DISTRIBUTED OBJECT MANAGEMENT IN RAVEN

By

MARCEL SUTANTO

B.A.Sc.(Electrical/Computer Engineering),
University of British Columbia, Canada, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
October 1989
© Marcel Sutanto, 1989
Abstract

*Raven* is an object-oriented distributed system designed to support research in building fault-tolerance distributed systems using an object model. Invocations on Raven objects are executed within a transaction mechanism with provisions for application controlled relaxations. Central to Raven underlying system support is the Object Manager which is responsible for secondary storage management of objects, concurrency control, transaction management, objects distribution and migration, recovery from failures and virtual memory management. Raven objects are encapsulated into object spaces and primary memory is simply used as a cache to object spaces in secondary storage or in remote machines. The physical layout of an object space in primary memory is identical to its layout in secondary storage. An object space represents a unit of concurrency control and recovery in Raven. Raven design integrates virtual memory management, secondary storage management and transaction management into one computation model implemented inside an Object Manager. This allows a highly efficient implementation of the Object Manager. The major work of this thesis is devoted to the design and implementations of the Object Manager in Raven.
Contents

Abstract ii

Contents iii

List of Figures vi

Acknowledgement vii

1 Introduction 1

1.1 Architectural Model 3

1.2 Related Works 6

1.2.1 Clouds 6

1.2.2 Argus 8

1.2.3 TABS/Camelot 9

1.3 Thesis Summary 10

2 System Architecture 12

2.1 Raven Kernel 12

2.1.1 Memory Management 13

2.1.2 Process Management 15

2.1.3 Interprocess Communication 16

2.1.4 Disk Server 17

2.2 Communication Manager 17

2.2.1 Naming of Remote Processes 18

2.2.2 Communication Services 18

2.2.3 An Example of Client-Server Communication 20

2.3 Development Environment 21

3 Object Storage Management 22

3.1 Segments 23

3.2 Object Spaces 25

3.2.1 Physical Layout of Object Space 28
3.2.2 Page Hint Table .................................................. 31
3.2.3 Logical Address of an Instance ............................... 32
3.2.4 Physical Page Layout ........................................... 34
3.3 Object Space Primitives ........................................... 36
3.4 Locating Raven Objects .......................................... 40

4 Page Memory Management ........................................... 44
  4.1 Managing Resident Objects' Pages ............................ 46
  4.2 Data Structures for Memory Management ....................... 47
  4.3 Memory Management without Paging Hardware ............... 50
  4.4 Memory Management with Paging Hardware .................... 53

5 Transaction Management ............................................ 55
  5.1 Modules within the Object Manager .......................... 58
  5.2 Synchronization and Process Structuring ..................... 59
    5.2.1 Data Structures Accessed by Object Manager's Processes 64
  5.3 Concurrency Control ........................................... 65
    5.3.1 Choosing a Concurrency Control Scheme for Raven ....... 66
    5.3.2 Granularity of Locks ...................................... 67
    5.3.3 Lock Management .......................................... 68
    5.3.4 Deadlock Resolution ....................................... 69
    5.3.5 Scheduling Waiting Requests ................................ 72
  5.4 Transaction Invocation and Completion ....................... 74
    5.4.1 Distributed Transactions .................................. 76
    5.4.2 Remote Invocation .......................................... 77
    5.4.3 Remote Mapping ............................................ 78
    5.4.4 Object Space Migration ................................... 79
  5.5 Implementation Details ......................................... 80
    5.5.1 Managing Cached Object Spaces ............................ 80
    5.5.2 Managing Transactions ..................................... 83
    5.5.3 Committing Transaction ..................................... 87
    5.5.4 Managing Locks ............................................ 92
    5.5.5 Managing Object Spaces of Various Attributes ........... 96

6 Recovery Management ............................................... 98
  6.1 Management of Commit Records .................................. 100
  6.2 Recovery Manager's Services .................................... 103
  6.3 Recovering Transactions after a Machine's Failure ......... 105
    6.3.1 Recovering Coordinator Transactions ...................... 107
    6.3.2 Recovering Participant Transactions ...................... 109
  6.4 Updating Segments' BlockMaps During a Transaction Commit 111
  6.5 Managing Object Space Creations and Deletions Across Failures 114
# List of Figures

1.1 Raven Architectural Model .......................... 3
1.2 Division of tasks in Raven ......................... 7

3.1 Layout of segments .................................. 24
3.2 Layout prior to calling Replace() ................. 26
3.3 Layout after calling Replace() ..................... 26
3.4 Layout of a case 1 object space on disk .......... 33
3.5 Layout of a case 2 object space on disk .......... 34
3.6 Physical page layout for instances that fit into a page ... 35
3.7 Locating an instance in an object space of case 2 layout .... 37
3.8 Layout of an object instance that is larger than a page .... 37
3.9 Locating an instance in an object space of case 1 layout .... 42

4.1 Mapping Object Spaces’ Pages ...................... 48
4.2 Memory Management Tables .......................... 51

5.1 Process Configuration in Raven ...................... 62
5.2 A Lock Scheduling Example ......................... 73
5.3 An Osid Control Block ............................. 82
5.4 A Transaction Control Block ....................... 85
5.5 Process interactions during transaction commits ........ 89
5.6 A Lock Table ....................................... 93
5.7 Delta List for time-outs ............................ 94

6.1 Layout of a commit record .......................... 101
6.2 Recovery Processes ................................ 106
6.3 Recovering a Participant Transaction ............... 112
6.4 Layout of blockmaps on disk before the update ........ 115
6.5 Layout of blockmaps on disk after step 1 .......... 115
6.6 Layout of blockmaps on disk after step 2 .......... 116
6.7 Layout of blockmaps on disk after step 3 .......... 116
Acknowledgement

First of all, I would like to thank my supervisor, Dr. Gerald Neufeld, for convincing me to join the Raven project and for his guidance and advice throughout my thesis work. I would also like to thank Dr. Sam Chanson for reading the final draft of the thesis.

Financial supports from the Computer Science department and from my supervisor's research grant are graciously appreciated.

Thanks to Murray Goldberg for supplying the original version of the Raven process threads, to Terry Coatta and Graeme Clark for the long hours of discussion on Raven design, and to Don Acton for proof-reading the final draft of my thesis.

I am much indebted to my parents and my brother and sisters for motivating me to excel and supporting me throughout my academic endeavor. Finally, I would also like to thank Nancy for her emotional support and understanding throughout my thesis works.
Chapter 1

Introduction

Raven is an object-oriented distributed system designed to support research in fault-tolerance distributed systems and object oriented programming. Many of the basic system design concepts used in Raven such as atomic operations, recovery from failures, and location transparency have been used in other object-based systems such as Argus [Lisk82], Eden [Jess82] and Clouds [Dasg85]. These basic concepts are well understood. The challenge has always been how to incorporate all of these concepts into a general purpose distributed system without incurring intolerable overheads in performance and complexity. This thesis presents the design and implementation of the lower layers of the Raven distributed system. The major portion of the work is devoted to the management of Raven objects. We introduce a few salient features to the design of data management in Raven.

Raven object management shares the responsibility of virtual memory management with the kernel, in particular, the handling of page faults. In most conventional data management systems