, while the user simply expects service to be delivered in a location-transparent fashion.

Other systems have directed their efforts toward interconnecting workstation-based computers or multiprocessors to form a loosely or tightly coupled network and allow sharing of resources located at a single node, a subset of nodes, or each and every node in the network. In this way a node may or may not support a device or service and the resources available to the system equals the sum total of the resources on all the nodes. These resources may, however, be protected or even shielded from a user if he cannot log on to the node at which the resource is located, or if he does not hold sufficient privileges to do so. Although these systems were formerly referred to as network operating systems because each of the connected nodes was autonomous and user processes were strictly confined to the processor on which he or she was logged, the line between network and

distributed systems has become less defined as resources on other nodes become transparently accessible. The majority of systems developed within the last five to seven years might be subsumed by this category, including prominent systems like Mach [Ras86], CHORUS [Roz88], V [Che88] and Sprite [Ous88]. Some industry-developed systems, such as QNX [Qua92], would also be included in this group, although they are not quite as well published in research circles.

Regardless of its precise system genus virtually every system follows the trend to put the core functionality of the system, such as interprocess communication (IPC), memory management, scheduling, and process/task management in a protected and trusted micro-kernel that exports a kernel interface. All other functionality may then be implemented as servers running in user space, making calls to the kernel to accomplish communication or request system services. From this situation arises the client-server model, where servers export an interface that a client may import to make requests (usually synchronous) to the server. Since the design of these servers has tended to be more or less distributed in nature, most of these operating systems, typically including a micro-kernel and a number of tailored servers, have come to be recognized as distributed operating systems. These systems should also support the basic notions of network-transparent IPC, virtual memory, and more recently shared virtual or distributed shared memory (DSM) (see [Das88], [Fo88], [Ram88], [Li89], [Stu90] and [Moh91]).

One important server-level function that receives a deservingly high degree of attention is that of file service. As a major contributor to system and end-user performance, for good or for ill, the file server is a prime target of analysis and improvement. Attempts to determine access patterns, traffic flow, and average file lifetimes have led to studies being conducted on file systems such as the de facto standard Unix 4.2 BSD file system in [Ous85].

File systems in distributed networks have been designed to be distributed services so that files may be accessed both remotely and locally with no regard for, or even knowledge of, their location. With this new network-oriented mandate file systems have again been examined and measurements taken by [Bak91] showed that file caching at both server and client nodes produce noticeable savings. These benefits can be passed on to clients and end-users by designing file servers that will implement file caching and integrate it into access and transfer protocols.

1.1 Motivation

The future of operating systems may be an open systems approach that allows for modular, standardized, and portable system components. Specialization can then occur as server layers, taking the form of either system or user servers. These servers can be developed and tested in user space without infringing on or perturbing the running system, and then, after being completed, possibly be moved into system space for performance considerations.

When a set of system-specific functions is configured into system servers, that may manage physical or logical resources, we may refer to the whole as a subsystem, as described in [Gie91]. The concept of subsystems as being segregated from the kernel, yet running in kernel space, provides the autonomy that makes modularity possible, while still yielding performance and security that is associated with the system address space. For these reasons both CHORUS and Mach have implemented Unix as a subsystem (see [Alb91] and [Gol90]) and still maintained performance results comparing to Unix kernel operating systems. Other reasons for subsystem-based additions relate to integrability and scalability. If servers are layered up to the application level, it is within our means to run two or more different versions of the same server, such as BSD 4.2 and BSD 4.3, simultaneously. Not only can software components and applications be easily ported from

one machine to another, but the network itself can be upgraded in a seamless way that will make network renovation a menial task.

Over the course of the last decade the object-oriented trends took seed in the minds of system researchers and a number of distributed object-oriented systems were born. Some of these included Eden [Laz81], Emerald [Bla86], Argus [Lis87], Clouds [Das88], Amber [Cha89] and SOS [Sha89]. Although these systems provided the user with the same distributed services found in micro-kernel based operating systems they had object-orientedness at the top of their agenda and were not designed with open systems in mind.

An alternative was to develop a server layer or subsystem that would support, or be a basis for supporting, object-oriented programming and techniques. This tack was taken by PORTLANDIA [Por89] and COOL [Hab90], both of which understood the importance of object persistency. With a low-level server layer implementation of persistent objects available as a base, other servers could build applications ranging from object environments to file services on top, without being tightly coupled to a language or specific hardware resources. The server would subscribe to the open systems philosophy and would bring with it all of the associated benefits.

In the process of maturation distributed services offered more sophisticated functionality, network transparency, improved performance and lower communication overhead costs. Since most of the network bandwidth was consumed by the file system, efforts were concentrated on ameliorating the costs of its operation. Much of this was accomplished by caching, or keeping a local copy of more recently accessed data. Caching in distributed environments can significantly reduce the amount of network communication that would otherwise be required to implement distributed file servers and applications.

Some file systems, like Cedar's CFS [Sch85], assume the existence of local disks at each user node in order to cache distributed immutable files, while others like Locus

[Wal83], Andrew [How88], NFS [San85], and Sprite [Nel88] use replication and RAM-based caching to locally access remote data. This decision allowed clients to be diskless, or reduced disk traffic if they were not. Assumptions that main memory availability in workstation-based nodes would dramatically increase proved to be correct, and cache access times were improved in some cases by five to ten times those achieved by disk caches. Since imposed or statically-defined memory limits dedicated to caching could hamper either the file system or virtual memory system, Sprite permits the two to negotiate with each other, so that as their needs vary so also does the amount of memory each may use for their own purposes. With the confines of the cache space dynamically changing caching can acquire more memory when it requires it, to provide quick data access times and concurrent access, or data sharing, but not at the expense of other subsystems.

When data is shared in a distributed or loosely coupled architecture the consistency problem described by [Li89] arises, and can not be resolved without non-trivial communication costs. Data coherence can be maintained with varying degrees of consistency. In the NFS and Andrew file systems opening a file guarantees that the local cache will be updated with the latest version of the file ([Sat85]). Although this sequential write sharing is beneficial it is still a rather loose constraint and undetermined results may be produced when the file is opened for simultaneous reading and writing. A more intuitive and appropriate semantics for sharing would be to permit concurrent readers and writers, as does Sprite, and guarantee that a read will always yield the most recently written data. A necessary performance hit will obviously be taken in these circumstances of concurrency, but in the absence of such situations efficiency should be preserved.

At the secondary storage level, traditional data storage or file systems use data structures stored on disk that are referred to as index nodes, or inodes [Rit74]. These inodes assist in the management of the file data and must be paged in and out in order to reference different portions of the file, since typically the file may span multiple non-

contiguous blocks. The operation of paging a portion of a file into a buffer often involves first paging in one or more inode pages to find the disk block addresses of the data and then doing multiple reads, since the page may be composed of many different disk storage units, or sectors, each of which may be at a different location on the disk. Although smaller allocated units of disk storage allows higher disk utilization to be attained it also creates more fragmentation as well as a great deal of disk head contention. Amoeba [Tan90] traded off some space for higher performance when it chose to create completely contiguous files in its Bullet file server. A modest balance that would permit acceptable disk utilization levels, yet allow the unit of paging (the cache page frame size) to be read into memory in a single disk read would therefore seem to be a desirable goal. With the unit of disk storage larger, the inode table shrinks, making it possible to cache more of the table and remove extra costs associated with inode paging.

1.2 Why Mach?

With as colourful and rich an assortment of operating systems as was currently available the selection process that would eventually lead to a suitable implementation base initially appeared to be a difficult one. The process was not as arduous as was thought, however, because of specific criteria exacted by the project. In fact, the short list of finalists included only two candidates, each of which was developed on a different side of the Atlantic, but resembled the other to a surprising degree. Although both CHORUS and Mach contained much of the same functionality the latter was chosen mostly because of its understandability, availability, and profile. The features that are referred to in this section, therefore, are not indigenous to Mach alone, but exist also in CHORUS, though perhaps in dissimilar form. They are discussed in the Mach context for obvious reasons of relevance to this thesis.

The Mach kernel was designed to be at the centre of the system, with a number of system or user servers executing around it. Because the client server model is so strongly

condoned, the hooks and interfaces to servers are well-defined and services like Mach Interface Generator (MIG) [Loe92c] are available. Furthermore, more sophisticated server arrangements can be made to place trusted system server components in places of prominence, so that they have the ability to make changes to critical and protected objects.

Like most distributed operating systems Mach supports network transparency. As a key ingredient in distributed servers the Network IPC subsystem [Jul89] provides a basis for inter-nodal communication, without requiring explicit knowledge of location. This is pivotal in the design of distributed services, since a client should be able to access the service without needing to be aware of, or having to specify where the service is located.

One of the important Mach abstractions that is relevant to this discussion is that of a memory object, which is implemented through the use of ports. Just as memory-mapping may be used to transmit large amounts of data or messages between processes, so also can messages be sent to memory objects. This duality is detailed in [You87] and creates a favourable backdrop, in front of which a clean and flexible object-based system appears. Viewing all user and system entities as objects sets the stage for object-oriented design strategies and engages the actors, or tasks, in object-to-object communication.

From a user perspective, the memory object translates into a flexible virtual memory system that supplies backed regions of virtual memory. But the separation of the paging support from the kernel, seen in [Gol91], led to an exceptionally useful external memory management facility that permits a user-level data manager task to manage secondary storage memory objects. These external memory managers, or pagers, can implement the memory objects in any way they choose to, since the pager will be sent messages when input and output operations are requested by the kernel. Because the bidirectional kernel-pager message exchange protocol is asynchronously enacted a great deal of attention to synchronisation must be paid by a distributed pager. When implemented correctly, however, performance gains can be obtained.

Although it has not been, by any interpretation, universally accepted as the next generation of Unix, Mach has attracted a considerable following. Developed at Carnegie-Mellon University its greatest number of converts are still to be found in academic or research-oriented environments, with perhaps the biggest exception being the Open Software Foundation (OSF). As an association with close industry ties, its endorsement of the Mach micro-kernel has played a key role in the evolution of the Mach operating system.

Despite the occasional bugs, irregularities, and installation difficulties Mach has shown itself to be relatively stable in its current state of development. While adhering to the tenets of micro-kernel-based distributed systems, it has pursued innovative and attractive ideas that provide a suitable foundation on which a distributed persistent object server could be built.

1.3 Objectives

The goal of this thesis, then, was to exploit many of the Mach features described in the previous section to construct a distributed persistent object service. The service would be implemented for a distributed system of loosely coupled nodes and would be incarnated by a server running in user space on each node in the system. Object invocation by clients would be network-transparent so that unless the client explicitly requested it, he wouldn't be aware of the node on which the object is stored.

The objects, themselves, would be persistent, with a dedicated secondary storage partition on their nodes of residency to store their contents. As such, they would: 1) outlive the process that created them and 2) survive server and host crashes and shutdowns, being able to be operated on or invoked upon server reboot. Fair to reasonable reliability would be offered, but not to the extent of including replication strategies. Node and link failures will be detected and accounted for, with the assumption that service would continue only for those nodes that were in a running and sane condition. Servers should be

able to be rebooted after server or node failures and be fully integrated back into the object system already functioning.

The objects would be medium to coarse-grained and would be appropriate as general-purpose objects. Their usefulness would be made manifest by applications ranging from relatively small persistent message-passing to large database schemes. Of particular significance would be their application to a distributed persistent object-oriented environment and a distributed file system.

As distributed objects they would be sharable, implying that an object may be accessed by any of the nodes in the network. The semantics of object sharing would include conditions of both sequential and concurrent write sharing, meaning that reading from an object would always produce the most recently written data, regardless of whether or not the object was simultaneously open for reading and writing. No restrictions or weakening of the semantics would result from the number of concurrent readers and writers.

To reduce network traffic and improve performance in the distributed environment each object would be cached both at the resident node server and at client nodes. The caches would be completely coherent at all times, with synchronization and data integrity being supported by server-level protocols, and without any need for client management or intervention.

The goals of the persistent object project may therefore be summarised in the following list of features:

- distributed user-level servers running in user space
- medium-grained objects suitable for varied applications
- persistence

- network transparency
- sequential and concurrent write sharing of distributed objects
- caching at both server and client levels
- moderate reliability

1.4 Organization of Thesis

The remainder of this thesis discusses the implementation of such an object server, starting with a background of the Mach operating system in the ensuing chapter. Chapter three will outline the design of the server, including a description of objects and their definition in the context of this project. Chapter four will detail the actual implementation decisions that were made and reveal some of the specific features of the system. The next chapter will describe some potential applications for the system and present an example application that was implemented. Chapter six will analyze performance measurements and compare them with presently existing systems. The final chapter will summarise with a discussion of the conclusions that were reached, and discuss areas of potential enhancement and possible directions for future research.

Chapter 2

MACH OVERVIEW

The design of a distributed persistent object server in this thesis drew heavily on the structure, concepts and functionality of the Mach operating system. Since the design and implementation of the system was carried out under the Mach environment, it will most often be logical and practical to express and describe the system in Mach terms, using the abstractions and terminology of the operating system on which the server was built. An overview of the operating system is thus merited to the extent that the fundamental concepts and details explained in this chapter will aid the reader to identify with and understand the chapters to follow.

Initially, Mach was developed as a variant of BSD Unix, with binary compatibility built into the kernel. The release of version 3 demonstrated that all of the Unix functionality could be lifted out of the kernel and implemented as a system server. What remained was a Mach 3 kernel, with a native set of functions combined to meet goals of simplicity, flexibility, and extensibility.

This project was written using version 3 of Mach (MK78) as a base. The interfaces described in this and other chapters represent Mach 3 and differ slightly from the previous version 2.6. They are not, however, anticipated to change significantly in future releases. Only the relevant aspects of the Mach kernel are presented in this chapter, with the reader

being referred to [Ras86], [You87], [Loe92a], [Loe92b] and [Loe92c] for a more explicit and detailed description.

The basic Mach kernel abstractions are few in number and have remained quite constant over the course of kernel development. They may be generalized:

- **execution control.** Mach provides facilities for the creation of low overhead independently-scheduled units of computation. In addition to kernel-provided scheduling, the manipulation of program execution is under direct user control through the use of kernel calls.
- **inter-process communication.** A secure, extensible message-passing model provides communication functionality, and encourages object-based server interfaces. The kernel interface uses the IPC model to name system objects, while users may use it to address and communicate with the kernel or each other.
- **virtual memory management.** The Mach interface provides flexible management of a protected paged virtual address space. User programs may, with minimal restrictions, control the contents, protection, and sharing of virtual memory and adopt customized semantics through the use of a general-purpose VM interface.
- **external memory management.** A specified region of memory may be specially allocated to be managed by a user-level memory manager. Although the kernel provides a default memory manager to back regular non-persistent memory against system paging space user programs may communicate with the kernel to manage persistent memory and govern the semantics of the region in an application-specific manner.

Because these tightly coupled abstractions are so interconnected and dependent upon each other it is difficult to describe each of them in isolation. Attempts to do so, however, with references to the other abstractions only when necessary, are made in the sections below.

2.1 Execution Control

The Mach execution control facilities make the creation of concurrent and parallel programs possible. The traditional notion of a process is broken down into tasks and threads, allowing a server, such as a memory manager, to use multiple threads of control while still sharing global data structures and other resources.

2.1.1 Tasks

The task is the basic unit of resource management. Each task has its own paged virtual address space, access to some amount of system resources, and access to communication with other tasks. Its virtual address space may be protected, copied, or shared between the owning task and other independent and dependent (descendant) tasks. Because of the resources available to it, a task brings with it non-trivial creation and management costs. In and of itself, a task does not represent any form of control flow, or point of execution. It may instead be envisaged as a container of such control flows, and a provider of shared resources and communication access to all the flows of control contained by it. The relevant task interface is shown in Table 2.1.

task_create(target_task, inherit_memory, child_task)
Returns a new child task based on the target task.
task_terminate(target_task)
Destroys the target task and all of its threads of control, and deallocates any resources held by the task.
task_suspend(target_task)
Suspend execution of all threads of control within the task.
task_resume(target_task)
Allows all flows of control within the task to continue execution.

Table 2.1 Subset of task interface

It should be noted that even though the task itself is not a flow of control the task may be suspended and resumed by suspending and resuming all of the control flows contained by the task.

2.1.2 Threads

The thread is the basic unit of computation, or a control flow contained within a task. It belongs to one and only one task, and shares the virtual address space, system resources, and communication facilities supplied by the task with the other threads executing within the task. It is a relatively cheap entity with low overhead that includes such things as its machine register state and some resource parameters. Although threads are scheduled independently by the kernel, on a multiprocessor multiple threads from a single task may be executed in parallel. Thread scheduling is also dictated at a higher level by the task, which may suspend or resume simultaneously every thread executing within it.

The Mach kernel does not provide any explicit mechanisms for synchronizing threads that access shared memory and data structures. Included in the Mach environment, however, is a CThreads package that does provide the desired functionality, allowing synchronization to be achieved through the use of condition variables and mutual exclusion locks.

2.2 Interprocess Communication

The interprocess communication interface from the kernel enables user tasks to communicate with other tasks both synchronously (remote procedure call) and asynchronously. The basic communication abstraction ensures secure object naming and access, which permits the same interface and abstractions offered to user-level programs to also be used between the kernel and user-level programs.

<code>thread_create(target_task, child_thread)</code>	Creates and returns a new thread within the target task.
<code>thread_abort(target_thread)</code>	Abort any instructions, primitive traps, or page faults (if allowed by the hardware) currently being executed by the specified thread.
<code>thread_suspend(target_thread)</code>	Suspend execution of the specified thread.
<code>thread_resume(target_thread)</code>	Allow execution of the specified thread to resume.
<code>task_threads(target_task, thread_list)</code>	Returns the set of threads belonging to a task.
<code>thread_get_state(target_thread, flavour, old_state)</code>	Returns the requested execution state for the specified thread.
<code>thread_set_state(target_thread, flavour, new_state)</code>	Sets the selected execution state for the specified thread.

Table 2.2 Subset of thread interface

2.2.1 Ports and Messages

A message in Mach consists of a fixed-length header, followed by one or more data parts, each of which is preceded by a type descriptor that describes the type of data and the size of the data part. Many message types have been defined to be used as templates for often-used and base types, while the option of defining additional user-defined types remains available.

A port is a unidirectional communication channel on which messages can be sent and received, e.g., by a client who requests a service and a server who provides it. Each port is essentially a message queue that is protected by the kernel and manipulated only via operations that transmit and receive messages. The state of each message queue is the number and ordering of the messages queued. While the port message capacity, or the number of messages that can be simultaneously queued, is modifiable the kernel makes

only one guarantee about message ordering. The kernel will ensure that messages sent by one particular task on a single port will be received in the order that they were sent. No ordering guarantees are made about messages from different tasks to the same port, or messages from the same task to different ports.

Each port can be accessed exclusively by means of a port right, or an entity that indicates the right to access the port in a specific way. Three types of rights may be held by a task:

- **receive right** - allows the holder to receive messages from the associated port
- **send right** - allows the holder to send messages to the associated port
- **send-once right** - allows the holder to send a single message to the associated port; expires after the message is sent

Each port has a single receiving task, or task that holds a receive right to the port, while any number of tasks may have access rights to send to the port. The receiving task also holds a send right to the port and is authorized to create new send rights to give to other tasks. Once a task has a send right it may send messages to that port until either it destroys its own right, or the port is destroyed.

Port rights can be copied and moved between tasks in messages, by specifying in the message that it contains a port right, and the type of right that it contains. The kernel can then, when the system trap routine is entered, perform the appropriate port management procedure to authorize future port operations by that receiving task, or even transfer ownership of the port, when applicable.

Upon request, the kernel can provide notification, to a specified port, of certain port right conditions. Some useful conditions include when no more send or send-once rights

exist for a port, or when the receive right has been destroyed, and hence the port no longer exists.

Ports and port rights do not have system-wide names that allow arbitrary ports or rights to be manipulated directly. Ports can be manipulated only via port rights, and port rights can be manipulated only when they are contained within a port name space. A port right held by a task is specified by a port name, or integer index into a port name space. Since each task has its own unique name space for its port rights the same port may be referenced by different names in different tasks. A port name only makes sense, therefore, within the context of the task that is referencing it.

The relationship between tasks, ports and rights is illustrated in Figure 2.1. In this figure, Task A holds the receive right for Port X (which implicitly gives him a send right as well). Another task, B, possesses a send right to Port X, but the name of its right may be located anywhere in its namespace, independent of where the receive right is located in Task A's namespace. The kernel maintains internal tables to provide port security, and associates port structures, such as message queues, with each port. The task namespaces must be flexible enough to allow the kernel to provide the task with an indication of the status of the port to which the name entry refers (if any). The four possible values that may be stored in each name space entry are:

- **MACH_PORT_NULL** - no associated port right is stored there
- **MACH_PORT_DEAD** - a right was associated with this name, but the port to which the right referred has been destroyed
- a port right - a send-once, send, or receive right for a port
- a port set name - a name which acts like a receive right, but allows receiving from multiple ports

Figure 2.1 Port and port right relationships

When a new port right in a task is acquired a new port name is generated. All send and receive rights for a given port share the same port name, but each send-once right bears a different port name from the send and receive name and other send-once rights. When a task is terminated its associated port name space is deallocated, resulting in the destruction of all of the port rights it previously held. A summary of some of the relevant port operations may be found in Table 2.3.

2.2.2 Object References

In addition to their uses for communication, ports are used to represent objects, much the same as capabilities in capability-based systems. Messages sent to a port representing an object can be interpreted by the receiver, or object manager, as requests for operations to be performed on that object. The act of sending a message and receiving a reply corresponds to making a remote procedure call on a potentially remote node.

mach_port_allocate(task, type, port)
Allocate a new port.
mach_port_allocate_name(task, type, port_name)
Allocate a new port with the specified port name.
mach_port_deallocate(task, port)
Deallocate a task's right to this port.
mach_port_destroy(task, port)
Deallocate all task rights to this port.
mach_port_get_receive_status(task, port, status)
Return the information about queued messages and send and send-once rights that have been created by this port.
mach_port_get_refs(task, port, right, refs)
Get the number of user references a task has for a right to a port.
mach_port_mod_refs(task, port, right, delta)
Change the number of user references a task has for a right to a port by delta.
mach_port_names(task, ports, ncount, types, tcount)
Return all names in the port name space of the target task that represent a right for a port, port set or dead name.
mach_port_rename(task, old_name, new_name)
Change the name of a right in the name space of the target task.
mach_port_mscount(task, port, mscount)
Change the tally of send rights created by this receive right for a port.
mach_port_set_qlimit(task, port, qlimit)
Change the queue capacity of the specified port to hold less than or equal to the specified number of messages.
mach_port_type(task, port, pmask)
Return a mask indicating the rights that the task holds for the specified port.

Table 2.3 Subset of port operations

Figure 2.2 Remote message send from Task A on Node X to a remote Task B on Node Y with Network Message Server acting as a transparent proxy.

The Mach kernel can itself be considered a server task with multiple threads of control that implement the task and thread objects. A request to create a task or thread returns a send right to a port that represents the new task or thread, and that can be used to manipulate it. Every task or thread in the system can be uniquely identified by the rights that exist in the system and are held by other tasks (even though, as explained above, the names of the rights may be different in each task that holds a right). Since a running thread must be able to acquire the rights to access itself and its containing task in order to make calls to the kernel and receive the replies, the kernel interface exports some primitive trap calls that provide the names for these rights. They are listed in Table 2.4.

mach_task_self()

Returns the port name for the calling task.

mach_thread_self()

Returns the port name for the calling thread.

Table 2.4 Identification traps

Because messages are sent by value and port rights can be transferred in messages it is possible to transparently forward a message received on one port by sending it on another, or by otherwise transferring the message contents. An important application of this principle is the Network Message Server [Jul89], a task that transparently forwards messages across a network. A message destined for a port on a remote host is in fact sent to a local Network Message Server proxy port, which then transfers that message across the physical network to be received by the remote Network Message Server proxy (see Figure 2.2). The remote proxy then completes the one-way communication process by sending the message to the remote destination port on the same node.

The indirection afforded by message-passing allows objects to be placed arbitrarily in a network, without regard to location or programming details (performance aside). A thread may then suspend and resume another remote thread without even knowing or caring on which host the thread is executing. It follows that in such an environment objects may be freely and easily migrated from host to host with complete service transparency.

2.2.3 Port Sets

Mach provides a port set abstraction that allows a server to receive messages from multiple ports without being multi-threaded or engaged in continuous polling. Ports for which a task holds a receive right can be added to a port set, after which the task can issue a call to receive from the port set and get the first message available from any of the ports in the set. The ports may not be prioritized and the port from which the next message is received is unspecified. Once a port has been added to a port set all messages queued on that port may only be accessed by actively receiving on the port set, and will be inaccessible by receiving from the port directly. A port may be a member of zero or one port set, and may be removed from a port set at any time, retaining the messages that may still be queued.

<code>mach_port_set_status(task, set_name, members, count)</code>
Return the current set of ports that are members of the named port set in the target task.
<code>mach_port_move_member(task, port, port_set)</code>
Cause a port to be a member of the specified port set.

Table 2.5 Port set specific operations

Like ports, port sets are represented by integer names in the same task port name space. Unlike regular ports, however, ports sets have no associated rights, meaning that receive rights for port sets cannot be passed in messages. This implies that only the allocating task (that created the port set) will ever be able to receive from, or be aware of the port set, while messages sent by other tasks to the individual ports in the set will be indiscriminately queued on the ports without any knowledge of whether the port belongs to a port set or not. The port set simply contains the collection of receive rights of the ports that are members of the set. Since port sets share the same name space a task may have multiple ports sets, and ports may be moved from one set to another. Table 2.5 indicates some specific port set operations, with the remainder of the operations supported by calling the regular port interface and passing a port set as an argument.

2.3 Virtual Memory Management

The Mach virtual memory system provides a machine independent interface that supports logical address maps and the mapping of a virtual address space. The kernel naturally allows tasks to access and modify the contents of such an address space, but also permits characteristics of sharing and protection.

The implementation of virtual memory is strongly connected with the more abstract memory object entity. This section briefly touches upon the features of virtual memory.

without delving too deeply into memory objects, a topic that will be covered extensively in a following section.

2.3.1 Virtual Address Spaces

A virtual address space defines the set of valid virtual addresses that a thread executing within the task owning the virtual address space is allowed to reference. It consists of a sparsely-populated indexed set of memory pages that is broken up into regions. A region is an internally-grouped virtually contiguous set of pages with the same characteristics, and must be allocated, even if in multiple calls to the kernel, to be addressable. It is named by its virtual address range within the address space. A region is typically allocated by making a kernel call to allocate the optionally specified address range in virtual space, with the characteristics stipulated in the call. Although the region abstraction is predominantly an internal system notion after having been allocated, and is usually transparent to the client task, a task may query the system for information about the region encompassing a specified virtual address (see Table 2.6).

Every task, at creation time, gets its own protected virtual address space, which is suitably destroyed when the task is terminated. When a new task is created its virtual address space is empty, and must be filled by allocating regions or ranges of paged virtual address space. The kernel may be requested to supply or allocate zero-filled memory that begins on a page boundary and spans an integral number of virtual pages, the virtual page size itself being an integral multiple of the actual hardware page size. Previously allocated regions may be similarly deallocated from the task virtual address space, affecting the range of addresses rounded outward to virtual page boundaries.

Figure 2.3 Task virtual address space snapshot

A virtual page will only ever be loaded into main memory when a client faults, or references an address, on the page. It will remain in memory as long as the kernel does not need the memory that the page occupies, but this is dictated by cache management and page replacement algorithms embedded within the kernel. When the memory is needed then the affected page will be paged out to a paging file on disk. A snapshot of a task virtual memory address space is provided in Figure 2.3, with the sparse regions (of varying page counts) shown either to be in main memory, on secondary storage, or a combination of the two. Table 2.6 gives a list of operations that a task may invoke upon virtual memory (belonging to itself, or another task).

vm_allocate(task, address, size, anywhere)
Allocate new zero-filled virtual memory at the specified address or anywhere space can be found.
vm_copy(task, old_address, size, new_address)
Copy a range of memory from one region to another.
vm_deallocate(task, address, size)
Deallocate a range of addresses, making them no longer valid.
vm_inherit(task, address, size, attribute)
Set the inheritance attribute for a region of virtual memory.
vm_protect(task, address, size, set_maximum, protection)
Set the protection of a region of virtual memory to be either the maximum or some protection lower than the maximum.
vm_read(task, address, size, data, count)
Read a portion of the target task's virtual memory.
vm_region(task, address, size, protection, max_protection, inheritance, shared, object_name, offset)
Return information on a region within the specified task's address space.
vm_write(task, address, data, count)
Write data to the virtual memory of the target task.

Table 2.6 Virtual memory operations

2.3.2 Virtual Memory Characteristics

Because all of the threads in a task share the task resources the virtual memory that is allocated within a task is shared memory. Any and all threads may view and modify allocated virtual memory, and contents changed by one thread are immediately visible to the others upon completion of the kernel function.

In addition to the read/write sharing amongst threads within a task, the Mach kernel also supports read/write sharing of virtual memory amongst tasks of common ancestry through inheritance. When a child task is created with inheritance specified, each region of its parents address space may be shared, copied, or omitted in that child as defined by the attributes of the region:

- **VM_INHERIT_NONE** - the memory range is left undefined in the new task
- **VM_INHERIT_COPY** - the range is copied (with copy-on-write optimizations) into the new task at the time of task creation
- **VM_INHERIT_SHARE** - the new and old tasks share access to the memory range

The inheritance attribute may be set to one of these three values at any time to permit or impede the inheritance of a range of virtual memory, but changes will only affect tasks that are not yet created. Existing shared regions, for example, would not be affected by changing the region inheritance attribute to **VM_COPY_NONE**, and the sharing task might not even notice if the originating task were to deallocate the region altogether.

A protection attribute of a range of virtual memory may be specified as any combination of permitted read, write and execute privileges, letting the task decide which permissions are appropriate for its application. Over and above the current protection attribute for an address range is a maximum protection value. In cases when a parent task may want to restrict the permissions that a child task will be able to grant itself when accessing shared memory, the parent may set the maximum protection value, allowing the child(ren) to increase the current protection up to the defined limit, but not surpass it.

2.4 External Memory Management

The external memory management (EMM or XMM) system allows user-level servers to manage memory regions in the virtual address space of a client task. It is used by the virtual memory system in the preceding section and also opens the door to various other applications with requirements to externally manage (at the application level) permanent storage, consistency, and data object sharing. This section is of particular consequence and relevance to the design and implementation discussed in this thesis.

2.4.1 Memory Objects and Memory Managers

All of the Mach notions of memory, above the physical level, are built on or around the abstract memory object. Named by a port, a memory object represents a collection of data bytes, or range of virtual memory that is mapped into a client address space, and exports functions to read and write that data. The port representing the memory object serves numerous purposes, one of which is to provide the kernel with a specification of the object being mapped. The port is also used for data transfer and communication purposes, as will be seen later. From a client's perspective, the memory object is just a memory range that is managed by an external task with a publicized port. The procedure that a user-mode task follows to utilize such a memory object is:

1. A task establishes a new memory range, or maps the memory object into its address space, specifying a port to name the memory object that backs the range.
2. The task attempts to reference a portion of this memory range (either by reading or writing it). Since the referenced page does not presently exist in memory, the client task takes a page fault. The kernel sends a message to the abstract memory object and requests the missing data page. The reply from the memory object resolves the requesting task's page fault.
3. The resident pages of the memory range, any of which may be modified by client tasks, are at some point evicted from memory. Such pages are sent in messages to the abstract memory object for their disposition.
4. The client task de-establishes the memory range, or unmaps the memory object. When all mappings of the memory object are gone the memory object is terminated.

The server that implements a memory object is called a memory manager, and is responsible for the initial contents of the object and for the permanent storage of its data if appropriate. When a client wants to map an abstract memory object into its address space,

using the *vm_map* call listed in Table 2.7, it must know the port of the memory manager task that backs, or manages the non-resident state of the object. The memory manager must make this port available, either in its own well-defined way, or by the Network Name Server supplied by the Mach environment.

```
vm_map(task, address, size, mask, anywhere, memory_object, offset, copy,
       current_protection, maximum_protection, inheritance)
```

Map a portion of a memory object into the specified task virtual address space according to the given address parameters. The memory object is copied before being mapped if so requested. The initial protection and inheritance attributes may also be specified for the new mapping.

Table 2.7 Call to map a memory object

The memory manager task is responsible for supplying non-resident pages to any requesting kernel, or saving dirty pages back to secondary memory for permanent or temporary storage. The semantics of any given abstract memory object are completely dependent upon the decisions of the memory manager, but due to their general duties of paging data back and forth these memory managers have also come to be known as pagers.

A special case, or example of a memory manager is the default pager, the quasi-user-level task that pages virtual memory back and forth to and from disk, supplying each task with a large paged virtual address space. Allocating a range of virtual memory using *vm_allocate*, as was discussed in section 2.3.1, equates to mapping an abstract memory object with a specified number of pages and the default pager as the memory manager. The default pager, although it runs in user space, is an integrated and trusted component of the system, but can nevertheless be exchanged with a user-defined pager when requested by a user with the appropriate privileges. Only under rare and controlled circumstances would this be performed, however, since a different default memory manager might change the semantics of virtual memory itself.

There are occasions when a pager may supply client services over and above the supplying of data on page faults and storage of dirty page write-backs. In these cases, the client has direct interaction with the memory manager as a user-level server. But the kernel, after being called to map the abstract object (and again to unmap upon completion), reverts to being a completely invisible player from the point of view of the mapping client.

In actuality the kernel plays a key role in the physical accessibility of the memory object, and acts as a kind of residency manager for memory object pages. Since physical memory must be controlled by the kernel, it exports a special cache object that represents the resident state of the memory object in the kernel, and manages the resident pages that are cached and available to clients. When pages are not cached the kernel initiates a page fault on behalf of the client, and requests the page from the memory object. When the pager supplies memory object data it does so to the memory cache object of the requesting kernel, which causes the data to be resident in the kernel that owns the cache object.

Figure 2.4 Object causal interaction sequence. Sequence of interactions between client, kernel and memory manager.

2.4.2 Interfaces and Communication

Over the course of its life, a memory object may be mapped simultaneously or at different times by a number of clients on local or remote nodes. To allow effective and efficient management and consistency of memory objects, communication between the memory manager and the various kernels in the network has been standardized to adhere to a defined protocol with specified interfaces. These interfaces provide sufficient messaging capability between the pager and a cache object on each host so as to grant the pager enough power to dictate the resident state of each page of the memory object on any particular node, as well as specify allowable permissions.

<code>memory_object_change_attributes(cache_object, cache_attribute, copy_attribute, reply_port)</code>	Set the attributes of the memory object on the host of the specified cache object.
<code>memory_object_data_error(cache_object, offset, size, reason)</code>	Indicate to the kernel that the memory manager cannot provide the specified data.
<code>memory_object_data_supply(cache_object, offset, data, data_count, deallocate, lock_value, precious, reply_port)</code>	Supply the kernel with a range of data for the specified memory object. Is normally sent in response to a <code>memory_object_data_request</code> message.
<code>memory_object_data_unavailable(cache_object, offset, size)</code>	Indicate that the memory manager cannot provide data for this region, and that the kernel may provide zero-filled data instead.
<code>memory_object_destroy(cache_object, reason)</code>	Initiate a termination of the memory object on the specified kernel.
<code>memory_object_get_attributes(cache_object, object_ready, cache_attribute, copy_attribute)</code>	Return whether or not the kernel can issue new data and unlock requests on the memory object, and the caching and copy attributes associated with it.
<code>memory_object_lock_request(cache_object, offset, size, should_return, should_flush, lock_value, reply_port)</code>	Request that the kernel do any specified combination of cleaning, flushing, or modifying the access protection of the specified range.
<code>memory_object_ready(cache_object, cache_spec, copy_attribute)</code>	Indicate to the kernel that the memory object is ready and is requesting certain caching and/or copy attributes for the object in the context of that kernel.
<code>memory_object_set_attributes(cache_object, ready, cache_attribute, copy_attribute)</code>	Request that the kernel set the readiness, caching or copy attributes for the specified memory object.

Table 2.8 Cache object interface

memory_object_change_completed(memory_object, cache_attribute, copy_attribute)
Confirmation from the kernel that the requested attributes in a previously-received memory_object_set_attributes have been set.
memory_object_data_request(memory_object, cache_object, offset, length, desired_access)
Request the specified data from the memory manager. Is issued by the kernel when it has faulted. The access it requests depends on the type of fault it took.
memory_object_data_return(memory_object, cache_object, offset, data, data_count, dirty, kernel_copy)
Provide the memory manager with data that has been modified or evicted from the physical memory cache. It may or may not be dirty and may or may not remain cached. The memory manager must act accordingly.
memory_object_data_unlock(memory_object, cache_object, offset, length, desired_access)
Request more permissions or access for a specified range that is already cached by the kernel.
memory_object_init(memory_object, cache_object, memory_obj_name, page_size)
Initialize a memory object and indicate that it has been mapped by a client on the same host as the specified cache_object.
memory_object_lock_completed(memory_object, cache_object, offset, length)
Confirm action taken by the kernel as a result of receiving a memory_object_lock_request message.
memory_object_supply_completed(memory_object, cache_object, offset, length, result, error_offset)
Confirm receipt of data pages as a result of memory_object_data_supply message(s) sent by memory manager. If an error occurred then indicate the offset.
memory_object_terminate(memory_object, cache_object, memory_obj_name)
Indicate to the memory manager that all clients from the source host have unmapped the memory object. Return the receive rights of the cache object control and name ports.

Table 2.9 Server interface

To maximize concurrency and performance of a memory manager the dialogue that occurs between the memory manager and the kernels is asynchronous. This does not restrict single-threaded pagers to be blocked in a remote procedure call (RPC) for every kernel demand or data provision. A measure of flexibility is afforded the server when responding to requests, acknowledging that in many situations the sender is using asynchronous communication for RPC semantics and is essentially blocked willingly waiting for the response. With asynchronous communication comes two individual and distinct interfaces -- one exported by the kernel (cache object) and the other exported by the server (memory manager). These two interfaces are shown in Tables 2.8 and 2.9, respectively.

Each of the interface calls that are made in either direction have implicit meanings and connotations associated with them. Although they can be used in other ways between the servers themselves (see [For86]), message exchanges with the kernel assume certain conditions, and the protocol should be well understood and adhered to by all memory managers.

2.4.2.1 Initialization and Termination

A call to indicate that a mapping has been made on a host is provided as a convenience to the memory manager, and allows it to initialize data structures and prepare to receive other requests from kernel on that host. Conversely, a call to indicate that all of the clients have unmapped the memory object on a given host is also provided to allow the memory manager to clean up any data structures that may be associated with those mappings. Together, they give the memory manager a clean view of the hosts on which the memory object is mapped, simplifying issues like sharing and consistency.

When asked to map a memory object for the first time, the kernel responds by making a *memory_object_init* call on the memory object. Included with this call are two ports:

- memory object control port - Port that is used by the memory object to send messages to the kernel from this memory object. Also known as the cache control port.
- memory object name port - Port used by the kernel to identify this memory object to other tasks.

The kernel holds send rights to the memory object port and both send and receive ports to the control and name ports, while the memory object holds send rights to the control and name ports and both send and receive rights to the memory object port. Both the control port and the name port are used to identify the shared cache that is used by the kernel to support the mappings of a memory object. The distinction between the ports is that only the control port can be used to affect the contents of the main memory cache on the host, and should only be available to the memory manager. Requests sent to the name port are ignored so that it can be used to identify the host mappings to less trusted parties, like clients themselves.

When a memory object is mapped into the address space of more than one task on different hosts the memory manager will receive an initialization call from each kernel, containing a unique pair of control and name ports. If a client maps a memory object that is already mapped by another client on the same host, however, the kernel does not send another initialization call. The interface does not identify individual mappings, allowing any mapping to use data in the main memory cache without any additional requests being made to the memory manager.

Figure 2.5 Initialization/termination message sequence. Mapping and unmapping of a memory object by a client will cause the ordered events to occur; either 4a or 4b will cause 5 to be sent.

After receiving the *memory_object_init* call the memory manager may respond with a *memory_object_ready* call to the kernel (control port) indicating that it is ready to accept subsequent calls for that memory object being mapped on that kernel. In this call it specifies caching attributes, possibly requesting that the kernel maintain object pages in cache even after all references have gone, object page size, and a copy strategy for the object. When the kernel receives this message it may then assume that it is free to pursue subsequent memory object operations.

When all mappings of a particular memory object have been unmapped on a particular host then the kernel on that host makes the *memory_object_terminate* call to the memory object. The receive rights to the control and name ports are also passed in this message so that the memory manager can destroy the ports, perhaps draining them first to see if there were any unreceived messages sent by the memory manager. When the kernel performs the *memory_object_terminate* it releases its send rights to both of the control and name ports as well as the memory object port.

Figure 2.6 Client read/write fault handling sequence. The memory manager may respond with either of 3a, 3b or 3c, but if 3c then instead of continuing, the client will trap on an invalid address reference.

A memory object may request that the kernel deallocate all mappings on a host by sending the *memory_object_destroy* message to the kernel. This will prompt the kernel to initiate the unmapping procedure that will conclude with it sending the usual *memory_object_terminate* message back to the memory object.

2.4.2.2 Reading and Writing

The most commonly used calls in the memory object interface are those used to read or write the contents of the memory object. The calls themselves are fairly straightforward, with the complications arising in the consistency calls that often precede or follow the data read or write calls.

When a kernel encounters a cache miss, or initiates a page fault, it issues a *memory_object_data_request* call specifying the range of the memory object that it faulted on. The kernel will only ever request multiples of the page size specified in the *memory_object_init* call. When the memory manager is able to supply the page data it responds with a *memory_object_data_supply* and passes the data to the kernel as an

argument in the call. If the memory manager cannot supply the data because of an error then the memory manager may respond instead with *memory_object_data_error*, causing any waiting client threads to take a memory failure exception. Alternatively, the memory manager may reply with a *memory_object_data_unavailable*, indicating that the data was not available to be sent, and the kernel may supply the client with zero-filled memory.

The choice of a caching paradigm implies that the kernel only writes back data that has been modified, with clean pages being flushed by the kernel at its own discretion, and without necessarily informing the memory manager. To clean dirty pages the kernel sends a *memory_object_data_return* specifying the data to be written back to secondary storage. The memory manager does not need to respond to it, but must realize that the data that it is receiving has been allocated in its own address space by the IPC subsystem, and must be deallocated when it is no longer needed.

2.4.2.3 Consistency

The kernel allows memory managers to withdraw access to its data from the main memory cache, either temporarily or permanently, or to extract the contents of pages that have been modified. A single cache control interface call allows any combination of these operations to be performed atomically, with a corresponding server interface call that returns a confirmation that they were in fact completed. When a memory manager wants to activate such an action, it makes a *memory_object_lock_request* call to ask that the kernel:

- Alter the access restrictions specified in the *memory_object_data_supply* or a previous *memory_object_lock_request*
- Write back modifications made in memory (clean the pages)
- Invalidate its cached data (flush the pages)

These operations will be applied to a page range as specified in the call, but only those pages that were cached by the kernel at the time the message is received will be affected.

When the kernel completes the request it performs a *memory_object_lock_completed* call on the memory object, so that the memory manager may know at which point it may assume that the relevant pages have been written back or changed in the way they may be accessed.

When a kernel makes a request for data in a *memory_object_data_request* call it specifies the permissions that it is asking for. These permissions are granted for the data pages when they are provided in the resulting *memory_object_data_supply* call. But if a kernel asks and is granted a page with read privileges and then takes a page write fault the kernel must ask the memory manager for the additional permissions, without asking for a new copy of the data. This is accomplished by the kernel sending a *memory_object_data_unlock* to the memory object and the memory manager responding with a *memory_object_lock_request*, granting the kernel the new permissions, without sending any data. All that need be exchanged between the two parties is the range in question and the protection attributes.

Chapter 3

DESIGN

An object has evolved to be defined in many different ways by many different people. Regardless of the group queried all would agree on the need for persistency. This agreement would emerge because of the indisputable applicability of objects that persist across system shutdown, failure, or maintenance. In addition, with the dawning of a new generation of distributed object-based and object-oriented operating systems and/or kernels the persistent object must itself be distributed in nature, or at least be managed by a distributed object manager or server. This chapter discusses some of the design issues and decisions that relate to distributed persistent objects and the server that manages them.

3.1 Objects

In the object-oriented paradigm an object is a combination of two things: state and behaviour. The object state, or data part, is stored as part of the object and can be queried or modified through an interface that is implemented by a set of operations that manipulate the data directly. The representation of the object is therefore encapsulated by the interface, since a client need only know the interface in order to access the object and has no way of directly accessing the data (see [Atk91] for a more detailed coverage of the object-oriented paradigm). The object can be visualized as a boiled egg (see Figure 3.1), with the inner yolk being the data, and the surrounding white being the operations that operate on the

object data. The operations, themselves, can only be invoked by a client through the encapsulating shell that is the object interface.

Figure 3.1 Boiled egg model of an object

A client makes use of the object interface by choosing messages from it to send to the object. The object is then the receiver of the message and must recognize the message sent to it. It must also implement the operation that is to be called as a result of the message. These operations and the messages that an object should understand are typically stored in the object class, so that it may be able to identify valid messages sent to it at any time in the future. A class object is a grouping of objects with the same behaviour and the same state structure. With these things in common to all of the members of the class the structure of the state and the behaviour, or operations, may be stored in the class object for all of the members of the class, avoiding duplication of storage. Each individual object in

the class may, however, differ from others in the class by its actual state data. These individual objects, or instances of the class, usually need only store their state data and an indication of the class they belong to. Then, when a message is passed to a particular instance the appropriate class is found from the object and the relevant operation is looked up.

The objects implemented in this project follow the conventional specifications of an object in the context of an object-oriented system, as described above, except in a few areas. The Mach microkernel does not profess object-orientation but instead claims to be an object-based operating system. As such, its objects exhibit most of the above properties, plus others that were not discussed (e.g. polymorphism, when an object reference may refer to any object, the exact type of which may be determined at run-time), but lack a key feature that characterizes object-oriented systems--that of inheritance. With inheritance a class hierarchy could be established, which would allow subclasses to be members of other classes, or let objects inherit behaviour from every class along its ancestral class line. Mach objects do not support this notion, and for the sake of consistency and generality the persistent objects developed for Mach do not provide inheritance either. Extra server layers that may be built on top of the distributed persistent object server, such as an object-oriented layer (see COOL [Hab90]), would desire and require such provisions.

By themselves, Mach memory objects are neither persistent, nor coherent by design. In fact, they are rather loose abstractions supported by a user-defined external memory manager, with the object semantics and characteristics defined by the memory manager and its implemented usage of the XMM protocols. The objects implemented by the distributed persistent object server described in this thesis must increment the basic notion of Mach objects with added characteristics that include sharing and persistency.

This means that issues such as object naming, consistence, caching, coherence, and persistence had to be addressed and resolved.

In an object-oriented context, the objects referred to in this thesis might be described by a single class called `DistributedPersistentObject`, where each object created would be an instance of the class, and the server would be the class object that exports or implements the behaviour. Instances would differ from each other by the values of their instance data such as name, size, actual data contents, and so forth. Unlike other systems, exact replicas of instances may not be created (at least the name must be different).

3.1.1 Object Persistency

Since our aim was not to build an object-oriented environment, but only to provide a template for persistent Mach object manipulation, some of the inherent problems that plagued other persistent object implementors did not apply. Inheritance, for example, if not already a characteristic of the existing system base, might be a non-trivial feature to achieve.

One of the principal difficulties that most persistent object systems must overcome is that of object referencing. The question of what to do with a reference to one object embedded within another (the reference being embedded, not the object) requires attention due to the temporal differences of persistent object referencing. Since objects may be resident in different portions of memory during different sessions memory references that applied to the object the first time could point to erroneous or unused memory at some time in the future.

Once again, the Mach server can defer this, and many other problems relating to usage of the objects within a specific environmental context, to the application or environment itself. In this way the environment will be restricted as little as possible by the constraints of the underlying server, and can make appropriate decisions for itself.

Clearly, persistent objects must be backed by secondary storage to outlive volatile memory after the software that created them has terminated. For normal operation and to maintain a minimal amount of fault tolerance the object must be read in from backing store the first time it is accessed and written back after each object modification. Since backing storage device I/O is always done in coarse-grained blocks, and because Mach virtual memory management granularity is even coarser, the price of implementing fine-grained objects could be higher than for coarse-grained ones. The reason for this is because the cost of I/O would be the same, except that fine-grained objects would have to be administered as smaller pieces of coarse-grained ones, incurring an additional management cost. For this and other reasons the objects are created to be of virtual memory page granularity.

To reduce I/O costs at the device level the Bullet File Server in [Ren89] uses contiguousness and makes the files immutable. Files are stored contiguously on disk and may only be created, read or deleted. Although this approach produced some handsome performance results, we have taken a somewhat gentler position and decided upon objects that are semi-mutable. Objects may be created, read, written, or deleted, but not increased in size. Thus, once an object is created it remains the same size, with its contents being modifiable. Because there are no other design restrictions that cause object sizes to remain static, modification would be fairly straightforward should it be warranted. The performance of such object growth may not be as favourable since some gains have arisen from assumed static object size.

3.1.2 Distributed Objects

In a distributed system of nodes, each one optionally supporting a disk device, objects may be dispersed around the system. If they are persistent then they will reside on secondary storage (disk) somewhere in the network. For our purposes an object resides (its principal copy is resident on disk) on exactly one node. It is managed by that node and

is unavailable in the event that the node is down or severed from the network. If the node on which it resides is not down, then object communication (messages sent to the object) will eventually get handled by the resident server, which will operate directly on the object.

Most distributed object systems assume that the objects may be of any type, as described in [Ry90]. In this context the object may be a simple string object, and be queried across the network, or it may be an application to be remotely invoked. In fact, not only can objects have different types, but they can have different (location-dependent) implementations of the same type. These concepts point toward the notions of heterogeneous systems with specialized nodes that may be suited for specific jobs.

Another argument may be made for not typing the objects at all. Untyped objects leave the typing decisions, which are often language-dependent, to a higher level application and opens up portability doors to different platforms and languages. Like Opal [Cha92] we have opted for untyped persistent objects that are both application levelled and language-independent, providing outer layers with a resource to use in its own application-specific manner. The distribution of the objects may be transparent to those applications that do not want to know details such as location, or they may be accessible by querying the object. In this fashion the applications that want to exploit spatial characteristics of objects may do so at their own discretion.

3.1.3 Object Naming

Techniques to specify and locate objects in the network range from using special logical memories (see [Buh89]) that also confront the referencing problem described in section 3.1.1, to unique identifiers (UID's) that are global in scope (see [Por89]) and may contain components indicating location or other data. The most commonly-used scheme is the latter, although it does imply some management.

An important factor in such a scheme is the generation of UID's in the network, since at creation-time an object must be assigned a UID that must be unique and known by all of the nodes in the system. This problem is tackled in [Hei89] and may be generalized to cases when objects may span domains, i.e., a UID is generated outside of the immediate network and must be mapped in to the system.

In order to enable the user convenient access to existing objects a string-based naming mechanism is often used. From the user end the object is named by a string, which is then mapped, either by the object manager or a separate name server, to a UID. In some systems, like Arjuna [Dix88], the object location is embedded in the string itself, which is stored in the name server.

In the Mach context an object is referenced by a port. This port may take on different names according to the task that holds a right to the port, remembering that every task has its own local port namespace. To operate on the object a task need only send a message to the port that names it. Upon reboot, the task must have some sort of a stored handle or reference that it can use to retrieve the same object in a consistent way. Since the task rights given to it take on a random port name (assigned by the kernel) this would not be possible. Seeing that the port cannot replace the UID another option was explored.

To facilitate the necessary string-based interface to clients, yet avoid a situation of double mapping (string to UID and UID to port) we use a combination of the two. When a user creates an object he may specify a string name, which will be the global UID of the object instead of letting the system generate a numeric one. By using string-based UID's only a single mapping need occur (string to port) and the separate and redundant user layer need not be added, since it is already incorporated into the design of the server. Also, by using a virtually infinite namespace there is no danger of overrunning a pre-specified or hardware-dependent numeric data type that would be used for numeric UID's.

This approach, although it produces some substantial savings from the perspective of clients and the environment in which the server runs, is not without its drawbacks. Firstly, when the object is created the string must be examined by each node to ensure uniqueness and that it is presently unused in the system. Although this may be true in most numeric UID generation schemes the internodal IPC that must transport strings reports increased communication costs, since strings may be much longer in length than numeric identifiers. An 8-bit byte reduces to naming only a subset of perhaps 200 alpha-numeric and punctuation characters (only 100 or less of which may be actually used) out of a possible 256 binary number representations. Added to this is the storage cost of object table mappings and, lest we forget, permanent (secondary) storage of the UID's. Hashing and other string operations combine to produce a non-trivial price to pay for user-oriented objects. But when seen from the vantage point of applications that will use the objects and the server that exports them, it becomes apparent that if such convenient naming did not exist an additional and equally expensive distributed layer (such as a distributed name server) would need to be implemented, doubling the cost of object retrieval. Alternatively a system could implement a centralized server, nullifying altogether the advantages gained from the distributed object server design.

3.1.4 Object Sharing

If an object is to be distributed then it is taken for granted that each and every node should be aware of its existence. Every node should also be able to send an object messages (to query or mutate it). Different nodes accessing and modifying the same object is known as object sharing (see [Dan93], [Atk91] and others).

Many systems comparable to persistent object systems are file systems, which incorporate the notion of opening, or a client indicating intent to perform I/O to the file. Since the client indicates at file-open time that he wants to either read, write or both, the server can effect what it deems to be appropriate locking or management of file contents.

In most systems, for example, once a client has a file open for writing, other clients are prevented from opening the file until the file has been closed by the writer. Thus, even though the file may be read and written by each and every node in the system, no two writers may have the file open for concurrent writing. This is called sequential write sharing, and is an obvious restraint that hinders clients from concurrently writing to the same file. When two clients concurrently share write access to the same file it is called concurrent write sharing, and is a much more flexible and powerful semantics. This is supported by few file systems including Sprite, as described in [Nel88]. Some systems that do not, such as Unix, do not provide any mechanism to stop clients from placing themselves in precarious and undefined situations where concurrent write sharing exists, but only go so far as to caution users that doing so may be hazardous.

When a client intends to perform I/O to an object managed by the Mach persistent object server it maps the object into its address space (see 3.1.7 for more detail) and then reads and writes directly to the mapped region. Because there is no notion of the object being open or closed, only whether the object exists at a node, concurrent write sharing is supported. Many clients, either on the same node or on different nodes, may map the object and thereby concurrently read and/or write. The server manages the consistency of the object data by communicating with the kernels involved, and ensures that no two clients actually write to the same offset in the persistent object at exactly the same moment in time. In this way, clients need not concern themselves with data consistency, as is necessary in the SSE Data Storage System [Wri89], or with avoiding circumstances that may be perilous or have undetermined or untrustworthy outcomes.

A key design decision that affects the sharability of Mach persistent objects perhaps more than any other single issue is that of the granularity of object management. Because state information is maintained for every page of object data, not only can clients on different nodes perform simultaneous reads, but they can also simultaneously write

different pages of object data in true parallel fashion, with no communication collisions of any kind. With the resolution of sharing increased to medium-grained data pages, a shared object becomes much more prone to being used in a shared manner, and lends itself to a wider range of parallel and distributed applications that may require the ability to concurrently read and write the same object.

3.1.5 Object Data Distribution

Even though an object is shared, it is resident on only one node. This node acts as a sort of home base for the secondary storage of the object and is the node on which the server or object manager for that object runs. All messages destined for that object are sent to its resident node and handled by the resident server.

Figure 3.2 Distributed object page caching

When object data is read or written to it is often inefficient to perform these operations over the network, i.e., receive the requested data from, or send the data to write to the resident node. If a subsequent operation were to take place immediately following the first, at the same object offset or at the address immediately succeeding it, a similar data transfer would again have to transpire. It is clear that this would lead to needless and excessive network traffic, and could be avoided by simply allowing the kernel some data privileges. Instead of performing all of the operations at the resident node, a copy of the data can be sent to the kernel of the client node and stored there temporarily (cached) to be used to perform the operations locally. The result is that clients can access and modify object data as if it were local, without incurring network communication expenses with each reference.

The situation that the object must deal with is shown in Figure 3.2, where pieces of its copied data are distributed throughout the network, or cached by the various kernels on client nodes. Management of these data regions, especially when permission has been given to a kernel to modify the data, must be meticulously carried out by the server so that the object data remains consistent. This can be accomplished by communicating with the kernels through the use of the external memory management protocol described in section 2.4.2.

3.1.6 Object Characteristics

To meet the needs of as many applications as possible the persistent objects exported by the Mach persistent object server are designed to be general purpose in nature. Because of their lack of specialization they are not able to beat the performance of such special-purpose servers like Amoeba's Bullet Server, but they are able to outperform other general purpose file servers, showing that they are well-suited to many applications as opposed to only one.

When studies were conducted in [Ous85], [How88] and [Bak91] that measured and recorded statistics on persistent data in networks (files, in these cases) some emerging trends were noticed. It was found that the majority of files were small in nature, while the largest part of the network bandwidth was used up by a few big files. To respond to this, smaller persistent objects should remain relatively cheap and fast, while very big objects should be handled with as much discretion and careful management as possible.

The size of an object is an issue that has always related to both performance and storage utilization. In an effort to trade off these two determinants four points were noted:

1. Clients continually seek better performance
2. Primary and secondary memory has become cheaper and more accessible
3. Applications have also raised their memory requirements, allowing less to be used by others
4. Memory still costs money, and is a finite resource

Concluding from these statements that if one of the two could be sacrificed slightly more than the other it would be storage utilization, the granularity of objects was chosen to be the size of the virtual page, 4K on the present platform. This means that no object may use less than 4K, and that objects may be any multiple of 4K-sized pages in length. Although utilization may be lower than for a more conservative choice, fragmentation is virtually non-existent, as will be seen in subsequent implementation sections.

When an object is created, it is by default a shared object, and will be transparently accessible from all of the nodes in the network, without regard for its actual location. An occasion may arise when shared objects are not needed or wanted. In these situations objects can be created to be exclusively local, since shared objects bring with them an associated cost that may not always be warranted. In these cases the object may be

specified as a local object, and will not be able to be transparently accessed by other nodes. If remote access is later necessary then it is possible to gain a reference to it by explicitly going through the server local to the object.

3.1.7 Object Operations

The process by which a client may operate on an object begins with a request to the server, specifying the desired persistent object name. The server will return a reference (send right to the object) to the client, which the client may use for all subsequent object operations until either the client discards it (by either explicitly doing so or by terminating), the object is unavailable (because of server shutdown, or node/link failure in the network), or the object is explicitly deleted.

After the client has obtained a reference to the object a number of subsequent operations may be invoked upon the object through the use of message-sends. The client-object interface is listed in Table 3.1.

Conspicuously absent from this interface is a set of object data accessing and modifying operations. If there were such a suite of operations then it would not only allow a client to waste network bandwidth to perform explicit I/O across the network, but also make it possible to circumvent the prescribed method that performs I/O on local cache copies of the object. If the client desires to access or mutate object data he must map the object into his task address space through the use of the kernel *vm_map* call (see 2.4.1). Once mapped, pages of referenced object data are cached by the local kernel and used to serve the client. In this manner all I/O is done locally, and the only data that travels on the network is either an object page being supplied to a kernel for caching from an object manager or an object page being returned to an object manager for cleaning from a kernel.

po_getname(object, object_name, nchars)
Returns as much of the object name string as will fit in the supplied memory location.
po_gethost(object, host_name, nchars)
Returns as much of the name of the host where the object resides as will fit in the supplied memory location.
po_getsize(object, object_size)
Return the size in pages of the specified object.
po_delete(object)
Asynchronously delete the specified persistent object (without waiting for completion).
po_deletewait(object)
Synchronously delete the specified persistent object (waiting for completion before returning).
po_flush(object)
Synchronously cause all dirty object pages to be written back to the object manager of the specified object.

Table 3.1 Client-Object interface

3.2 Persistent Object Server

With persistent objects potentially resident on any node in the distributed system, it is necessary for a server to also exist and be executing on any node in the system. A persistent object network, then, is a collection of nodes, each of which is running the persistent object server. Whether or not objects exist at any given time on a node is immaterial to the server, since it must run in order to serve its local clients in any case. If, additionally, there are objects that reside on that node, then the server will be the object manager of those objects and will serve clients on other nodes as well. The role of the server is therefore twofold:

- 1) to respond to requests made by local clients
- 2) to act as object manager for all locally resident objects

3.2.1 Server Requests

Without an existing object handle there would only really be two requests that a client might want to make of the server. They are creation of a new persistent object and retrieval of an existing one, and are both included in Table 3.2, making up the client-server interface.

<code>po_create(server, object_name, object_size, local, time_flushable, object_key, object)</code>
Create a new persistent object with the specified name and size. Set its characteristics according to specified flags and values. Return a send right to the object.
<code>po_retrieve(server, object_name, object_key, object, object_size)</code>
Return a send right and the size of the named object, given a valid object name and key.

Table 3.2 Client-Server interface

3.2.1.1 Object Creation

When a client wants to create a new persistent object, the default server that it contacts is its local one. Whenever any server receives a request to create an object it does so on its local node, thus, by default, objects are typically created on the same node as the client. When a persistent object is to be created on a remote node, the port of the persistent object server that is running on the remote node must be obtained by issuing a look-up call to the Mach Network Name Server (a publicly accessible server that all persistent object servers register with). Once the server port has been retrieved the client simply makes the creation request to that server, and the server proceeds to create the object on its node. The server has no knowledge, therefore, of where the client is located, and nor does it care, but it will most often be the case that the client will be a local one.

The uniqueness of the object name must be checked and guaranteed by the server, since the object will be globally accessible and may not be ambiguous. This is done by broadcasting in an ordered manner to all of the nodes in the network and waiting for replies stating that no object is being created simultaneously on those nodes. Because the broadcast is done to an ordered set of nodes ties will not be possible. The only room for ambiguity is a very small subset of cases, one of which is where all of the nodes in the network are down except for the node initiating the creation broadcast, and one of the downed nodes did have a resident object with the same name. In this situation, when the node came back up again a name collision would occur and to prevent ambiguity the older of the two objects would not be allowed to rejoin the system under the same object name.

When specifying the size of the object, a client indicates an integral number of pages to be allocated on secondary storage by the server for that object. This size will remain static throughout the lifetime of the object, and can be used by the server to allocate sufficient resources at creation-time to alleviate the cost of doing so later on, or while an operation is in progress.

An access key is also specified in the *po_create* call and will be used to retrieve the object in subsequent retrieval requests. This will be discussed in more detail in section 3.4.

3.2.1.2 Object Retrieval

The other significant responsibility of the server is to provide a facility whereby clients may obtain a handle to an existing object. The handle, or port send right, is generated by the local server, and can be obtained by specifying the object name and key in the *po_retrieve* call to the server.

A local synchronous message dialogue is all that is necessary to retrieve an object handle thanks to the existence of a distributed object table. Each node has a copy of the table that gathers all of the object names in the system, each one mapped to the port of the

object that is named. The table can be maintained and updated to reflect new mappings by the creation broadcast described in the previous section. When objects are deleted another broadcast occurs to deregister them from the object tables that abide at the non-resident nodes. By allowing the object table to be stored at every node the object retrieval operation reduces to a local RPC call instead of the broadcasting that is done by other persistent object servers, such as PORTLANDIA [Por89]. This trade-off optimizes retrieving object handles, an assumedly more frequent operation, over creation and deletion. The alternative still exists, however, to create local objects and escape the broadcasting expenses associated with object creation and deletion. The obvious sacrifice is the sharability of the object in the network.

Due to the transparency of ports, and the lack of ability to be able to determine the location of the holder of the receive right for that port, object retrieval is also completely transparent. A client request to look up an object, or retrieve a handle for that object, results in a quick reply that contains an object port right and the size of the object. The location of the object is not known (except by making an explicit call to the object itself, asking on which host it resides). The size is returned so that the client may be able to properly map the object into its address space.

The interactions between clients, servers, and kernels are graphically demonstrated in Figure 3.3. The client-object interface arc that extends to the server on Node Y applies when the client on Node X operates on an object that is resident on Node Y. In the more general case, an arc should extend to the object server of every node in the network (that supports resident objects). It may even be the case that the object is a local one, so an arc would also be directed to the local server (on Node X).

Figure 3.3 High-level client-server-kernel interactions

3.2.2 Object Manager

The persistent object server does very little work as an actual server, as demonstrated by the minimal client-server interface shown in Table 3.2. Instead, the bulk of its resources are dissipated in managing the objects that reside on local secondary storage.

3.2.2.1 Object Manager Clientele

To the object manager, clients may be either users of an object, running in user space and operating on the object by sending it messages and/or reading and writing the object data, or kernels running in protected system space at the same site or at any remote site. When users send messages to the object, the messages relate to the object characteristics, and are received directly by the object manager and handled accordingly. Likewise, when the requests are from kernels, with the distinction that the messages relate mostly to object data, and the interface is a predefined one that is specified by the External

Memory Management subsystem (outlined in section 2.4.2). The kernel message exchanges are for reasons of cache coherence and object data consistency, since the kernel benefacts the server with cache space and communication opportunities. The kernel provides its own cache management, but also permits client (server) involvement. Indeed the kernel must be careful not to allow its clients too much interfering leeway, lest it not be able to supply adequately for itself. The net result is to the servers advantage, and object data may be cached at remote sites up to the limits of available RAM space. Additionally, the kernel will extend the service of local client cache sharing, allowing all clients on one node the right to access the same cache copy without even bothering to trouble the remote server. Sharing will obviously be conditional upon equivalent access privileges, and will result in a page request to the server if new privileges, not already allowed for the data page in question, are required. The only stipulation that the server must live with is that the kernel may evict pages of object data of its own volition, when its virtual memory or system buffering needs must be met, or when it simply needs the memory for more recent object operations.

For the purposes of the remainder of this section, as well as the following two sections, the term client will be used to refer to the users that send messages to objects, and not the kernels that coordinate the data caching amongst the nodes. In subsequent sections, however, clients may refer to either of the two, depending upon the context in which they appear.

The operations that a client may invoke upon an object (already shown in the interface description in Table 3.1) are available to clients that have an object handle in their possession. The first three, *po_getname*, *po_getsize*, and *po_gethost* are simple querying methods that permit a client to inquire of the object and find out certain of its basic characteristics. The first and second would not appear to be too useful, since in most cases the handle was obtained by doing a retrieve in the first place (see section 3.2.1.2) and the

name and size was either known (passed in) or returned (passed back). This is not always the case, however, since retrieving an object handle equates to being given a send right to the object port, which in turn can be copied and passed on to another client at will. A client that receives a handle from another client typically will not know the name or size of the object, since all it has is a handle, or reference to the object. It must send messages to the object if it wishes to know more details about it. The host on which the object resides is not known by any client, unless it explicitly queries the object and sends a *po_gethost* message to it.

3.2.2.2 Object Deletion

In order to delete an object, removing it from secondary storage and system existence, a client must have a handle for that object. Objects may be deleted by any such client, irrespective of whether or not it is locally or remotely located. Since there are no server-offered safeguards in place that could warn clients of object deletion, clients should ensure that there is adequate coordination and synchronisation between users to prevent the possibility of clients trapping from invalid memory address accesses. Clients may register with the kernel to receive notification of object port deletion, but since the server does not maintain any user references and only communicates with clients when it receives a request, it cannot be expected to notify clients of such circumstances. In this case either an application or third party server would provide the locking to help clients from deleting an object right out from under other clients.

Two flavours of deletion provide the same functionality but with different communication semantics. A simple *po_delete* call on the object will return almost immediately, and the actual deletion of the object will occur behind the scenes on the server's own time and at its discretion. When a client wants to be guaranteed of the deletion, perhaps because it may need the resources, or want to recreate the object with different characteristics, then the synchronous *po_deletewait* call may be used. In this case

the server will return to the client only after the object is out of commission and the resources previously tied up by that object have been freed.

When the deletion of an object is initiated a deregistration of the object must occur at every node, removing it from the object tables that it registered with when it was created, or when its local server was rebooted (if it crashed since object creation). Once this one-way communication has transpired (the object manager need not wait for replies), the object will not be retrievable from any other site, and any attempts to do so will result in an error indicating that the object does not exist. This will follow even though the object may not yet be fully removed (still existing somewhere in secondary memory). The remaining primary and secondary resources can then be reclaimed by the object manager in batches, or when it has some spare background cycles.

3.2.2.3 Object Flushing

In the course of sharing, objects will be read and written by clients on various nodes. With data offered to kernels in page quantities it would be possible, in the absence of any other requesting kernel, for a single kernel to have a page of object data cached for long periods of time. When another client at another site write faults on the same page then its local kernel will contact the object manager and the object manager will automatically make a request to the caching kernel to write back the data. If no such fault occurs, however, the caching kernel may continue to cache, and let clients write to the data indefinitely.

Under the same conditions, the client may have written to the page of data and not want to rely on other clients to access it in order to instigate page cleaning. Although the unmapping of the object by the client would induce the desired writing back, the client may not be ready or willing to terminate its mapped condition, since it may need to perform subsequent I/O to the object and not want to have to remap the object.

Other clients, even though they may not need to reference data in the cached page, may require that the object be updated for reasons of performance and reliability. For this, and all of the reasons stated above, it is reasonable to grant to clients the ability to flush all dirty object data back to the server to be cleaned. Not surprisingly, a handle for the object is required. The operation of flushing is achieved by sending the *po_flush* message to the object, which will then engage in a dialogue with all of the caching kernels and eventually conclude with a return reply to the client confirming that all of its dirty pages have been cleaned and written back.

The flushing semantics for client-instituted flush operations are precisely defined and should be clarified to avoid misconceptions by assuming clients. Firstly, by flushing, it is not meant that the pages held in the caches of various kernels are invalidated, only that the dirty ones are written back to secondary object memory on the resident node of the object. Secondly, the client that launched the flush cannot expect, upon receiving a reply, that the object has not been modified in the intervening time since the flushing call was made.

The second clarification weakens neither the consistency guarantee, nor the assurance that a read will always return the most recently written data. What it implies is that the object data that is written back, and is in place on the resident secondary storage of the object, is only the most recent up to the time at which the call to flush was made by the client (or received by the object manager). For example, if a page that was cached by a kernel on the client node was cleaned immediately following the request, yet requested for writing again by another kernel before the rest of the object pages have been cleaned, the object data that was cached has been modified and no longer contains the most recently written data upon return from the call. In fact, neither does the resident secondary storage contain the most recently written data at that moment. A subsequent read or write operation by the client will not produce stale data, however, since the kernel will fault and request a

new cache copy of the page, furnishing the client with the most recently written data. The moral of the client flush operation is, therefore, that the resident node will contain the most recently written back data only up to the time the flush was initiated (and not up to the time the call returns).

3.2.3 Server Concurrency

The server is multi-faceted in nature, and is required to concurrently support multiple objects, as well as handle client requests. To allow the server to accomplish its mandate of concurrent processing it must itself be designed to be a concurrent process, either as a multi-tasked entity, or a multi-threaded task.

With an elaborate set of task-related calls exported by the Mach kernel, it was originally a difficult choice to decide whether to make the server multi-tasked or multi-threaded. After further consideration, the sharing of task resources by a number of threads, all executing within the same task, emerged as a beneficial alternative to the more expensive option of individual tasks each having their own address space. With much of the reasoning that supports this decision deferred to the next implementation chapter, it suffices to say here that communication, resource costs and resource sharing all played a considerable role in the design and implementation of the server. These issues all motivated the multi-threaded design approach.

Even with the availability of the CThreads routines, distributed as part of the Mach distribution package, dynamic thread creation and termination could easily become an expensive endeavour. If a thread were assigned to serve one particular object, then the life cycle of a short-lived object could incite unwarranted thread management costs that could also tie up main server threads. On these grounds, a static thread approach was employed.

Figure 3.4 Multi-threaded request dispatch

When the server is booted, the operator specifies the number of threads that will be available to receive requests. In the absence of a specified number, a default thread count is assumed. The number of active threads does not limit the number of clients that the server may attend to, or even the number of objects that it may support. Each thread acts as a proxy for both the server and all of the objects that are locally resident, receiving request messages and replying as appropriate (see Figure 3.4). Indeed, any number of threads may be serving clients that have outstanding requests for the same object. Clearly, the sum total of synchronous requests, destined either for the server or for any of the objects, cannot exceed the number of active threads or it must wait in a queue to be served.

The homogeneous thread approach that was taken circumvents the danger of making limiting assignments of threads to objects, and averts the potential of blocking requests for some objects while other threads sit idly by waiting for requests for their own assigned objects. It is desirable to allow threads the flexibility to serve requests for different objects if many objects are actively being operated on, yet also be able to band

together when the demands made on one specific object become great. To facilitate this, objects must be structured in such a way as to be concurrently accessible, yet at the same time maintain secure and consistent state and data.

3.3 Fault Tolerance

Establishing a degree of fault tolerance in the persistent objects and the servers that manage them was considered to be a vital component of the design. But the objectives of the project did not entail 100% reliability and tolerance of all conceivable hardware and software errors. Instead, an attempt was made to sustain, to a reasonable extent, the continuation of execution in the face of a number of more commonly occurring faults, or at least gracefully detecting and reporting those that could not be reconciled with. This section will outline some of the strategies used and describe how they aid in the reliability of the persistent object system.

3.3.1 Object Reliability

To adequately protect an object from software crashes both the object state and data should be safely conserved in non-volatile memory. The conservation process should involve frequent updating of the object to secondary storage in order to prevent, as much as possible, loss of recently-written data. This is primarily accomplished through the means of two specific features, system-initiated object flushing and immediate write-back.

When data is cached at a site and the situation arises where the data page has not been requested by any other client (see section 3.2.2.3 for a description of client flushing) then the object may continue to be written without being written back to secondary memory. If at any time during this caching period the site should crash then the data that had been written only locally would be lost. To minimize the possibility of this occurring, and also minimize the amount of data that would be unsalvageable in the event that it did, the server performs a system-initiated flush on objects at regular system-defined intervals

(currently set at 30 seconds). The effect is exactly the same as if a client had requested a flush, so that all pages with write access are requested from the various caching kernels to be returned and written back to secondary storage.

Objects may be selected for system-initiated flushing at object creation time and will continue to be flushed at the end of every interval as long as the object persists and the object manager is running. Flush intervals that are set too small (less than 5 seconds) are discouraged as the amount of traffic that they generate, and the increase of data that must traverse the network, increases dramatically when the object is heavily shared. From a users perspective performance would only decrease slightly when long periods of writing to the same object occur, since the data to be cleaned would only have to be sent one way. The cached page would not need to be updated unless another remote writer joined the foray. When higher reliability is required then the interval may need to be set at a very low number to prompt frequent flushing, effectively trading off the benefits of write caching for less potential data loss.

In the normal sequence of events, kernels that have pages cached with write privileges will eventually allow a client to write to them, causing them to become dirty. At some point in the future, either because the object manager requested it or because the kernel had to evict it, the page will be written back to the object manager. The object manager virtually caches (caches in virtual memory) all object pages that have been referenced and may choose to write data to persistent storage whenever it chooses. Whether it does this often or not will not effect the operation of the server, since all requests may be satisfied from the cached copy of data. To ensure that recently written data is preserved, the object manager immediately writes back dirty pages that it receives from kernels. This will always keep the copy of the data on secondary storage consistent with the cached copy on the server.

3.3.2 Server Reliability

There is no perfect or sure method to guarantee that a server will not crash, but assorted software trap handlers can gracefully halt the server and render some insight that may assist in determining probable causes. More importantly, to maintain functionality to the rest of the system, other servers should be adequately prepared to detect and deal with the situation.

When servers terminate, normally or abnormally, a server on another node will get an error message back from the kernel indicating that the port is dead, or that the holder of the receive right has destroyed it. This allows the server to delete the terminated server from its list of eligible and functioning ones, and discontinue sending messages to that node. It also detects the objects in its object table that are resident on a server that is down for the same reasons, and when a client attempts to retrieve one such object it gets returned to it an error code indicating the unavailability of the object.

When a server is rebooted in the midst of a functioning persistent object system, it must specify a command line switch that manifests its intention. When specified, the server enters a synchronisation mode that involves performing a broadcast to accomplish the following:

- the new server announces to the other nodes that it intends to join the system and gives each node in the system a port send right
- the new server registers its own resident persistent objects with each of the nodes
- each of the servers gives the new server a port send right
- the first able server gives the new server a list of the existing objects in the system to register in its object table

With this initialization protocol in place any server at any registered node in the system may be rebooted safely and reliably, with the consequence that new nodes may be easily and seamlessly integrated into an existing system.

3.3.3 Client and Hardware Failures

When client tasks terminate, the kernel automatically deallocates the client resources, including unmapping any objects that it may have mapped. This unmapping will lead to the generation of a message going to the mapper, or the object manager of the object that was mapped. When the ensuing message reaches the object manager, the object manager may proceed to act just as if the client had explicitly unmapped the object. Dirty pages that had been cached will also be written back, and the server need only administer the unmapping to complete the chain of events.

In the presence of hardware failures, such as node and link downages, the kernel notification mechanisms again come into play. When servers register for notification of certain events like port deaths, they will, in due time (although it is documented as being immediate there are some complications that cause slight delays), be notified by the kernel. This event notification permits the servers to be informed that certain kernels have aborted, or that certain nodes have been disconnected. Once apprised of the circumstances, the server may take similar action to what it would have done in the case described in section 3.3.2 when it had been determined that the remote server had stopped running.

In these instances notification is realised by the kernel, in conjunction with the Mach Network Message Server. Because of the implementation details of the message server there can be a non-trivial delay (tens of seconds) in response time while it is being resolved that the port has in fact died. The delay seems to be proportional to the size of the network, regardless of whether or not the nodes in the network are part of the persistent object system.

3.4 Security

A minimalist view of security was taken for the initial stages of the project, with reflection given to its viability and feasibility in the future. Although the present version of the server has but limited protection mechanisms, it leaves itself open for the integration of sophisticated security measures, and also encourages higher layers to increment existing measures with their own policies.

Many attitudes toward security and file protection have been examined in current systems, including the most dominant one characterized by Amoeba in [Mul90] that used capabilities. A simplified capability-based scheme that uses keys to protect persistent objects from unauthorised access is employed by the persistent object server. Though primitive, it effectively demonstrates the ability of the server to support object safeguarding.

When a persistent object is created the client must supply the server with a 64-bit key that is encoded by the server and stored with the object in the non-volatile storage area. Every subsequent retrieval request must in turn provide the correct object key along with the object name. The server encodes a received object key in a retrieval operation and compares it with the owner-supplied one to grant or deny access.

Two elements of this tactic merit mentioning, since they both relate directly to ways that access may be implicitly granted to a client.

Once a client has access to an object, it may transfer or grant access to other clients as well, as was previously mentioned in section 3.2.2.1, since an object handle is merely a send right for the object port. With the facility offered by the Mach kernel to produce send rights from others, the object key does not need to be transferred when a client decides to

pass object access privileges to a peer. It is only required when retrieval is achieved via the server interface.

A most basic premise in this strategy is that access is an all-or-nothing affair. When a client has access rights, it can perform any and every operation upon the object, including reading and writing the object data. While inappropriate for some applications that may desire the existence of low-level read-only access, the majority of applications can implement their own security features tailored to suit their own requirements.

Chapter 4

IMPLEMENTATION

Modularity and loose coupling in the persistent object server design combine to produce a system that will be easily maintained and enhanced. Just as importantly, system operation and integrity must be addressed to ensure that the functions of the server are executed according to specifications, and that the semantics of the functional requirements are preserved. The implementation details demonstrate how this is achieved and those that are relevant are described in this chapter.

4.1 Interfacing

One of the main goals of the external memory management subsystem was to incorporate into Mach a well-defined and consistent interface that could be imported by any server that desired control of its own external memory facilities. This interfacing principle is dominant in virtually all of the Mach servers, and is adhered to by the persistent object server as well. What follows in this section is a discussion of some of the general issues, and some of the specifics, relating to the server interface.

4.1.1 Mach Interface Generator

A standard Mach tool for specifying server interfaces and making them accessible to clients is invaluable to both parties, and institutes an IPC relationship between the two that

does not demand cognizance of the technical aspects of message-passing. With an uncomplicated and easily imported server interface made available to clients there will be neither hesitation nor hindrance to clients making use of server functionality.

The Mach Interface Generator (MIG) permits a server to define the interface calls and the number and types of parameters in a definitions file which can be compiled by MIG. Compiling the definitions file causes three output files to be generated by MIG. Two files are C files (MIG services a variety of languages, but presently works best with C) that must be compiled, one of which must be linked in with the server and the other linked in to the client. The third file serves as the exporter of the server routines, and is an include file that is to be included by the client. Once the header file has been included, and the client and server modules have been compiled and linked in to the client and server, respectively, the only other major component that must be considered is the server implementation of the routines themselves.

A pertinent feature of MIG that elevates it above a more technical or raw method of client-server interaction is its blanket of simplicity. When interfaces are defined, MIG establishes an outer layer that obscures the IPC primitives from both the client and the server, shielding both sides from the kernel-level calls that must be accompanied by a host of intricate and technical arguments. The communicating parties may then leave the specifics of communication to the generated modules, and accept the default values that were not explicitly overridden by the server in the definitions file.

When interface requirements necessitate sophisticated or specialised communication attributes then MIG allows the specification of such intricacies in the definitions. Designers with the knowledge to do so are therefore offered the option of specifying or overriding MIG default parameters to the Mach messaging calls, protecting against loss of flexibility resulting from the simplistic view of communication extended by MIG interfaces.

4.1.2 Server and Object Interfaces

The server and object interface calls are all handled by the server task, whether it is acting as a server or object manager. Due to the fact that in both cases the call is made by a client and destined for the server, all of the interface definitions may be made within a single definitions file. This circumvents the need for clients to include/link two separate header/object files (one for creating and retrieving, and the other for querying, flushing, and deleting objects).

Server requests involve creation and retrieval, both of which are synchronous in nature (although an argument can be made on behalf of an asynchronous creation alternative). When a synchronous interface call is made on a server the MIG modules on the client and server side enter a message exchange, with a stub executing on each end, as seen in Figure 4.1. The client stub, once it has communicated to the server stub, waits for a reply from the server stub and, since it is running within the client thread context, blocks the client thread from further execution. The server stub eventually calls the server function that implements the interface call, and then replies to the client stub once the server function has returned. In synchronous routines, such as *po_create* and *po_retrieve*, the return statement implicitly implies that the server stub is being returned to, and can respond to the client stub and subsequently resume the client.

The majority of the object interface follows the synchronous RPC style of invocation, with the exception of *po_delete*. When an asynchronous routine is called by a client, the client stub sends a message to the server stub and does not block, or wait for a reply. The value returned by an asynchronous call only represents the kernel-returned result of sending the message to the server stub, and is not indicative of any return code produced by the server. It may even happen that the server has not even had occasion to receive the message from its port at the instant that the client has been returned to. In this

Figure 4.1 Synchronous client-server messaging using MIG

context, the call may be seen as "dropping off" a request for the server to perform some operation (in this case object deletion) whenever the server can manage it, without being notified as to the outcome (success or failure) of the operation. Since object deletion offers situations warranting one or the other form of client-server messaging, both a synchronous *po_deletewait* and an asynchronous *po_delete* are provided in the interface description.

4.1.3 Input and Output Parameters

In a network messaging environment, where clients and servers may be located at different sites, it is vital that the semantics be consistent, keeping in mind the cost this may incur in the network scenario. The Mach IPC paradigm supports both by-value and by-reference semantics, but also allows copy-on-write sharing to reduce the amount of data that must be transported over the network.

All of the input parameters in both the server and the object interfaces are passed by value, since their small sizes do not warrant sharing of virtual memory. Even when strings are passed as input parameters, and pointers are supplied, the string contents are copied by

value to their destination, with a reference to the new string storage supplied to the receiving server.

Output parameters appear normal when furnished by clients, with pointers to local allocated type variables pointing to server-supplied data upon return. In actuality, however, data is being passed by value across the network and copied into the contents of the supplied address. Thus, even passing by-reference implies by-value copying, an unavoidable occurrence in a network predicament. Clearly, though, output parameters are not feasible in an asynchronous message setting, since the client unblocks without giving the server a chance to make return contact.

An output parameter that passes back a port send right requires special treatment. Since the scope of port names within a task is limited to the task itself, it is not possible for a task to simply make a kernel call to create a port right and then pass the port name to another task in an output parameter. In the context of the port namespace of another task the name is invalid. What must happen instead is a kernel-assisted port right generation that occurs during the message transfer, causing the receiving task to receive a new port right, valid in its own portspace, in its returned parameter. Fortunately, this is accomplished through MIG specifications, and is employed in calls where port rights must be returned to the client, e.g., *po_create* and *po_retrieve*.

4.1.4 Internal Interfaces

In some instances, such as object registration, servers must communicate with each other. To accommodate this communication another definitions file was created to define the calls that servers may make to other servers. The calls are all synchronous and both request the execution of a remote operation and require the result of the request in order to continue processing.

4.2 Objects in the Server

Each server must manage its own set of resident objects, and do so in a way that makes efficient use of available memory. Since the benefits of server caching will obviously be manifested in client response times, it is in the interest of the server to cache as much as possible, yet not squander memory by permitting relatively unused or dead objects to deplete memory resources by their very existence. To accommodate the needs of the server object cache, virtual memory is used to store the memory-resident object state structures. The kernel is thus permitted to decide on a demand-driven basis when to relinquish memory from one object and page another in to main memory. Cache space will not be left unused, and only the most frequently-referenced objects will remain in memory.

Figure 4.2 Object finite state machine

4.2.1 Object State

To more easily describe the behaviour of objects, we can define a set of states and transitions between the states, and constrain every object to be in exactly one state at a time. This is sufficient to graphically display their life cycles in the form of a finite state machine (FSM), shown in Figure 4.2. It should be noted that the states shown in the FSM are not real implementation states, but abstract ones that serve to aid in the understanding of objects, and the basic actions that may be of consequence to them.

The states in Figure 4.2 are further specified in Table 4.1. Mention of caching in the state descriptions refers to caching by the server.

NON-EXISTENT:

- no object exists in server or secondary memory

EXISTENT:

- object port created
- object structure in initial state
- no object page data is cached (or allocated)
- disk memory is allocated

ACTIVE:

- one or more pages of object data are cached
- each page follows its own state machine (see 4.2.4)

DELETE_LOCKED:

- page transitions and processing is locked while deletion sequences for each page are initiated based on present page states

PENDING_DELETE:

- zero or more pages have already been deleted
- waiting for ownership of remaining pages

FLUSH_LOCKED:

- page transitions and processing is locked while flushing sequences for each page are initiated based on present page states

PENDING_FLUSH:

- zero or more pages have already been flushed
- waiting for flushing of remaining pages

Table 4.1 Object states

4.2.2 Object Structure

To track the state of an object, and preserve pointers to its data pages, the server allocates an object structure in virtual memory for each object that resides on its node. The object structure is comprised of the fields listed in Table 4.2.

The *obj_name*, *size*, *local*, *time_flushable*, *key*, and *hnode* fields will remain constant over the lifetime of the object, and are stored in non-volatile memory as well as virtual memory. The *obj_port* field contains the port that is used to receive all messages sent to that object, and is filled with a newly created port when the object is created, and again each time the server is booted. Clearly, in the situation when a server goes down, all of the clients that had handles to the objects must re-acquire them once the server has been rebooted, since the original port rights that they had would no longer refer to a valid port.

An array of pointers to the data pages is stored in the *pages* field, and is initialized to null when the server first creates the object, or when it boots. The data pages are then demand-paged from secondary storage, filling in the applicable slot with a virtual memory pointer. The location on disk of the object data pages, as well as the persistent object state can be obtained from the *hnode* field.

The *control_ports* field keeps an updated set of all of the kernels that have at least one local client that has mapped the object. This set is dynamic and will vary according to client mapping durations. When a client on a node maps an object, and is the first client on that node to do so, then a send right that the object manager uses to send XMM protocol messages to the kernel on that node is inserted into the *control_ports* set. When other clients map the object, there is no change by the memory manager or the kernel, since the kernel serves all of its clients from the same local cache. When the last client unmaps the object then the kernel indicates this to the object manager and the kernel port is removed from the *control_ports* set.

lock	mutual exclusion lock to protect the object fields from concurrent access
obj_name	name string of object used for retrieving the object
obj_port	object handle used for sending messages to the object
size	size of the object (in pages)
local	flag indicating whether the object is local or not
time_flushable	flag indicating whether the object is flushed automatically at the end of every flush time interval
key	encoded key used for authentication of clients wishing to retrieve
hnode	index of hnode on secondary storage (disk)
pages_lock	mutual exclusion lock to protect page structure pointers from concurrent access
pages	pointers to object page structures
control_ports	set of kernel cache object ports having one or more clients that have mapped object into their address space
del_cond	condition variable signalled for each page that is deleted
del_count	tally of deleted pages
flush_cond	condition variable signalled for each page that has completed its flush cycle
flush_count	tally of flushed pages
flush_done	condition variable signalled when flush operation has completed

Table 4.2 Object structure

The object *lock* field is used to keep the main object fields consistent in the face of concurrent object structure access. Mutual exclusion to the object structure is guaranteed when the lock is held, but the lock can be released once finer-grained locks have been acquired. The lock protecting the data page pointers (*pages_lock*) is an example of a more fine-grained lock, as well as the locks that exist for each individual data page. The process of locking and lock procurement is explained in greater detail in section 4.3.4.2.

4.2.3 Object Translation

An extravagant feature of Mach tasks grants each task an extremely large (4 gigabytes) and potentially sparse paged virtual address space. Since the object structures are allocated and can exist anywhere in virtual memory, there does not need to be any set pattern of addressing. Object structures are allocated at random addresses, and hence can be seen as "floating" in virtual memory. This eases restrictions of contiguousness induced by tables, and permits virtual memory to be used for other purposes, without regard for location in the address space.

When the object manager receives a message for an object, the only information that it receives is the port by which it is known, or the port to which the message was sent. The object structure must then be examined to process the message. One approach to achieving this translation from object port to structure address is to maintain a table of port-to-address mappings, and looking up the object structure address whenever a message is received. This mapping table is made unnecessary, however, by adroitly choosing a port name for the object when it is created. Since the name of the port may be explicitly specified when the port is created, the object port name may be specified to be precisely the address of the allocated object structure (both being 32-bit quantities). The translation mechanism then becomes a direct addressing operation, with no table look-up and no extra table storage. The only catch to this approach is that a little extra time must be taken to create objects when collisions occur, since there is a faint possibility that a port name that matches the structure address may already be in use, and the structure will have to be freed and reallocated at another location in virtual memory.

4.2.4 Object Data Pages

The data part of each object is divided into medium-grained pages, allowing more sharing to occur within an object than could occur if objects were atomic. If object data can

be requested by kernels on a page-by-page basis then multiple kernels may be able to demand and be granted complete access to different pages, without colliding or having to do so serially.

The page then becomes the finest-grained implementation entity that has dynamic state associated with it. Accordingly, the state of each page within every object is tracked and maintained, and is better understood by the FSM shown in Figure 4.3. The states shown in the FSM comply with the actual implementation (with the exception of the NON-EXISTENT state), and the transitions that cause state changes are included in the message processing routine code. The states are described in Table 4.3.

Figure 4.3 Page finite state machine

NON-EXISTENT:

- page does not exist (because object doesn't)

NOT_CACHED:

- (object and) page slot exists
- page structure not allocated in server
- no kernel has copy

READ:

- server and 0 or more kernels have valid read-only copy

WRITE:

- server has stale copy
- 1 kernel has ownership (valid writable copy)
- no readers or writers are waiting

FLUSHING:

- server and 0 or more readers have valid read-only copy
- flush requests have been sent to all readers
- 1 or more writers are waiting
- next owner is first waiting writer
- 0 or more readers are waiting

FLUSHING_UNLOCK:

- same as FLUSHING first waiting writer needed only access, not copy of data (had a read-only copy)

READWAIT:

- 1 kernel has ownership (valid writable copy)
- clean & flush request has been sent to owner
- 1 or more readers are waiting
- no writers are waiting

WRITEWAIT:

- 1 kernel has ownership (valid writable copy)
- clean & flush request has been sent to owner
- 1 or more writers are waiting
- 0 or more readers are waiting

DELETED:

- clients will not have access to page data
- page structure, data and disk memory still exist

Table 4.3 Page states

The FSM shows that only one writer will ever have ownership of the page at a single time, since in no state do two or more writers own the page. Multiple readers, however, may be reading the page at will, as long as there is no writer. When writing is in progress the read requests are queued and a clean and flush request is sent to the writing

kernel that owns the page. After the owner has cleaned and flushed its copy then an acknowledgement message is sent back to the object manager and the read requests are all satisfied.

4.2.5 Page Structure

Each page gets its own allocated structure that resides in virtual memory and stores the state and pointer to the page data. The structures are only allocated when the page is first accessed (by a request for either a read-only or read-write copy of the page data), at which time virtual memory for the page data is also allocated and cached. Those data pages that are never accessed do not need, or get, page structures allocated for them. Consequently, the object pages that are not referenced have no associated overhead costs.

The page structure is composed of various fields to assist in the maintenance of the state of the page data, and to preserve the integrity and consistency of the data. The fields in the structure are listed in Table 4.4.

lock	mutual exclusion lock to protect the page fields from concurrent access
state	current state of the page
owner	current owner (if any) of the page
data	pointer to page data
reader_set	set of kernels that (would like to) have copies of the page data
flushing_set	set of kernels that have read-only copies of the page and have yet to acknowledge and confirm flushing them
writer_q	queue of write requests (for ownership of the page)
flush_flag	flag to indicate that page has been flushed
hnode	index of hnode on disk that refers to this page

Table 4.4 Page structure

With the exception of the *flush_flag* and *hnode* fields the rest of the fields in the structure are used primarily to keep the caches of all of the nodes coherent. As messages are received the *state* and *owner* fields provide the basis for state transitions to occur, and dictate the action taken (serving or queuing) when requests are received. The state of the page may be any one of the states listed above, while the owner may be any kernel, local or remote, or even the object manager, or the object itself.

The *reader_set* is an unordered set of kernels that have, or have requested, a read-only copy of the data (depending on whether the page is in READ or READWAIT/WRITEWAIT state respectively). This set of kernels must always be known, so that when a write request arrives the reader caches can be invalidated.

The *flushing_set* field contains an unordered set of the kernels that had read-only copies of the page data, and have been sent requests to flush or invalidate their caches. This occurs when the page is in the FLUSHING or FLUSHING_UNLOCK state. Once the acknowledgement has been received confirming that the kernel has indeed flushed its cache of the page then that kernel is removed from the *flushing_set*.

Since only one write request per page can be granted at a time multiple requests are placed on the *writer_q* queue. Although it is not possible to order them in a completely and omnisciently fair manner, i.e., in the true order that they were globally submitted, the requests are placed on the queue in the order that they arrive at the object. Even this approach is subject to scrutiny, since, as will be seen later, the messages may be reordered due to multi-thread message handling.

The locking of the page is accomplished by the mutual exclusion *lock* field, which guarantees single-thread access to the page structure. The *flush_flag* field gets set after the page flushing processing has been completed for the page and the *hnode* field refers to the

2 of/de 2

location of the data page on disk. These three fields will be explained in subsequent sections on locking, flushing and disk management.

4.3 Server Concurrency

To facilitate concurrency, the server must be able to serve a number of service requests in a non-sequential fashion. Acting as both a retrieval server for all objects in the system and an object manager for the node-resident ones, there is potentially much work that it may be requested to perform. The strategy to managing this laborious effort is the key to providing an efficient and productive object server, and plays a critical role in the value and practicality of the persistent object system. In addition, the difficult issues of locking and consistency, must be addressed and attended to.

4.3.1 Server Port Set

The server is multi-threaded, meaning that all of the task resources are shared by all of the threads. Commonality of ports suggests that multiple threads may receive messages from the same ports, and simulates a multi-queue/multi-server model, with the next available server thread able to serve the next request coming from any of the ports.

This is accomplished through the use of a Mach port set. The main server port and the ports of all of the resident objects may be added to the server port set to effect the sharing of requests by all threads. Once the ports are made members of the port set their requests will be received by the waiting server threads, without making any unnecessary assignments of threads to ports.

Each thread begins its life by passing execution control to a special routine called *mach_msg_server*. This routine loops forever, taking the next message from any of the ports in the port set and calling the relevant server function that implements the call. Every

processing thread can therefore receive messages that relate to any of five specific interfaces:

- 1) Client-Server - used by clients for creating or retrieving objects
- 2) Client-Object - used by clients for querying or operating on objects
- 3) Server-Server - used by servers to register and deregister objects and synchronize each other
- 4) Server-Object - used by servers to query the object for key verification
- 5) Kernel-Object - kernel side of XMM used to co-manage the caching of object page data; also used by kernels to notify objects of client and kernel termination

The server function that relates to any one interface call will only get invoked in response to the interface call being issued by one of its respective clients, and the parameters passed to the function must supply enough context to allow the function to be able to process the call in a fitting manner.

4.3.2 Number of Threads

The threads that are created at server startup time are created to be C threads, which are in turn mapped into basic Mach thread entities (see [Loe92b] for more details on how this mapping is achieved and the thread scheduling dynamics). Each thread has a non-trivial stack (default is 64K) and register store that, if dynamically created and destroyed could lead to non-trivial performance hits. On these grounds threads are statically established, the number of which should be indicated by the system manager. In order to achieve maximum parallelism, and still be prudent in the use of system resources, the number of threads should be fine-tuned to suit the number of objects that may reside on a node, or in the system. These threads are also used to receive startup messages between servers, thus the number of threads may also influence the time taken for servers to boot.

To protect against insufficient thread counts, a minimum of two threads is enforced, with the current default set to five (based on testing using lower object counts).

4.3.3 Message Order

Even though Mach ports guarantee that the order in which messages are dequeued or received will be the same as the order in which they were queued, our object server does not make the same promise. Like most multi-threaded servers, it reserves the right to reorder requests.

The first reason that requests are not served in the temporal order in which they were submitted is because of non-uniform distances in the network. Some nodes may be closer in proximity to the object site than others, so it may be possible to issue a request at a moment in time that is later than a request made by another node, but still have it arrive before the earlier request because it has less distance to travel to reach the object. Without a global clock it would be costly to maintain a total temporal ordering, hence the Mach strategy of queuing in the order of arrival suffices.

Figure 4.4 Multi-threaded message reordering

Another layer of message reordering appears at the receiving end. We consider the case in Figure 4.4 in which request message A is queued, and then a later request message B gets queued on the port. If request A is first received by thread X, and then request B is received by thread Y, it would seem that they would be processed in the same order as they arrived. But if thread X gets rescheduled or switched out before it gets a chance to process the message, and thread Y gets switched in then request B will get processed before request A.

To rectify this seemingly unhappy situation, the XMM interface offers sequence numbers to accompany each interface message that is received on the port. The sequence number represents the queuing position of the message on the port. The proposed solution, then, would be to store the sequence number of the most recently completed request, and only process another request if its sequence number is one more than the number stored. This does not help, however, for two reasons. Since user requests may arrive at the same port, but do not carry sequence numbers, there would be holes in the ordering. More importantly, processing a request would infer that the previous request has been completed, reducing a concurrent processing model to a sequential one.

Instead, care has been taken in the handling of messages, so that in cases when a previous message has been assumed to have been received, a condition is checked to ensure that it did in fact get processed. If it did not then the current processing thread suspends itself and waits until the condition is signalled. In most cases client and XMM messages follow single message request-response protocol, so threads may process requests in any order, and concurrency is achieved.

4.3.4 Deadlock

A serious predicament that is prevalent in concurrent systems is when processing ceases, waiting for an event that can never happen (usually because the thread that is

handling the event is itself indefinitely suspended). To avoid this condition of deadlock every conceivable circumstance that could lead to deadlock occurrence must be imagined and averted.

Deadlock, and starvation, its close relative, can only happen when no progress is being made, or when a thread is suspended. One way of finding paths that may lead to deadlock, therefore, is to examine all situations that cause a thread to become suspended. Since being switched out by the scheduler never constitutes starvation under conditions of equal priority, it does not need to be considered as a possible danger.

4.3.4.1 Remote Procedure Calls

A common reason for a thread becoming suspended is a synchronous remote procedure call made by the thread. Once the call is made the client thread is left waiting for its reply. This becomes a realistic danger when all of the server threads are busy serving client requests, each of which makes a remote procedure call to a local object. The difficulty then is that the RPC messages will get sent to a port that is served by the same threads that are waiting for the replies. Since the threads are all blocked waiting for replies there will never be a thread available to receive the RPC message and reply to it in order to unblock the others. This same scenario holds true in a circle of nodes, each waiting for a reply from another node in the circle, none of which has a free thread to receive the message and send the reply.

The first local situation may be avoided by a characteristic in the object table that distinguishes whether an object is local or remote. Whenever a potential for server or object RPC arises, a test is done to determine whether the server is itself, or the object is local, and if so then the local procedure is invoked instead of going through the standard MIG interface. By reducing an RPC message to a local procedure call, the need to use a

port that could potentially block the server thread is eradicated, and the thread will not block.

The distributed version of the problem that involves multiple nodes is less likely, but still may be detected or averted. In this case, a dynamic approach to thread creation may be taken. The number of busy threads may be stored in common server space by requiring each thread to increment the busy count when it gets a message to process. If a thread finds that it is the last free thread then it must create another thread, so that there will always be a thread available to accept messages.

4.3.4.2 Locking

A frequent and more conventional way that threads end up getting suspended is by waiting for a lock that is already held by another thread. If the holder of the lock is the thread itself (it requested the lock without realising that it already had it), or another thread that is in turn waiting for a lock that may be held by the first thread, or even some other thread that is suspended for some reason, then deadlock arises.

A reliable means of insuring that deadlock does not occur due to unsafe acquisition of locks is to impose an ordering on the locks. If all threads know and comply with the ordering when they acquire the locks then there will be no possibility of deadlock due to an improper locking sequence. Delinquent threads are not conceivable since the only users of the locks are server threads.

The relevant locks that need to be ordered are those that pertain directly to objects. While there are others (such as a lock for the object table itself), they are acquired under self-governing circumstances and are independent of object accessing. The locks are described in Figure 4.5 and are to be obtained in order of granularity. The most coarse-grained lock for an object is the object lock described in section 4.2.2. Once this is obtained, the rest of the fields in the structure are accessible for both reading and writing.

Figure 4.5 Locking hierarchy

Typically, when page state is being accessed, the page lock, or the lock that protects all of the page structure pointers (see section 4.2.2), must then be secured. Having this lock guarantees that no other thread will be able to access the page structure table, so that if the page has not yet been cached then two threads will be able to concurrently allocate page structures for the same page. The object lock can then be released, since once the next level of lock has been obtained the previous one is no longer necessary. If it is released then other operations, such as object querying routines, can resume, since they do not need any other locks.

After the pointer to the page structure has been determined, the page lock must then be procured. By releasing the page structure table lock other threads may access the other pages in the object. Acquiring the page lock secures the page structure, but does not imply that the data is owned by the lock-holder. Copies of the page data may exist anywhere in the network, since the other fields in the page structure, combined with the XMM protocol, dictate which nodes have access to the data, and what access rights they have been granted.

A consequential corollary of releasing locks upon acquiring the next level is that upper level state may change at any time after the lock is released. If, for example, a data page has been requested and a thread has weaved its way down to the page structure, holding the page lock, the object lock has been long released so that the object may itself be deleted. Cases such as these must be considered and are discussed in greater detail in subsequent operation sections (see section 4.4).

An unfortunate repercussion of this locking scheme is derived from the same corollary. When two threads contend for the same page one of them will eventually get to it. In fact, the first one to obtain the object lock will also be the first to obtain the page lock, since it will not release its currently-held lock until it obtains the lock at the next level. But the next thread will get to the page structure table lock and hold it, waiting for the page lock to be released. In effect, it is blocking access to all of the object pages until the first thread has finished with the page, and the second can obtain the page lock (releasing the page structures lock). Thus, when there is contention for a single page, there is contention for the entire page set.

4.4 Primary Operations

A practical way to illustrate the operation of the server is to reveal the way that it manages some of the primary client operations that take place. Although these might include all of the client interface calls to the server and object manager, it is not necessary to detail the simplistic operations like querying the object, or retrieving an object handle from the server, since there is no real complexity involved. However, the basic functions of object creation, deletion, and flushing certainly warrant a good deal of explanation, and are delved into in the following sections. Reading and writing object data are two operations that do not take place through an interface, but are actually quite divorced from the server in that they are carried out at the kernel level, and not directly by the object server. The server role in those operations is discussed in section 4.5.

Figure 4.6 Object structure allocation. When port name X is already in use in (a), then (b) is attempted by offsetting the structure in the region. When (c) is reached, without success, then a new region is allocated and the process is restarted.

4.4.1 Creation

The creation of an object is induced by a client request, and is preceded by a lookup in the local object table to see if an object of the same name already exists. If the name is currently in use then an error can be returned immediately, otherwise the creation process may continue. Other hurdles must be crossed, however, before object creation returns successfully.

4.4.1.1 Object Port

By keeping a pointer to the last allocated region of virtual memory, the next address that is free for allocation can be calculated. A virtual memory region that is big enough to store the new object structure may then be allocated. To achieve highly efficient mapping from object port to object structure (discussed in section 3.2.1.2), the object port is then created to have the same port name as the virtual address of the object structure. But since port names are often assigned by the kernel, there exists the unlikely possibility that the port name is already in use, and a first attempt at reconciliation is made.

When a region of virtual memory is allocated, it is done so using the page as the minimum unit, so that only an integral number of pages may be requested. Since the object structure takes much less than a single page, it may be shifted forward within this region by as much as the difference between the size of the region and the length of the object structure. Thus, when the port name that corresponds with a zero offset in the region is in use then the object structure may be assumed to begin at the next offset, and an attempt is made to allocate an object port with that port name. This strategy, shown in Figure 4.6, is pursued until either a vacant port name is found, or until there is no more space in the region in which to offset the object structure.

In the extremely unlikely event that no port name is found to be vacant after the offsetting procedure has been completely exhausted, a more expensive endeavour is undertaken. Since there are no free port names in the existing region, it is deallocated and the cycle of virtual memory region allocation for the object structure begins once again, this time at a different virtual address. Obviously the new region must not overlap with the previous one, else the solution will be no closer than before.

Once the object port has been successfully created, with a port name that coincides with the address of the structure, the port is ready to be made a member of the server port set. Although it does not occur until the end of the creation process, when it finally does become a member, the object will be eligible to receive messages and respond to client requests.

4.4.1.2 Object Registration

After the object structure has been initialised, the other nodes in the network must be informed as to the existence of the newly-created object. In actual fact, the object does not really exist until after the completion of a communication exchange with the other nodes, but at each phase the node must assume that the object exists so that progress can be made.

Simultaneous creation of the same object, or when two or more nodes initiate a creation request to their local servers at or around the same time, introduces the risk that more than one object with the same name may concurrently exist in the network. As this would violate the uniqueness rule, care must be taken to ensure that a situation that might lead to such co-existence is averted. This is achieved by imposing a total ordering of the nodes, rendering the object registration process a sequentially-natured operation.

Before the client gets the object port returned to him the object must have been registered with all of the running nodes in the system. The server simply follows the

sequence that is prescribed by the node ordering by synchronously registering with each node in turn, including its own. Since each node, when requested, registers an object in its own object table in an atomic fashion, using a mutex lock, then no two servers will be successful in registering an object of the same name in the network. As soon as a server encounters a node where the object already exists then it yields to the server that got there before it, and returns an error code to its client.

Under normal conditions, the first server to register on the first node in the ordering will be allowed to successfully create the object, and thwart other servers that may be racing to create. If, however, just after a server has registered an object, the first site in the ordering should go down, and then a second server wishes to create a like-named object, then the second server would start its registration from the second node in the ordering. If the second server was able to register on that node before the first one, then it would be granted permission to create, and the first server would return with an error. In instances like these, where a server has registered on some of the nodes before it has been denied creation permission, a logical course of action might be to deregister from those nodes. This would not be necessary, however, since if a server was able to get ahead of this one in the ordering, then the previous nodes must have gone down, and do not require deregistration.

After successfully registering the object in the network, the server may then proceed to allocate the correct amount of local persistent storage for it. According to whether or not the client specified flushing, the object may also be enqueued on the flushing queue so that it will be automatically flushed at regular timed intervals. After this work has concluded, the server will return a result indicating success to the client, and pass back the object port that will be used as a handle.

4.4.2 Deletion

The task of removing a persistent object from the system is not only considerably complicated but also may require a good deal of effort on the part of the server. In general, all object processing intersects with the process of deletion, since in most cases deletion must be prohibited for some period of time while the object, or a page of the object, is in some non-interruptible state. When an object is to be deleted, it should be assumed that one or more of the object pages may in fact be in a position where it does not expect to be intruded on. Similar safeguards must also, therefore, be taken by the deleting thread so that it does not unfairly interfere with certain operations in progress.

4.4.2.1 Starting Deletion

The object lock that protects the object structure (see section 4.2.2) is the first hurdle that is to be surmounted by a thread servicing a deletion request. Without the procurement of this lock deletion cannot continue, not even to the point where other threads are even aware of the deletion request. After the lock has been obtained, the thread sets a deletion flag within the object to indicate its intent to delete. Once its intent has been manifested, the thread lets go of the object lock. Other threads may then have occasion to enter the object by the object lock, which resembles a gate through which all desiring threads must pass before gaining access. By following the convention that all processing in an object must first be preceded by acquiring the object lock and then checking the delete flag, a thread may be assured that the object is not being deleted before it attempts to service a request for that object.

Conversely, threads engaged in processing do not want deletion to intervene. If the aim of a thread is to get to a page, then the locking path that it follows must guard against permitting deletion before the thread is prepared to surrender to it. The order that locks are

acquired, from coarse to fine-grained, does indeed lock out a deleting thread, until such a time as the individual page has been locked and the integrity of the operation is no longer in jeopardy.

After the deleting thread has released the object lock, it initiates the deregistration phase. Whereas registration was a serial and synchronous procedure, deregistration would initially seem to be well-suited for a parallel asynchronous approach because of the asynchronous client interface. Although there is no need to be wary of clients racing to delete (thanks to the delete flag) the deregistration is done in synchronous mode to ward off a potential create-delete race. The object is deregistered from the nodes in reverse order to preclude the possibility of another being created before the first is deleted.

4.4.2.2 Page Availability

To the object server, the asynchronous deletion call (*po_delete*) is looked upon as a lazy operation, and is processed in a low-priority manner. The synchronous *po_deletewait* version was added as an afterthought, provided as a convenience rather than a basic function. A more important consideration of deletion is to accommodate the in-progress data operations that are not even aware that the object has been deleted.

Reading and writing object page data is realised by clients making reads and writes to cached copies of the data. While these operations occur remotely, the state and nature of the operations is retained at the site of the object manager that directs the activity and movement of the data. It is at this resident site that the object manager can determine whether or not the page is eligible for immediate deletion.

In any multi-paged object there may only be a subset of pages that get referenced. The unreferenced pages are never paged into memory and as a consequence do not have page structures allocated for them. These pages can clearly be deleted without delay.

Pages that have been referenced will be cached by the server and may either be active, but idle (not cached by any kernel), or cached by one or more kernels in either read-only or read/write mode. In the first case, the page may again be deleted without delay. If the second situation applies, then a request is submitted for ownership of the page. It is worth remarking that kernel requests for the page may be queued, and that the deletion request will be serviced following these requests. When the deletion request is finally serviced, then all other subsequent requests for that page will be denied.

With some of the pages deleted promptly, and the rest with outstanding deletion requests, the deletion thread needs some means of ascertaining when all of the pages have in fact been marked as deleted. By maintaining a count of the deleted pages and waiting on a condition variable the correct number of times, the thread that serviced the deletion request will be signalled when each and every page, including the final one, is deleted. Once this has transpired, steps may be taken to destroy the object port, invalidating any handles that a client may have on the object. New clients seeking object handles would have been denied from the time that deregistration had concluded, so that the object could no longer be retrieved in the network.

4.4.2.3 Patient Reclamation

As was stated in the previous section, deletion is a non-hurried operation, and neither is there a requirement for hasty clean-up of object resources. Rather, they can be reclaimed en masse in a practical and non-burdensome manner, and be available for reuse within seconds of deletion. Whether or not the object appears to exist, or is accessible, is of more significance to the process of deletion than whether its related structures have been deallocated. In more succinct terms, if the client has no knowledge of an object and thinks that it is deleted, then it is.

To fulfil the deletion requirement of inaccessibility, the object is destroyed on all nodes that had cached copies by supplicating the kernels of those nodes to terminate all client mappings of the object on their respective nodes. This is achieved through the XMM protocol, and is explained more thoroughly in section 4.5. With no more messages able to be queued, the remaining messages in the object port queue can be quickly processed (processing in these cases will mostly involve returning errors stating that the object has been deleted). After it has been deduced that the port is empty, the object structure is susceptible to being deallocated. To insure that no other messages arrive over the network, perhaps delayed by network latency, the deallocation is performed by an interval thread that runs after the preset interval has expired. The time interval that determines its frequency of execution is user-configurable (actually the same as the auto-flushing interval) and signifies the period of time that the thread sleeps before awaking and conducting its business.

As a very last resort, and to make absolutely certain that no messages will be left unprocessed, an inspection of the port queue is conducted. When it is reported that the port is devoid of messages the thread will proceed to destroy the object port and deallocate the object structure.

4.4.3 Flushing

A client request to flush an object implies that the client has a need for dirty data to be written back to persistent storage. This may occur because a client has realised that it has been placed in a particularly error-prone situation and would like to ensure that object data is preserved in case of failure, or simply because it requires that the object data be written back for clients on other nodes to be able to access it more rapidly. In the majority of cases the client would prefer to be informed as to the completion of the flush, so the flush operation must be serviced synchronously. Plans for an asynchronous variant are in the works for clients that may only initiate flushing for reasons of network bandwidth management, or flush in a controlled manner to avoid network crowding.

Flushing is similar in nature to deletion (substituting deallocation and reclamation of storage space for data write-back). The strategy that was employed to implement flushing mimics, to a large extent, that used when deleting a object in that the object level locks and condition variables play a critical role in the synchronisation and timing. Not surprisingly, therefore, the initial phase involves acquiring the object lock, and then, after confirming that deletion or flushing is not already under way, setting the flag signifying that flushing is in progress.

Before an object can be considered flushed, each and every page of object data must have been updated on secondary storage accordingly. This means that no write operation can have occurred, either locally or remotely, before the time of the flush call, without being reflected in the object store. It is always possible that writes, issued after the time that the flush was requested, may take place before the call has returned to the client and not yet be reflected in the object store. This cannot be averted, since even if it was guaranteed to the time that the reply was sent to the client, there would be no way of determining how long the reply would take to arrive, or how long to lock out writing. For these reasons, if a flush request finds the object already in the midst of processing a previously-issued flush request, it cannot wait for the first one to complete and then simply return to its client. It must recommence the flushing procedure at the conclusion of the previous one.

A more crucial issue surrounding flushing is to ensure that every page does get flushed. This is realised by sequentially inspecting the state of every page and deciding what action, if any, is to be taken. A page that remains uncached by the server can clearly be considered flushed, since the fact that it is not cached implies that no client has written to it.

All other pages are cached by the server, but still may not require that an individual flush action be taken. If, for example, a page is cached by zero or more kernels in read-only mode (equivalent to READ state in Figure 4.3), then the last write operation has

already been updated on disk. Likewise, when the state of a page is **FLUSHING** or **FLUSHING_FOR_UNLOCK**, a write operation is pending, but has not happened yet (flushing in this context means that the cached copies are being flushed or invalidated from the kernel caches). Pages in these states are deemed flushed, and the flushing thread will increment the flush count on behalf of these pages.

Any page that is found to be owned by a kernel, it having read/write permissions for the page, is flagged in its page structure as being flush-pending. If there are no pending read or write requests then a message is sent to the kernel entreating it to write back its dirty data. When the data comes back, then the pending flag can be cleared and the flushed page count can be incremented accordingly. If the kernel has no dirty data then a confirmation message indicating this is returned, so that the page can be counted as being flushed.

A page that is owned by a kernel, but has requests pending, gets its flush flag set, but a message does not need to be dispatched to the owning kernel. The very fact that requests were pending indicates that a write-back request message was already relayed to the kernel, and that the data would be, in due course, written back. The flag will indicate to the thread that processes the returned data that a flush sequence is under way, and that the flushed page tally should be incremented on behalf of that page.

The object manager thread that accepted the client's flush request waits on the flushing condition variable and gets signalled for each page that is flushed, similar to object deletion processing. When the accumulated total of pages has been signalled, then the related variables may be reset and the client is replied to.

The responsibility of system, or automatic flushing, is shouldered by a special thread created and dedicated to this purpose. It embarks on a sleep-wake cycle that causes it to suspend itself for the amount of time specified as the flushing interval, wake up and perform its duties, and resume its dormant state once again.

When objects are created they may be specified as automatically flushed, in which case they will be added to a queue of objects that will be flushed by the server flushing thread. The thread, when it awakens, acts as a client and sends flush messages to each object in the queue, ignoring error codes that may be returned. Once it has done this, it performs a bit of housekeeping before returning to sleep.

We recall from the previous section 4.4.2 that object structure deallocation is done in a lazy fashion, well after the object is deactivated from the system. This deallocation function is performed by the system flushing thread before it goes back to sleep. It identifies the objects to be deallocated by maintaining a queue of dead objects that are enqueued by the deletion thread and dequeued by the flushing thread after it has completed the freeing. After the flusher has completed its reaping, it must not forget to dequeue the dead objects from the flush queue to keep it clean and current, and also avoid an invalid memory reference (the object structure has been deallocated) during its next flushing cycle.

4.5 External Memory Management

With the support of the XMM subsystem, the implementation of distributed shared object data is complex and time-intensive. Without such a facility the project would have been an almost insurmountable venture. As the incarnation of both persistency and distribution, the external memory management concepts and protocol are tightly coupled with object processing, in particular with the operations of reading and writing object data. It is through the well-defined XMM protocol that the server communicates with the local and remote kernels, and maintains coherent page data that is cachable both at the server and client sides.

Much of the page processing is accomplished within the confines of the XMM functions that are provided on the server side. This set of functions are known collectively as the external memory manager, putting yet another face on the persistent object server.

The functions that must be written by the memory manager, or pager, are the XMM protocol messages that the kernel may send to the pager. In actual fact, messages relating to a particular object are sent to the object itself, and not the server, so that if categorizing was an issue this component would be part of the object manager.

Much of the generic abstract memory object protocol is explained in section 2.4, and hence will not be repeated in the sections below. The following sections give brief descriptions of the workings and content of each of XMM routines, insofar as they relate to the persistent object server.

4.5.1 Memory_object_init

To map an object, a client must make a *vm_map* call to the kernel. This call not only achieves the mapping, but also allows the kernel to set up a relationship with the object. This relationship is initiated by the kernel sending the *memory_object_init* (abbrev. *init*) call to the object. An object that receives this message is informed that the kernel desires to provide read and/or write privileges to a client, and will probably send a request for page data at some time in the future.

Before the kernel may request data, or even reply to the mapping client, it must receive a *memory_object_ready* (abbrev. *ready*) message from the pager. The pager sends this message after it has completed its business of establishing the kernel as a legitimate client node. To accurately extend the relationship, the pager adds the received cache control port of the kernel to the set of all kernels that currently have one or more clients mapping the object. This set of kernel control ports is maintained in the object structure and may be used for future detection of faults and failures. Before the ready message is returned, the pager registers the kernel port for notification, so that if the kernel should ever abnormally terminate then port notification would be sent to the pager and cleanup could be initiated.

Once the ready message has been sent, the kernel will return to the client. When subsequent local clients map the object, the kernel does not need to send the init message again as its control port is already a member of the kernel set in the object structure. After all of the local clients have unmapped the object, then the next client to again map the object will cause an init message to once again originate from the kernel.

4.5.2 Memory_object_terminate

When the last client unmaps the object, in a group of one or more clients on a node that had it mapped, the kernel sends a *memory_object_terminate* (abbrev. terminate) message to the object. The object may then remove the kernel control port from its set, and, since it gets the receive rights for both the control and name ports in the message, it may proceed to destroy the ports altogether.

The terminate message may also be generated automatically when a client crashes, if the client happened to be the only client that had the object mapped. If this was the case, then there may still be requests queued for object pages, which are no longer valid and should never be served.

The server may also explicitly invoke the terminate routine if it receives a dead port notification for the kernel. It does this because the notification infers that the kernel terminated abnormally, before it was able to advance the proper messages to the pager.

In order to extract any potentially "left-over" requests resulting from either kernel or client crashes, the terminate routine must examine each page in the object to check for requests or even ownership of a page by the kernel. If it finds any such baggage, then the request is extricated from its containing queue. Likewise when the terminating kernel owns the object, the state may be adjusted to reflect that the kernel no longer owns or possesses a copy of the page data.

4.5.3 **Memory_object_data_request**

When a client references a memory location that is within the range of a memory object that has been mapped into its address space then its local kernel first does a check to see if it has the enclosing page cached. If it does, then the client proceeds just as it would if it had referenced a local variable. If the referenced page was not found to be cached, then the kernel faults on the page and blocks the client while it sends off a *memory_object_data_request* (abbrev. request) message to the pager. Embedded in the request are the details of the request, such as the specific page and access (read or write) being requested.

When the pager receives the request, then it first checks to see if it has the page cached, and if not then it reads the page in from disk. After acquiring the appropriate page lock the pager decides what action to perform, depending upon the current state of the page. As a primary component of the engine that drives the page FSM in Figure 4.3, the request message routine contains much of the state processing that occurs when read or write requests are received. If the page state allows an immediate response to the requesting kernel then the page data will be sent in a *memory_object_data_supply* (abbrev. supply) message. When the page is currently owned by another kernel, and no other kernels are waiting, then the request is just queued. In this case a message is sent to the owning kernel petitioning it to invalidate its cached copy of the page. If requests were already outstanding, then the arriving request is simply queued, since the petition would have already been sent at the time of arrival of the first request that got queued.

When a page is sent to a kernel, it is assumed that the data gets to that kernel without interference or obstruction. This is not always the case, however, since there is always the potential for link or node failure. If the data were to not arrive at its intended destination, then the server would detect the failure either when a new ownership transition

was being applied (a write request came in) or when the object was being automatically flushed. Once the failure has been discovered, then appropriate steps are taken to reclaim ownership without communicating with the downed kernel.

4.5.4 **Memory_object_data_unlock**

A kernel that has a page cached with read-only access may write fault on the page and, instead of making a data request for the page with write permission, ask the pager only for the extra permissions. This is accomplished by sending the pager the *memory_object_data_unlock* (abbrev. *unlock*) message and waiting for a *memory_object_lock_request* (abbrev. *lock_req*) message that indicates that the new access permissions have been granted. This avoids sending the entire page over the network to a kernel that already has a copy cached. This situation can only be realised when no other kernel has made a write request in the interim between the time that the kernel acquired its read-only copy and the time that its unlock message reached the pager.

From the pager perspective, an unlock message is treated similarly to a write request, as they both are essentially requesting ownership of the page. But an unlock message is simplified by the limiting conditions under which it may be received. As it will only be sent by a kernel with a read-only copy of the page the set of possible states that the page may be in when the unlock message arrives will be limited to READ or one of the two FLUSHING states. This renders the processing of the message to be an easier task than that of processing a write request.

A seemingly special case is when the kernel, after making the unlock request, has flushed its cached copy of the page of its own accord. This would make its received permissions useless as it no longer has the data on which to operate. The pager will have no way of knowing this when it grants the access and sets the kernel to be the new owner, so the pager cannot avert the situation. The protocol itself partially solves the problem, by

dictating that the kernel issue a write request for the page under these circumstances. Then, when the request gets to the pager it can notice that the owner is requesting the page and, understanding that the data had been flushed before it could be modified (otherwise it would have been returned for write-back), resupplies the data to the kernel.

If the other case applies, (i.e., another kernel made a write request in the interim) then a race exists between the unlock request arriving at the pager and the resulting flush request that will be sent to the unlocking kernel. The pager can deal with this by ignoring the unlock message when it arrives, knowing that it will be flushing its read-only copy and issuing a write request afterwards.

4.5.5 `Memory_object_data_return`

Any kernel that has a read-write copy of a page may send a *memory_object_data_return* (abbrev. data return) message at any time to write back dirty page data to backing store. The pager is aware of this fact and is prepared to accept and write to disk the page data, as long as it comes from the owning kernel of the page.

In most instances the kernel is deciding that it can no longer cache the page, and so it writes back the page before flushing it from its cache. In other cases the data is being written back in response to a supplication from the pager, which may not demand that the kernel flush its cache, but only clean its copy back to disk. Included in the data return message is the caching state of the page, so that the pager is apprised as to whether the page remains cached or not. If the first situation applies, then the pager may change the state of the page, since without a cached copy the kernel should no longer own the page. The kernel will request the page the next time a fault is taken, again asking for ownership (if the next fault is a write fault).

It may happen that the site at which the owning kernel is located may seem to go out of service due to link failure. When the pager discovers that the node is disconnected from

the system, it will revoke ownership from the node, possibly allowing another kernel owning rights. If the link should then begin to function again, and the previously owning kernel at that time decides to write back its cached copy that is still perceived as being a read-write copy, then there is a potential for stale data being written to backing store. For this reason the pager ignores any data return message originating from a kernel that does not currently own the page.

Pass-by-value semantics in Mach IPC causes a page of virtual memory to be allocated in the address space of the receiving task when there is a chunk of data that is passed in the message. It would then be the responsibility of the receiving task to deallocate this virtual memory when it is finished with the data. By simply readjusting the page data pointer to point to the virtual memory page of the received data, and then deallocating the old data page, bulk data copying can be alleviated.

4.5.6 Memory_object_lock_completed

Before any kernel may own, or be awarded a read-write copy of a page, the pager must ensure that no other kernels have cached copies of the same page. If kernels do, then the pager sends a `lock_req` message that instructs them to invalidate their copies, and if one has a read-write copy that is dirty, to write it back and invalidate the cached copy. After a kernel that has received a `lock_req` message has completed the invalidation (also called flushing) of its copy, then it notifies the pager by sending it a confirmation *memory_object_lock_completed* (abbrev. `lock_completed`) message. This message allows the pager to track the progress of the caching kernels, and when a `lock_completed` message has been received from the last of the kernels the pager may proceed to grant the requesting kernel a read-write copy.

The `lock_completed` confirmation is necessary to guarantee that a client will always see the most recently written data. If, for example, requesting kernel A was granted a copy

before a previously caching kernel B had flushed its copy, then a client running on A might have written to the page, after which a client running on A could read from the same page. If kernel B had not yet flushed its copy then the client on B would read stale data from the kernel cache.

Since the `lock_completed` confirmation is given in response to each `lock_req` message, and because most transitions require confirmation, much of the processing and state transitioning is achieved within the `lock_completed` routine. Together with the request routine it drives the bulk of the page FSM engine.

One message reordering problem that the `lock_completed` routine must deal with occurs when a kernel that currently owns a page receives a `lock_req` message. In response, the kernel should clean its cached copy by sending it to the pager in a data return message, after which it sends a `lock_completed` message. If, because of message reordering due to multiple receiving threads, the `lock_completed` message is received before the data return one then the page will have had time to change owners before the returned data has been saved at the server node (causing the pager to ignore the data to be written back). To prevent this from occurring the `lock_completed` checks the state to see if data should be written back. If a data return is expected then the *page_written* flag and *write_cond* condition in the page structure are used to synchronize these two messages, ensuring that the `lock_completed` routine yields to a data return message.

While waiting for the kernels to flush their read-only cached copies, the `lock_completed` routine keeps track of those who have already done so (sent `lock_completed` messages) by taking them out of the flushing set. When a `lock_completed` message is received from the last kernel in the set then the ownership transition takes place and the first waiting writer in the queue gets the page sent to him in a supply message (if the writer submitted a write request) or just the permissions in a `lock_req` message (if the writer submitted an unlock request).

It may happen that there are multiple would-be writers waiting in the queue by the time the flushing sequence has concluded, or even that one or more readers have made requests in the meantime. Instead of spontaneously reinitiating the flushing sequence, directly after engendering the new owner, the thread first deschedules itself, bestowing to the writer an extra time slice¹ which it can use to write. When the thread gets to run again it sends the `lock_req` message to the owner and activates the sequence that will take ownership away from the current writer and give it to the next waiting writer or readers.

4.5.7 Control Right Disposal

With each of the above-mentioned six XMM protocol messages that is passed to the pager comes another send right to the cache control port of the kernel that sent the message (terminate passes the receive right as well). Because these rights accumulate as time goes on in the execution of the server, they should be appropriately disposed of when necessary. Doing so in each routine, however, would add a system call to the routine overhead, and add an unjustifiable expense to processing.

One method of disposing of these rights would be to let them accumulate, keeping a count of them for each control port and then, when they reach an unacceptable level, calling the `mach_port_mod_refs` system call to destroy them. This would involve keeping a set of control port aggregates each of which would store both the port name and a right count. It is less agreeable than storing simple port elements in the control port set, and could cost more in processing of the control port set than desired.

The alternative that was adopted performed the right disposal probabilistically, costing a single pseudorandom number generation with each use. With a user-defined

¹ The amount of time that will be granted will depend upon the Mach scheduler. The processing thread will yield to the other locally competing threads, allowing the remote node enough time to receive the data and begin processing before the page is demanded back.

probability level the rights are destroyed using two system calls, a *mach_port_get_refs* call to determine how many rights had accumulated, and *mach_port_mod_refs* to destroy the rights. One send right is retained, so that the memory manager will be able to reply to the kernel later on in the routine.

4.6 Secondary Storage

The persistent store maintains the existence of objects in the system after server shutdown, and provides backup storage for object data while the system is active. The actual devices that are used at each site may vary according to their availability, so that if one particular node has only a floppy disk drive then its objects may reside there, while another node may use a 180 Megabyte fixed disk for its own resident objects. Regardless of the device used, the persistent object partition will contain an identical format to all others in the system. A utility program called *mkpo* creates this partition (it must first be formatted by Mach) allowing persistent objects to be created on it.

4.6.1 Super Block

Analogous to a Unix file system a persistent object partition consists first of a super block, specifying the parameters and structures of persistent object storage on that partition. This block is read in when the server initializes itself and is used to compare server versions as well as diagnose the consistency of stored objects. The structure that is stored in this block is composed of the fields shown in Table 4.5.

In the *system_key* field is stored a magic number that uniquely identifies the partition as a persistent object partition. This is used to ensure that device or other errors have not been made and that the partition has been formatted for persistent object usage.

To check for compatible architectures, the size of the page is stipulated in the *page_size* field. This quantity is the unit that is read or written by the server, and stored

contiguously on secondary storage. Even if some architectures do not use the same page size in memory, the size can be faked by the server.

<code>system_key</code>	a magic persistent object system number
<code>page_size</code>	the page size (readable unit) of this partition
<code>hnode_size</code>	size of the hnode structure on this partition
<code>hnode_count</code>	number of hnode structures available on this partition
<code>first_hnode_page</code>	page offset of the first page of hnodes
<code>hnode_page_count</code>	total number of pages containing hnode structures
<code>first_data_page</code>	page offset of the first data page
<code>data_page_count</code>	total number of data pages

Table 4.5 Super block structure

Disk (for the sake of simplicity and practicality, we will henceforth refer to secondary storage as disk) pages are divided into two types, *hnodes* and data. The *hnodes*, or header nodes, are a derivation from Unix *inodes* and are used to allocate data pages for an object. The size of the hnode structure determines the number of hnodes that will be stored in a page, with the total number of hnodes stored in *hnode_count*. The page number of the first hnode page is typically the page succeeding the super block, with *hnode_page_count* - 1 pages contiguously following it. The first data page will follow the last hnode page and there should be a total of *data_page_count* of them. Obviously, if each hnode structure is used to manage one data page then there should be an equal number of hnodes to data pages.

4.6.2 Hnode Table

When a persistent object system partition is created, the available storage space is broken up into the data part, and the hnode part. This is done by taking a precalculated percentage of the total pages and reserving some for hnode structures. With a sufficient number of hnodes reserved for use, every data page will have an assigned hnode to link or allocate it to an object.

The contiguous pages of hnodes form an hnode table that is read in to virtual memory by the server. At object creation time, when backing store is to be allocated for the object, each new data page that is to be allocated is done so by obtaining a new (unused) hnode. By getting assigned an hnode index in the table the object can store the index in its object structure and retain a pointer to the hnode of its first data page. The hnode of the first object page, if there are multiple pages in the object, will contain a pointer to the next hnode, and so on until there is a chain of hnodes, one for each page.

The first time that page data for a particular page is read in by the server, the hnode index of that page is stored in the page structure of the respective page. This will avoid traversing the list for the hnode index of the page and facilitate speedy write-backs to disk.

To simplify the process of finding the disk address of the data page being sought after, and finding out information about that page, the hnode index translates directly into the location of the page relative to the first data page. The data pages can be perceived as being in a table, i.e. a data page table, each entry of which is a page of data. In this light, an hnode with a specific index in the hnode table refers to the data page at precisely the same index in the data page table. The index is consequently just the offset in pages of the referred-to data page from the first data page. The hnode-data page relationship is illustrated in Figure 4.7.

Figure 4.7 Persistent object disk partition

The hnode table is cleaned after each instance of object creation or deletion, since this is when fields in an hnode may change. Only the hnode pages in the table that have been dirtied are written back to disk, except if there are clean hnodes in amongst dirty ones in the table. For example, although the type of fragmentation that normally occurs in file systems cannot occur in the persistent object system (because all allocation is done in uniformly-sized blocks) a variation of fragmentation can arise. If, after a single-paged and then a multiple-paged objects were created, the first was deleted, then if the next object to be created had more than one page the object would straddle the first multiple-paged one on disk. The hnode table would have a single dirty hnode, and then a string of clean ones followed by more dirty ones. Instead of issuing two separate disk write requests, a single

write request will be submitted starting with the first dirty hnode and ending with the last dirty hnode, including the clean portion in the middle.

4.6.3 Hnode Structure

The main function of hnodes is to store the attributes of an object. Every characteristic of an object that must persist is stored in the hnode of the first object page. Hnodes for the remaining object pages have no need to duplicate this information, causing the informational fields to become largely worthless to those pages. The alternative, to create a variant hnode structure for object pages that are not the lead page was not feasible, however, since the amount of disk space that would be exhausted storing hnodes would only increase, not decrease (the same number of lead page hnodes would still have to be created to allow the condition where all objects are single-paged).

The structure of the hnode is therefore uniform for all hnodes, even though only the lead page hnodes will use all of it, and is composed of the fields enumerated in Table 4.6.

obj_name	name string of the object
key	encoded key to access the object
size	number of pages in the object
options	special object attributes
status	whether the hnode is in use
next	hnode index of next page in object

Table 4.6 Hnode structure

The object name, key and size are stored and used when the server is rebooted and the object must be reactivated in the system. The options field encodes characteristics like flushability and scope (whether the object is remotely accessible). The next field contains

the index of the hnode for the next object page, or a null value if no more pages exist in the object.

As the process of allocating a data page essentially equates to finding the next unused hnode in the table, a hint is used to aid in this endeavour. The hint will contain an index that will probably be the lowest free hnode in the table. If it does not contain the lowest free hnode, the index that it contains will be a minimum level from which to start. i.e., the next unused hnode will always be further down in the table somewhere, if it exists. An hnode may be interpreted as used, or unused, by checking its status field bit mask.

Chapter 5

APPLICATIONS

As a core server that performs only the most basic persistent object services, the persistent object server yields fertile ground for the sprouting of a multitude of applications. Whether it be client-server based, or a complete environment, an application server may decide to import the functionality that pertains to its purposes, as well as append still more features to fulfil its mandated functional requirements. This chapter discusses a test application that was completed and presently runs as a peripheral layer to the persistent object server. A number of other plausible and practical applications are suggested and briefly discussed.

5.1 Test Application

To demonstrate the applicability of the server and observe its actual operation from the point of view of a client, an elementary distributed file system was constructed. While a full-blown file system was not produced in its entirety, the fundamental features and attributes of one were mimicked and can be found in the test application. From the perspective of a user, the file system looks and feels like a stream-oriented Unix-like file system, streamlined as it may be.

5.1.1 Interface

In an effort to compare this application with a similar existing one, a Unix interface was placed on the file system. This facilitated equality of functionality so that when it came time to do performance comparisons there was no grey area to obscure the actual semantics achieved. It also acts as a familiar face to those users who might later make use of the application. The limits of the interface extend to *fopen*, *fread*, *fwrite*, *fseek*, and *fclose* function calls, with the same number of accompanying arguments, they being of equivalent type. The way they are called, as well as the return values, all pretend to be Unix.

5.1.2 Opening a File

In actuality, the application server is not a separately executing task or thread, but a module in a library, just like the Unix implementation. A file system call to the test application is therefore just a jump to subroutine instruction as opposed to a message-passing interrupt and a resulting data message. The first thing that must be done by the server is the allocation of a file descriptor. Since the file system routines run as client threads, all of the resources that are exhausted are done so at the clients expense, so clients pay the cost of their own file usage.

The file system must then contact the persistent object server to retrieve the object handle if a "recreate" write mode is not specified, or make a *po_create* call if a suitable write mode is specified. Attempts to retrieve a non-existent object in cases where it should exist (such as read-only mode) return appropriate error codes.

When creating objects, the file system maintains its own secure file system key that it uses for every creation request. In this manner it alone may access the objects that it manages, without interference from other applications. Depending on the desired reliability of the file system, the objects it creates may be system-flushed or not.

The file server does find itself in a sort of dilemma when it must decide the size of the object, as streams do not typically get created to be limited in size. If it chooses to create large objects, then wastage might become rampant. Conversely, when all objects are small, then files are restricted by the statically defined size. What may occur, is that the interface could be slightly altered to accept variable arguments. When *fopen* is called under normal circumstances it creates, by default, small objects, but when an optional additional parameter is included that specifies the maximum size of the stream a sufficiently large object is created.

After successfully retrieving or creating the object, as the case may be, the object is mapped into the address space of the client. This is done behind the back of the client, so that it has no knowledge of the object being mapped in its own virtual space. At this point the file server is ready for subsequent client requests to perform input and output operations on the file.

5.1.3 File I/O

Thanks to the virtual address mapping of the object, reading and writing to a file translates to virtual address referencing. The object server and the kernel take care of all of the background data management, while the file manager only needs to manage the logical file view. The only costly issue that arises in this component is the copying that must take place. Since the client is not aware of the object being mapped, it tends to allocate separate buffer resources for the purposes of reading and writing data. When read and write calls are made, the file server must copy the data from one client (virtual) address to another so that data is transferred to/from the buffer/object.

Seeking becomes a trivially simple operation, since it only demands a new assignment to the logical position of the file pointer in the descriptor. Data access to one

location in the file is as easy as another since the entire object is mapped into virtual memory.

On the issue of read/write access, a compromise was reached. Although the notion of user-owned files could have been included, it would have had to ride on the heels of the Unix server, since Mach does not maintain any view of users that are logged on to the system. It would have also meant the addition of file system initialization, global data, and a needless decrease in performance. Notwithstanding this, the access privileges are controlled on an individual basis.

Managing the access privileges of the client-specified mode is accomplished in two ways. First, when the object is mapped into the client address space, it is done so with the specified access privileges. For example, if a read-only mode was specified, then only read access would be requested on the mapped memory that would represent the object data. A mode that included writing would prompt the file server to map the object in with write permissions. Second, the access field in the descriptor of the open file is updated so that I/O requests that pass through the interface can be legitimately caught without causing the client to trap on a protection violation.

Presently there is no internal locking available for a multi-threaded client to concurrently access the same file using a shared file descriptor, so appropriate locks and synchronisation techniques should be applied by the client. For concurrent access, however, the same file may be opened more than once, so that a new descriptor is created for each instance of the open file, creating the appearance of numerous windows into the one file. The object will be mapped into the client once for each time that the file is opened, but only a single copy of the actual data will be cached by the kernel. Virtual memory may be thus expended, but physical memory usage will remain the same no matter how many times the file is opened. I/O operations can be freely carried out under these conditions without danger of data inconsistency or excess performance penalties.

5.1.4 Closing a File

After a client calls *fclose*, passing in a file descriptor, the object mapping that was created during the *fopen* call that originally allocated the descriptor is unmapped. This will have no effect on any other mapping of either that object or any other object that had been mapped by another *fopen* call. It will cause the termination and deallocation of the descriptor, and the destruction of the object handle that was once stored within the descriptor.

5.1.5 Extensions

Still lacking in the file system is a directory server or method of logically storing files in groups or hierarchies. This can be reckoned with by electing to choose one of three possible options:

- 1) A Unix-like strategy of using files as directories and storing the contents of the directory in the special directory file
- 2) The philosophy of Amoeba and others that separates the file server from the directory server and directs all requests through the directory or name server
- 3) An integration of the directory functionality (somewhat like 1) but using special object table storage techniques ("tries") and hashing of the object pathnames to filter out directory contents

The first option would require mapping many objects into the client's address space (one for each directory name down the path) and will involve a tremendous potential expense of transferring the directory object data across the network for each directory until the desired object is reached. Although this would be feasible and easily achieved, it does not seem to be an attractive choice. The other two options relate more specifically to

modifying the persistent object server than the application, but promise greater efficiency and usability.

A directory server could be either centrally-located or distributed, but if distributed then an added layer of protocol would be required in the object retrieval process. If centralized, then fault tolerance is a concern.

The integration of directories into the object server is a more serious and effectual proposal, and probably would prevail if performance was the decision-maker. The installation of directories into a generic system for the purposes of only a subset of its applications might be treading in dangerous and somewhat untidy waters.

Also missing from the file system are hooks into the operating system that would allow users to issue commands on an interpreter or shell command line. These, though menial and unimportant at the present, would be vital in order to permit convenient file management at the user level.

5.2 Memory-Mapped Files

An adjustment to the test distributed file application would be to supply memory-mapped files to users. In effect, this would be a reduction of both service and overhead from the stream-oriented file paradigm. The logical view of byte streamed files would no longer be available, and the copying to and from buffers could be dispensed with. The service could be easily achieved even without an application-level interface. The object server itself provides enough functionality (minus a directory server) to support distributed memory-mapped files thanks to the manner in which it implements persistent objects. The method by which clients access object data (by mapping it into client address space) confers upon persistent objects the demeanour of memory-mapped files, and promotes behaviour that is in accordance with the operations that would be appropriate in mapped file scenarios.

To provide a clean and satisfactory application, a separate interface would be advantageous. File handles could be integrated so that users need not make direct contact with servers and relay messages in order to open and close files. Access permissions and mapping could be done by the interface much like the distributed file application, with the exception that the client would be aware of the mappings, and be able to designate the addresses at which the files were to be mapped.

5.3 Distributed Databases

The persistent objects exported by the object server are well-suited for database and distributed database applications. Because they are accessible in random order, and can be shared by many clients at different sites, they can be used to store medium and large data records in any type of database scheme, the choice of which can be made by the application.

Objects can be mapped into memory in a variety of ways. If certain database clients access only subportions of the database, then they need only map in the portions that apply to them. This leads to smaller lookup and access tables and quicker hashing and retrieval. Any particular client may also be restricted to viewing data that he has been granted access to, permitting an assortment of customized views, each of which may contain a combination of readable and writable data sections. With the ability to limit accessibility to data, as well as the operations that are allowed on it, a database manager may provide its clients with entire blocks of data records in response to queries or maintenance requests.

The database records may be distributed across a network in any way that may suit the application. By allocating objects at each site to store the data for that site, practical decisions relating to location can be made, so that frequently accessed data can be stored locally or at a closer node than less frequently accessed data. Again, a clever mapping of the objects into client virtual memory can alleviate many problems associated with

accessibility of data. To provide complete location transparency, the database record container objects may be mapped into client contiguous memory. When data records are referenced, they will be automatically retrieved by the kernel from the object server at the site at which they are resident. Similarly, modifications to records may be made without regard to location, and may be written back to their resident node at frequencies that are chosen by the database manager and appropriate to the application.

Objects are shared in page quantities, with the object server conceding separate pages of data to various and different kernels. This degree of sharing allows parallel access to database pages, and concurrent access to the same page of data within the database. A database manager may then embrace a concurrent model of execution, entertaining multiple end-user queries and servicing them without incurring the queuing costs that would normally eventuate from a sequential service model.

5.4 Object-Oriented Environments

Although the object server maintains fairly strict adherence to the performance of its own duties, that of managing distributed persistent objects, its very existence, and the service that it renders, makes it possible for environments that demand object support to come into being. An object-oriented environment, for example, can make excellent use of the untyped persistence offered by the object server.

In Mach, objects are referenced by the port that they may receive messages from. Consistent with this style of object referencing, the persistent object server returns a port send right to clients as they create or attempt to retrieve existing objects. With the port as an object handle, the type of the object the port refers to may differ, yet remain indistinguishable to the client. This polymorphic quality is vital to object-oriented languages, and may be integrated into an environment that would sustain dynamic binding of objects to object types (where the type of the object may not be known until run-time).

An object-oriented system is comprised of a number of objects and the interactions between them. Every object either refers to, or is referred to by one or more other objects in the system, otherwise it could not interact or play a role in the system. Although the embedded object reference problem is not considered in the object server context (but may be solved in an appropriate way by the environment), it may be assumed that most objects will have internal references to other objects. When the state of an object is to be saved, it necessarily involves saving the state of the objects that it may refer to, which may in turn involve saving the state of the objects the other object may refer to, and so on. The logical resolution of this chain is to save the state of all of the objects in the system by utilizing only objects that are persistent (such as those exported by the persistent object server). Only when all objects are persistent can it be determined that the state of the system is equivalent to the one previously saved.

Some other fundamental matters that help define object-orientation are those of classes, instances and inheritance. With every new class must come a class server to accept messages for the classes and also contain the valid protocol for the instances of that class. As the server is distributed, so also is the class, so that the environment contains distributed classes receiving messages originating from any possible site in the system. To encourage and facilitate this, the environment might contain a user package that templates a class server and furnishes the user with a convenient method of creating new classes. An instance may then be constructed by sending a message to its class, which would allocate a new object and initialize it according to its own class. The class must maintain ownership or hold a receive right to each of its instances, so that when messages are sent to it the class server may interpret the messages and directly operate on the object.

Creating a subclass that inherits from its superclass means that all of the class and instance protocol that the superclass implements may be valid for the subclass as well. Since this hierarchy must start (or stop) somewhere, we can see that the most basic object

would be the `DistributedPersistentObject` (named in section 3.1), which implements the most basic of protocol, namely creation, deletion, and so forth. All other classes would fall somewhere beneath this class and would inherit the operations that are already implemented by the main object server class.

Whenever a subclass is created, its superclass would be specified, so that the new subclass might retain a send right to the superclass server. Whenever a class operation is invoked upon the class, the class either overrides the operations of its superclasses or it invokes the one directly above it, and then performs its own object manipulation depending upon what may be appropriate to that class. Instances would do likewise, with each instance of a class being mapped in its class server to a superinstance in its superclass. Instance message handling is thus transferred up the hierarchy whenever transferral applies.

Two aspects of the main persistent object server are under investigation for the benefit of the server itself, but may also assist in the implementation of an object-oriented environment. The first is the ability to dynamically increase the size of an object, so that there may be no constraints on the allowable number of classes in the class hierarchy. The other is to change from string object naming to bitstring identifiers, to facilitate easier object referencing within an environment and permit symbol and object table storage at environment startup time. These two issues are discussed further in chapter seven.

5.5 Distributed Persistent Shared Memory

A slight perturbation of DSM yields Distributed Persistent Shared Memory (DPSM), a realisation of shared memory that persists across applications and system shutdowns. Exploiting the memory-sharing capabilities of DSM, a client can utilize an object as a message drop-off and receiving area to perform message-passing between itself and other clients, as well as other applications. With the persistence aspect affixed,

applications can, upon being restarted, refer to the saved message state of its message area and continue on a processing course that was only briefly interrupted by an application termination.

Shared access to state and data in the confines of DSM renders both system and application-level servers powerful enough in a distributed environment to execute distributed programs, without many of the programming difficulties that are usually attached. Persistence in DPSM saves the shared state and data and can be used as a tool to increase the fault tolerance and reliability of such distributed programs.

Chapter 6

PERFORMANCE EVALUATION

An important aspect of any new system component is an investigation and analysis into the way the component interacts and responds within the system. Although functional testing was conducted on the system to verify its correct operation, its performance must also be evaluated, since an unacceptably slow implementation would be as unacceptable as one with functional faults. Despite the existence of some specialised systems that retain their value, even in the face of unimpressive performance, the vast majority of systems must offer significant efficiency which will translate into high system throughput and low client response times. Only in cases where systems offer special required features, otherwise not found in existing and perhaps better-performing systems, can poor performance be overlooked.

The performance of the persistent object server may best be assessed by observing the response times of some of the common client interactions with the server and its objects. In some instances these operations are similar in nature to those encountered when performing file I/O, thus comparison with an existing network file system is in order. Other operations relate specifically to this server either because the functionality is not available or is inappropriate in the file system context.

6.1 Testbed Environment

To record accurate measurements without interference from external and non-Mach sources, an isolated Mach network was set up. The network consisted of four nodes, three of which contained 25 MHz i386 processors, the other outfitted with a 66 MHz i486 (used as an NFS user file server), connected together with 10 MHz/sec Ethernet. The three i386 nodes were used for measuring while the file server node was used as a workstation with windowing capabilities into the other nodes. Each of the three nodes had a local fixed disk, with only one of them having enough space to support a persistent object partition.

For the purposes of preliminary testing of distributed server functionality, and with distributed resident objects, tests were conducted using the floppy devices as persistent object storage. These tests showed only that servers had operational integrity, since performance was poor due to the huge disk latencies associated with floppy disk drives. During the performance tests, objects were only permitted to reside on the one i386 node that supported an object partition, with the other servers running in diskless mode on the remaining nodes.

The nodes were each equipped with 16 Mb of RAM to be shared by both the operating system and clients. Each measurement was taken after the relevant nodes had been rebooted in order to free superfluous virtual memory and prevent added unnecessary paging. This also caused the main memory cache to be clean and cold at the outset of each test.

The file system tests were done using mounted NFS files on the i386 nodes to maintain an equal comparison base. NFS was chosen as the rival file system not because of any of its relevant features, but mostly because it was available. It also proved to be a suitable operand since it was very well-known and widely used (perceived by many as a de

facto standard), and has undergone a maturation phase that has resulted in many fine-tunings and performance enhancements. It offers nearly standard Unix file semantics from which parallels and contrasts can be made with a number of persistent object applications.

6.2 Creation/Deletion

Although the persistent object server was designed to be scalable, distributed creation was not one of the operations that was optimized for performance. Instead of taking the PORTLANDIA approach and letting servers find the location of an object on a need-to-know or demand basis, distributed persistent objects in Mach notify every other server in the system. This registration translates into reasonably expensive creation costs when the system size is large, but makes for rapid object retrieval in any sized system.

When an NFS directory tree is exported to another node, the importing node is limited to creating or deleting only files in the imported directory tree. The exporting procedure acts as a registration process, except that it is done statically for pre-specified directories at NFS startup time. Creating and deleting a file is actually only a local operation, with the results only visible to those nodes who have been allowed to import a containing directory.

The results of file and object creation and deletion are recorded in Tables 6.1 and 6.2 for network sizes of one to three nodes. Since the files may be visible to any number of nodes that have imported the directory, the creation and deletion costs do not vary in NFS. The persistent object creation delay on a single node compares with NFS, and is equivalent to creating a local unshared object. The dominant cost, as the network increased in size, was the cost of object registration, since the batch object creation seemed to cause a flurry of messages that clogged the network. The delay-per-object for two and three-node systems was observed to markedly drop as the batch size decreased (to 50 objects), whereas larger batch sizes (500 objects) were satisfactory for local creation measurements.

System	Number of Nodes in Network		
	1	2	3
NFS	10.5	10.5	10.5
PO	11.3	54.6	96.8

Table 6.1 File/object creation delay. Creation delay (in ms) of NFS file and persistent object against the number of nodes in the network.

System	Number of Nodes in Network		
	1	2	3
NFS	5.8	5.8	5.8
PO	10.2	49.2	70.8

Table 6.2 File/object deletion delay. Deletion delay (in ms) of NFS file and persistent object against the number of nodes in the network.

6.3 Sequential I/O

The bytestream semantics that is so intrinsic to Unix has a deep-seated place of applicability in some applications. For this reason, tests were performed to determine the effectiveness of mapped persistent objects under conditions of sequential byte access.

A persistent object 1 Mb in length was created and accessed in a sequential manner using client buffers of different sizes: single byte access, 1 Kb, and 64 Kb. The entire object was read by a local client executing on the same node as the server so that network latency did not come into play. The experiment was then repeated a second time without rebooting, causing the cache to be warm, or giving the server a chance to use the data that it

had cached during the first run. The same processes were followed for a local NFS file of equal size. The throughput results for local read and write access, in both the cold and warm caching cases, are shown in Tables 6.3 and 6.4.

System	Cache State	Access Size		
		1 byte	1 K	64 K
NFS	cold	67.46	176.86	179.02
	warm	76.30	752.94	624.39
PO	cold	81.85	81.66	81.85
	warm	354.33	353.10	353.10

Table 6.3 Sequential local read access throughput. Throughput (in Kb/sec) for local NFS and persistent object clients performing sequential read access.

System	Cache State	Access Size		
		1 byte	1 K	64 K
NFS	cold	73.88	581.82	525.13
	warm	74.85	652.23	578.53
PO	cold	83.52	76.76	77.28
	warm	328.21	329.26	331.39

Table 6.4 Sequential local write access throughput. Throughput (in Kb/sec) for local NFS and persistent object clients performing sequential write access.

For sequential single byte access NFS cannot compete with mapped persistent objects. This is due mostly to the library calls that are involved with getting bytes one at a time. As the access size increases to 1 Kb, or even to 64 Kb, the system buffering of NFS produces higher throughputs in both the read and write cases. The object throughput remains fairly consistent across access sizes, and even does not differ by much more than 5 Kb/sec between reading and writing.

In Tables 6.5 and 6.6 the results of doing remote sequential I/O to the persistent object server and NFS file system are displayed. In addition to a warm server cache there is an opportunity for the data to be cached locally (on the client node) which is referred to as the hot cache case. Although NFS caching is referred to as cold and warm, the two are actually a combination of the cold, warm and hot, i.e., a cold NFS cache actually does connote some degree of data being cached in system buffers at the server side, and a warm cache involves data being cached at the client side. Isolation of NFS caching was not achievable to any greater extent, so this approximation suffices. A fair comparison, therefore, may be reached by comparing the best performance of both systems.

System	Cache State	Access Size		
		1 byte	1 K	64 K
NFS	cold	55.32	153.52	155.62
	warm	76.02	742.03	613.17
PO	cold	37.59	38.57	35.48
	warm	59.81	55.77	60.02
	hot	499.51	499.51	499.51

Table 6.5 Sequential remote read access throughput. Throughput (in Kb/sec) for remote NFS and persistent object clients performing sequential read access.

System	Cache State	Access Size		
		1 byte	1 K	64 K
NFS	cold	50.22	118.66	117.16
	warm	52.14	125.34	117.97
PO	cold	35.44	30.92	36.92
	warm	57.50	51.43	59.12
	hot	476.28	474.07	474.07

Table 6.6 Sequential remote write access throughput. Throughput (in Kb/sec) for remote NFS and persistent object clients performing sequential write access.

For read access, once again the object server cannot match NFS throughput in the 1 Kb and 64 Kb cases, although it comes closer (approx. 500 Kb/sec compared to 613 Kb/sec for NFS) in the 64 Kb case than it did in the local tests. In fact, the remote read throughput seems to surpass the local read throughput. This is due to the local caching of the data in the client kernel, a situation not mentioned in the local test results. The same hot caching performance was also observed during the local tests when the kernel was allowed to cache the object, above and beyond the server cache. This was not illustrated in the local cases as it is much more relevant in the network scenarios.

At the outset of testing, I/O latencies were only measured on either side of the I/O operations themselves, and did not include the opening and closing of the file, or mapping and unmapping of the object. But because NFS writes back its data on close (or every 30 seconds), part of its write cost was hidden because it was simply not included in the measurement. This did not have any significance in local writes, but for remote writing a 50% increase in write latency was noted. Its throughput decreased dramatically from reading, whereas persistent objects were written remotely with only a slightly lower

throughput rate than when reading. This is chiefly due to the asynchronous nature of writing back, allowing a client to continue without being blocked by the writeback if more than one client has the object mapped. The object writing bandwidth, through the use of hot caching, manifested a 300-900% increase over that of NFS remote writes.

6.4 Random Access

Many applications have random access needs for persistent data, and require fast access to frequently-referenced portions of a data object. The operations themselves may be random (reading or writing) in addition to the location of the data being referenced. Consideration is given to these applications in this section, as a series of random I/O measurements were taken in both local and remote contexts.

Again, a 1 Mb object was created and used as a data object, on which reads and writes were performed. The location within the object was chosen pseudo-randomly for each operation, and a write probability ratio was fixed for each test. Each test consisted of 1024 operations within the object, with the total I/O delay being measured at the conclusion of the test. The same tests were conducted on NFS files, with the calculated throughputs listed in Table 6.7 for local clients and 6.8 for remote ones. The actual data throughput is slightly higher than what the measurements indicate since the cost of generating the pseudo-random numbers is embedded in the delays that were measured.

The first noticeable detail is that the random access throughput drops considerably below what was measured in the sequential tests. The numbers listed are in bytes/sec, compared to Kb/sec in Tables 6.3 - 6.6.

When no writes were performed, then the cost of seeking played a major role in the low local NFS data throughput rate. The greatest local improvement of persistent objects over NFS files is seen in this case (1151 compared to 160 bytes/sec) and represents over a 700% increase in throughput when the caches are warm. The performance of both systems

was roughly the same across the write ratios when no caching was active, while warm caches allowed objects to achieve higher data bandwidths regardless of the write ratio that was enforced.

System	Cache State	Write Percentage					
		0 %	10 %	25 %	50 %	75 %	100 %
NFS	cold	89	92	99	108	92	125
	warm	160	176	197	265	388	332 - 682
PO	cold	96	90	93	94	95	94
	warm	1151	923	542 - 923	495 - 898	500 - 966	522 - 1191

Table 6.7 Local random access throughput. Throughput (in bytes/sec) for local NFS and persistent object clients performing random access operations with varying write percentages.

System	Cache State	Write Percentage					
		0 %	10 %	25 %	50 %	75 %	100 %
NFS	cold	63	45	36	28	28	44
	warm	176	101	66	40	36	93
PO	cold	35	31	34	34	33	37
	warm	64	51	55	55	58	65
	hot	25600	25600	25600	25600	20480	25600

Table 6.8 Remote random access throughput. Throughput (in bytes/sec) for remote NFS and persistent object clients performing random access operations with varying write percentages.

In some warm cache cases there is a range indicated instead of a single throughput calculation. The upper bound was acquired by repeating the experiment two to three more times, and allowing more read-only pages to be cached as read-write pages. This eliminated more write faults and alleviated messaging with the server that would have been required to secure write permissions for the read-only pages.

When remote random access was examined, the performance of NFS varied to a large extent, but seemed to drop as the write percentage increased (with the exception of the all-write case). As was true for the local tests, the write percentage did not drastically alter the throughput. The throughput was reduced to about 35% of what it was locally when done in a remote environment, compared to the NFS range of 25 - 70% of locally registered throughput.

The most impressive showing that was made by the object server occurred when it provided the client kernel with a local cache copy of the object. In this situation a whopping 20 - 25 Kb/sec was reached, a full three orders of magnitude higher than any other random access case. This performance was achievable because protocol and consistency measures were not considered, since there was no contention for the object (no other clients at other sites were operating on the object).

6.5 Multiple Objects

The latency of a file has been defined in the past (see [How88]) to be the time to open a file, read the first byte, and then close the file. This benchmark was used to see how clients could reference multiple objects and was repeated on both the object and NFS servers. One hundred objects were each retrieved from the server, mapped, the first byte read, and then unmapped before the next object was started. This was first performed by a local client, and then by a remote one, with the file and object latencies calculated and

recorded in Table 6.9. The files and objects were created before the clock was started, so that creation time was not a component of the metric.

System	Cache State	Local	Remote
NFS	cold	13.0	53.5
	warm	3.4	39.2
PO	cold	31.6	195.0
	warm	9.0	159.0
	hot	1.6	38.7

Table 6.9 File latencies. Latencies (in ms) for local and remote NFS and persistent object clients.

Local objects with cold and warm caching were found to be considerably slower than NFS files, by a factor of about three. Interestingly, warm caches caused the performance of both objects and files to increase by about 350%. When the objects were locally cached within the kernel (hot cached) then enormous speedup occurred, producing an object latency less than half that of the NFS file.

In keeping with the other tests, cold remotely accessed objects did not perform well, but when the objects and files were cached as much as possible at the client site, the object latency became slightly better than the latency of an NFS file. The practicality of both local and remote object access was demonstrated when caching is enabled and active. Once again, no contention was present.

6.6 Concurrent Write Sharing

As a highly touted feature, concurrent write sharing provides the facility for multiple clients to all have the same object concurrently mapped in either read or read-write mode. Since this facility is not available in NFS, it was not possible to pit the two against each other. Instead, only the performance of the object server could be evaluated, with the client response times taken to represent the ability of the server to correctly manage the object contention.

In order to contrast the performance of the server in the face of contention, against server performance when there was none, the random access test described in section 6.4 was run concurrently by multiple clients. The clients were divided evenly amongst three nodes, including the node on which the object was resident. In the first test case, a single client executed on each of the nodes, while for the second and third test cases 5 and 40 clients executed on each node, respectively. This generated situations where the total number of clients increased by an order of magnitude with each successive case, from 3 to 15 to 120. The server cache was warmed before each test case.

The difficulty in running the simulation was to simultaneously start all of the contending tasks. As a close approximation, a task was manually launched from the console of each node which spawned off the appropriate number of client tasks. The client tasks then proceeded to retrieve the object from the server, map the object into their address space, and execute the test. The response time was measured starting just before the first I/O operation and ending just after the last one completed. In this way, the time to actually perform client I/O was measured and isolated in order to see the precise effects of contention on the object.

The measured response times were then converted to throughputs (in bytes/sec) for write-read ratios of 0:1, 1:3, and 1:1 and recorded in Table 6.10 for the three cases. The average throughput for each of the clients on a node is given, since they did not vary significantly between each other. For the large-scale test, when a total of 120 clients were contending for the same object, the throughputs did vary, so the range is shown along with the mean or average throughput for all of the clients on that node.

Clients / Node	Client Location	Write Percentage		
		0 %	25 %	50 %
1 Total: 3	Node 1 (Local)	331	131	94
	Node 2 (Remote)	29	9	5
	Node 3 (Remote)	37	10	7
5 Total: 15	Node 1 (Local)	322	132	92
	Node 2 (Remote)	26	9	6
	Node 3 (Remote)	37	9	5
40 Total: 120	Node 1 (Local)	418 - 25600 2926*	398 - 20480 2275*	196 - 12500 1428*
	Node 2 (Remote)	25 - 29 28*	9 - 10 10*	5 - 6 5*
	Node 3 (Remote)	33 - 36 35*	10 - 12 11*	5 - 6 6*

Table 6.10 Throughput with remote contention. Throughput (in bytes/sec) for varying numbers of contending clients performing random access test (see Table 6.8) with differing write percentages (* indicates mean throughput).

Since the object was cached by the server before the tests were conducted, the results for Node 1 clients may be compared against the warm cache case in Table 6.7, since Node 1 clients were local to the object. Likewise the throughputs achieved by remote

clients (on Nodes 2 and 3) correspond to adding contention to the warm cached results of Table 6.8. In this light we can make the following observations:

- both local and remote clients lost considerable throughput when contending with other sites
- the amount of throughput lost increased as the write percentage increased
- there was no noticeable reduction of the mean throughput of remote clients when the total number of clients increased
- as the number of local clients increased so also did the mean throughput of the local clients

The first two points come as no surprise and are quantified in Table 6.11. The meaning of the entry for local clients at 25% write percentage, for example, is that local clients achieved only 24% (or lost 76%) of the throughput achieved when there was no contention from other nodes. The loss of throughput escalated as the write ratio got higher because a read or write to a page requires that the client kernel get either a copy or ownership of the page, potentially having to wait for another site to relinquish ownership. With a higher write ratio, the probability that another site had ownership became higher, causing more instances of waiting to occur.

	Write Percentage		
	0 %	25 %	50 %
Local	71 %	76 %	81 %
Remote	42 %	82 %	87 %

Table 6.11 Loss of throughput. Percentage of throughput lost from local and remote clients due to contention (node 3 clients in Table 6.10 used for remote statistics).

The data throughput remained quite static regardless of the total number of clients due to the balancing of two factors. First, as the number of clients on a node increased, so also did the number of client virtual address faults to the kernel. A fortunate compensation for the extra faults was that there was a higher probability that each of the faults could be satisfied by the kernel, causing the overall number of page requests made by the kernel to remain about the same.

Client response times were seen to decrease dramatically, though, for the 120 client case, or as the number of clients per node ratio was raised. As in the previous cases, the local clients were mostly serviced before the remote requests had a chance to arrive. But in this case more of the object had been cached by the kernel (more clients had made requests) so that the majority of the clients had a hot cache available to them. This is confirmed by comparing the upper end of the ranges in the local Node 1 client row in Table 6.10 with the hot case of Table 6.8. Again, not surprisingly, the hot range did not reach the same peak seen in Table 6.8 as the write ratio was increased.

As a final note, the only way that NFS may have been able to achieve similar semantics of multiple clients concurrently writing to the same file is if the clients were to open the file before each write and close it afterwards. This still would not guarantee consistency in the case of NFS, since locking is currently not enforced by the NFS server, and nor could it be because of its stateless design. In other Unix file systems, though, modifications could be made to accommodate such semantics. To see how expensive this would be, a simple test was performed. The above test was conducted with a single local NFS client (with no contention), with the conditions of opening and closing the file between operations included in the test. Under these ideal conditions the throughput was measured and found to be 72 bytes/sec, less than 22% of what the object clients got even with contention. Further comparison under these circumstances, is not possible, however, as contention is not supported by NFS.

Chapter 7

CONCLUSIONS

As an external server that accomplishes what has often been embedded inside an environment, or the kernel itself, the persistent object server meets the objectives that motivated its conception. It serves as a functional prototype that can be wielded as a basis for a variety of application servers or environments. While offering the option of local non-shared objects, it provides users or client servers the ability to create and operate on distributed network-transparent objects, allowing such servers the opportunity to share the objects amongst themselves, or with other applications. Of equal importance to current object applications is the characteristic of persistency, a property that may be exploited for purposes relating to data storage, reliability, fault tolerance, or other intentions altogether.

The performance of the server, and the objects that it exports, is quite acceptable, and in many instances is very impressive. Nevertheless, like most general-purpose systems, it turned out to be better-suited for some applications more than others. In particular, the servers ability to rapidly access random locations when either reading or writing, combined with the shareability of its individual object pages, lends itself to becoming an attractive database engine, among other random access, or even byte-sequential applications.

When stacked up against an established file system like NFS the persistent object server performs favourably in most of the tests. A critical component relied heavily on by the server is its ability to cache both at the server and client levels. When all possible caching is active and in effect, the persistent objects measure up to or surpass performance of NFS in 90% of all of the I/O tests conducted.

With processor speeds increasing faster than memory bandwidth capacities, and especially faster than secondary storage data transfer rates, it will become more and more beneficial, even essential, to cache data in main memory. As the cost of main memory diminishes, and the amount of RAM resident in hardware increases, RAM-based caching will not only gain more popularity, but also be more viable to cost-caring consumers.

The fusing of external memory object cache demands with those of virtual memory, as realised by the external memory management subsystem, afforded an elegant and seamless way of permitting secondary storage objects to claim as much unused memory as could be acquired. This same luxury was not available to other implementors, such as the designers of Sprite, who seriously considered this option but rejected it on the basis of extra effort and complication. In Mach, the XMM subsystem and interface was already in place, and greatly streamlined the design and implementation of the server.

The server implementation weighed in at just under 10 000 lines of C code (including header files) and took over 9 months, from initial design phases to the completed implementation. It underwent testing and performance evaluations that spanned a two month period.

7.1 Mach Concerns

There are problems, however, with the virtual memory and external memory cache trade-off. They mostly stem from the system-controlled process of main memory allocation. If virtual memory requires more main memory for its own purposes, then

memory that is being used for persistent object caching may be reclaimed. This effectively causes the cache to be paged out, invalidating the benefits of such a cache. In addition, the virtual memory needs of a system may differ dramatically from application to application. A single application, that may have high VM needs, may cause the user-level object server to suffer a severe performance penalty simply because of the transfer of cache ownership between the application and the server.

Although the Mach 3.0 release extended many advantageous features not previously included in version 2.6, its performance has been discouragingly slower than its predecessor. In particular, the Network Message Server that implements network-transparent messaging seemed to execute in slow motion. When combined with the MIG interface generator, and the generated stubs that implement message-passing, a point-to-point null RPC from a client to the pager took over 20 ms. One that returned a page of data took a full 110 ms. With network latencies like these, the Network Message Server must be drastically improved before remote object invocation can be more efficiently implemented. This seemed to have been a slowing factor in many of the remote object I/O performance tests.

A large number of researchers and support people are currently working on the development and enhancement of Mach. Even with all of their efforts, bugs and peculiarities emerged from time to time that thwarted the development of the persistent object server. In almost every facet, Mach is quite stable, but it has not yet reached its complete strength, or achieved full industry status. Given a little more time, Mach will undoubtedly iron out all of its expansion bugs, and mature into a solid system deserving of the praise that has already been heaped upon it by many. Currently under development by the Open Software Foundation (OSF) is the NO Remote Memory Access (NORMA) version of Mach that makes special considerations for distributed systems with no shared

memory. When NORMA Mach emerges it should prove to be a more appropriate base for such distributed applications as the persistent object server.

7.2 Future Work

One area that requires some obvious attention is the creation and deletion policies. As was observed in the performance evaluation, the strategy of contacting each node in the network upon both creation and deletion was grossly inefficient. The large number of broadcast messages that were generated and communicated, combined with the swollen message latencies induced by the Network Message Server across the network, inhibited the scalability of the persistent object system. In its place one might install either a distributed creation protocol, that would contact a partitioned subset of nodes before creating the object, or a directory server, replicated on more than one node for reliability. The directory server is a more popular approach, and would dramatically decrease the creation and deletion delays to compare to those of NFS and other directory-based systems. Although it would lead to improved scalability, it would require further investigation as the potential for bottlenecking would become a critical concern.

String-based object identifiers introduce further communication delay as the entire strings must be transferred between tasks, usually at different sites. It also provokes waste at the disk level, since space must be allocated for the object name. Centralized directory servers could assist in the management of objects, and allow shorter numeric IDs to be passed from task to task. An examination would need to be done in order to weigh the benefits obtained from saved disk storage and shorter network messages, against the extra string-to-UID mapping and centralized bottleneck that would be the result.

Most object systems have facilities to migrate objects from one site to another. This feature does not presently exist in the object server, but may only be done explicitly as a series of client-initiated creation, manual copy, and deletion operations. To be a fully

respected object system. Object migration should be implemented as part of the protocol, with the object exporting an interface call to clients, permitting them to invoke migration upon the object.

At the outset, the object design specifications did not allow objects to increase in size. As the design evolved, object growth became a possibility, but was not added for performance reasons. If enough applications required this attribute, then further steps might be taken toward making objects completely mutable.

Occasions have been known to arise when the hardware that transfers data to and from secondary storage media fails, causing data to be only partially read or written, or not done at all. In these cases it would be valuable to replicate the object, much like what was done in LOCUS [Wal83]. Checkpointing and transaction-based techniques have been explored by PORTLANDIA and others with some degree of success, and have been shown to provide improved fault tolerance and reliability.

Finally, to more fully secure the server, or allow clients only limited access to objects, another server layer was suggested in a previous section. This advice was successfully followed in the distributed file server test application. If the same security is desired at the object server level then another port representing a read-only access privilege could be incorporated into each object. Clients with read-only privileges would be given send rights to the read-only port of the object, and could be differentiated from clients with write access.

7.3 Summary

The persistent object server proved to be a successful project that demonstrated the feasibility of providing a general-purpose system offering sufficient overall performance. Developed and executed in user space, it stands as both an application base, and a foundation for further distributed Mach server development.

REFERENCES

- [Abr89] Abrossimov, V., Rozier, M. and Shapiro, M., "Generic Virtual Memory Management for Operating System Kernels", Proc. 12th Symposium on Operating System Principles, Litchfield Park, AZ, Dec. 1989.
- [Alb91] Albinson, L., Grabas, D., Piovesan, P., Tombroff, M., Tricot, C. and Yassaie, H., "UNIX on a Loosely Coupled Architecture: The CHORUS/MiX Approach", Proc. of the EIT Workshop on Parallel and Distributed Workstation Systems, Florence, Italy, Sept. 1991. (Also Chorus systemes Tech. Rept. CS/TR-91-49.)
- [Atk91] Atkinson, C., Object-oriented reuse, concurrency and distribution: an Ada-based approach, Addison-Wesley, Reading, MA, 1991.
- [Bak91] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shiriff, K.W. and Ousterhout, J.K., "Measurements of a Distributed File System", Proceedings of the 13th Symposium on Operating System Principles, Asilomar, CA, Oct. 1991, pp. 198-212.
- [Bla86] Black, A., Hutchinson, N., Jul, E. and Levy, H., "Object Structure in the Emerald System" Proc. OOPSLA 86, Portland, OR, Oct. 1986, pp. 78-86.
- [Buh89] Buhr, P.A. and Zarnke, C.R., "Addressing in a Persistent Environment", Proc. 3rd Int'l Workshop on Persistent Object Systems, Newcastle, Australia, Jan. 1989, pp. 200-217.
- [Cha89] Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M. and Littlefield, R.J., "The Amber system: Parallel programming on a network of multiprocessors", Proc. of the 12th Symposium on Operating System Principles, Litchfield Park, AZ, Dec. 1989, pp. 147-158.
- [Cha92] Chase, J.S., Levy, H.M., Lazowska, E.D. and Baker, H.M., "Lightweight Shared Objects in a 64-bit Operating System", Proc. OOPSLA '92, Vancouver, Canada, Oct. 1992, pp. 397-413.
- [Che88] Cheriton, D.R., "The V Distributed System", Communications of the ACM, 31(3), Mar. 1988, pp. 314-333.
- [Dan93] Daniels, J. and Cook, S., "Strategies for sharing objects in distributed systems", Journal of Object-Oriented Programming, 6(1), Jan. 1993, pp. 27-36.
- [Das88] Dasgupta, P., Leblanc, R.J. Jr., Ahamad, M. and Ramachandran, U., "The Clouds Distributed Operating System", IEEE Computer, 24(11), Nov. 1991, pp. 34-44.

- [Dix88] Dixon, G.N., Parrington, G.D., Shrivastava, S.K. and Wheeler, S.M., "The Treatment of Persistent Objects in Arjuna", Proc. ECOOP '89, Nottingham, England, July 1988, pp. 169-189.
- [For86] Forin, A., Barrera, J., Young, M. and Rashid, R., "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach", Tech. Rept. CMU-CS-88-165, Carnegie-Mellon University, Aug. 1988.
- [Geh89] Gehringer, E.F., "Name-Based Mapping: Addressing Support for Persistent Objects", Proc. 3rd Int'l Workshop on Persistent Object Systems, Newcastle, Australia, Jan. 1989, pp. 218-232.
- [Gie91] Gien, M., "Next Generation Operating Systems Architecture", Proc. Int'l Workshop on Operating Systems of the 90s and Beyond, Dagstuhl Castle, Germany, July 1991, pp. 227-232.
- [Gol90] Golub, D., Dean, R., Forin, A. and Rashid, R., "Unix as an Application Program", Proc. of Usenix 1990 Summer Conf., June 1990.
- [Gol91] Golub, D.B. and Draves, R.P., "Moving the Default Memory Manager out of the Mach Kernel", Proc. 2nd Machnix Symposium, 1991.
- [Hab90] Habert, S., Mosseri, L. and Abrossimov, V., "COOL: Kernel Support for Object-Oriented Environments", Proc. OOPSLA '90, Ottawa, Canada, Oct. 1990, pp. 269-277.
- [Hei89] Heiler, S. and Blaustein, B., "Generating and Manipulating Identifiers for Heterogeneous, Distributed Objects", Proc. 3rd Int'l Workshop on Persistent Object Systems, Newcastle, Australia, Jan. 1989, pp. 235-247.
- [How88] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, M.A., Satyanarayanan, M., Sidebotham, R.N. and West, M., "Scale and Performance in a Distributed File System", ACM Transactions on Computer Systems, 6(1), Feb. 1988, pp. 51-81.
- [Jon86] Jones, M.B. and Rashid, R., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", Proc. OOPSLA '86, Portland, OR, Oct. 1986, pp. 67-77.
- [Jul89] Julin, D., Network Server Design, Mach Networking Group, Unpublished, Sept. 1989.
- [Laz81] Lazowska, E., Levy, H., Almes, G., Fisher, M., Fowler, R. and Vestal, S., "The Architecture of the Eden System", Proc. 8th Symp. on Oper. Sys. Prin., Asilomar, CA, Dec. 1981, pp. 148-159.
- [Lea83] Leach, P., "The Architecture of an Integrated Local Network", IEEE Journal on Selected Areas in Communication, SAC-1,5, Nov. 1983, pp. 842-856.

- [Li89] Li, K. and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on Computer Systems*, 7(4), Nov. 1989, pp. 321-359.
- [Lis87] Liskov, B., Curtis, D., Johnson, P. and Scheifler, R., "Implementation of Argus", *Proc. 11th Symposium on Operating System Principles*, Austin, TX, Nov. 1987, pp. 111-122.
- [Loe92a] Loepere, K., ed., *Mach 3 Kernel Principles*, Open Software Foundation, 1992.
- [Loe92b] Loepere, K., ed., *Mach 3 Kernel Interfaces*, Open Software Foundation, 1992.
- [Loe92c] Loepere, K., ed., *Mach 3 Server Writer's Guide*, Open Software Foundation, 1992.
- [Loe92d] Loepere, K., ed., *Mach 3 Server Writer's Interfaces*, Open Software Foundation, 1992.
- [Moh91] Mohindra, A. and Ramachandran, U., "A Survey of Distributed Shared Memory in Loosely-coupled Systems", *Tech. Rept. GIT-CC-91/01*, Georgia Institute of Technology, Jan. 1991.
- [Mul90] Mullender, S., Rossum, G. van, Tanenbaum, A., Renesse, R. van and Staveren, H. van, "Amoeba: A Distributed Operating System for the 1990s", *IEEE Computer*, 23(5), May 1990, pp. 44-53.
- [Nel88] Nelson, M.N., Welch, B.B. and Ousterhout, J.K., "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, 6(1), Feb. 1988, pp. 134-154.
- [Ous85] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M. and Thompson, J.G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proc. 10th Symposium on Operating System Principles*, Orcas Island, WA, Dec. 1985, pp. 15-24.
- [Ous88] Ousterhout, J., Cherenon, A., Douglass, F., Nelson, M. and Welch, B., "The Sprite Network Operating System", *IEEE Computer*, 21(2), Feb. 1988, pp. 23-36.
- [Ous90] Ousterhout, J., "Why Aren't Operating Systems Getting Faster as Fast as Hardware?", *Proc. of Usenix 1990 Summer Conf.*, June 1990, pp. 247-256.
- [Pik90] Pike, R., Presotto, D., Thompson, K. and Trickey, H., "Plan 9 from Bell Labs", *Proc. UKUUG 1990 Summer Conf.*, London, England, July 1990, pp. 1-9.
- [Pop85] Popek, G.J. and Walker, B.J., eds., *The LOCUS Distributed System Architecture*, MIT Press, Cambridge, MA, 1985.

- [Por89] Porter, H.H., "Persistence in a Distributed Object Server", Proc. 3rd Int'l Workshop on Persistent Object Systems, Newcastle, Australia, Jan. 1989, pp. 258-271.
- [Qua92] QNX 4.1 Operating System, System Architecture (Preliminary Draft), Quantum Software Systems Ltd., 1992.
- [Ras86] Rashid, R.F., "From RIG to Accent to Mach: The Evolution of a Network Operating System Kernel", Proc. ACM/IEEE Computer Society 1986 Fall Joint Conference, Nov. 1986.
- [Ram88] Ramachandran, U. and Khalidi, Y., "An Implementation of Distributed Shared Memory", Tech. Rept. GIT-ICS-88/50, Georgia Institute of Technology, Dec. 1988.
- [Ren89] Renesse, R. van, Tanenbaum, A.S. and Wilschut, A., "The Design of a High-Performance File Server", Proceedings of the 9th International Conference on Distributed Computing Systems, Newport Beach, CA, 1989, pp. 22-27.
- [Rit74] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, 17(7), July 1974, pp. 365-375.
- [Ros91] Rosenblum, M. and Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System", Proceedings of the 13th Symposium on Operating System Principles, Asilomar, CA, Oct. 1991, pp. 1-15.
- [Roz88] Rozier, M., Abrossmov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W., "CHORUS Distributed Operating Systems", Computing Systems, 1(4), Dec. 1988, pp. 305-370.
- [Rym90] Rymer, J.R., mod., "Panel: OOPSLA Distributed Object Management", Proc. OOPSLA '90, Ottawa, Canada, Oct. 1990, pp. 331-333.
- [San85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., "Design and Implementation of the Sun Network Filesystem", Proc. of Usenix 1985 Summer Conf., Portland, OR, June 1985, pp. 119-130.
- [Sat85] Satyanarayanan, M., Howard, J.H., Nichols, D.A., Sidebotham, R.N., Spector, A.Z. and West, M.J., "The ITC Distributed File System: Principles and Design", Proc. 10th Symposium on Operating System Principles, Orcas Island, WA, Dec. 1985, pp. 35-50.
- [Sch85] Schroeder, M.D., Gifford, D.K. and Needham, R.M., "A Caching File System For a Programmer's Workstation", Proc. 10th Symposium on Operating System Principles, Orcas Island, WA, Dec. 1985, pp. 25-34.
- [Sha89] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M. and Valot, C., "SOS: An object-oriented operating system -- assessment and perspectives", Computing Systems, 2(7), Dec. 1989.
- [Sil88] Silberschatz, A. and Peterson, J., Operating System Concepts, Addison-Wesley, Reading, MA, 1988, ch. 9, 11, and 12.

- [Stu90] Stumm, M. and Zhou, S., "Algorithms Implementing Distributed Shared Memory", *IEEE Computer*, 23(5), May 1990, pp. 54-64.
- [Tan90] Tanenbaum, A.S., Renesse, R. van, Staveren, H. van, Sharp, G.J., Mullender, S.J., Jansen, J. and Rossum, G. van, "Experiences with the Amoeba Distributed Operating System", *Communications of the ACM*, 33(12), Dec. 1990, pp. 46-63.
- [Wal83] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G., "The LOCUS Distributed Operating System", *Proc. 9th Symposium on Operating System Principles*, Bretton Woods, NH, Oct. 1983, pp.49-70.
- [Wri89] Wright, S.L., "The Evolution of the SSE Data Storage System into a Persistent Object System", *Proc. 3rd Int'l Workshop on Persistent Object Systems*, Newcastle, Australia, Jan. 1989, pp. 248-257.
- [You87] Young, M., Tevanian, A. Jr., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D. and Baron, R., "The Duality of Memory and Communication in the Implementation of a Multi-processor Operating System", *Proc. 11th Symposium on Operating System Principles*, Nov. 1987.
- [You89] Young, M.W., *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*, PhD Thesis, Carnegie-Mellon University, Nov. 1989.

END

1 9-07-94

FIN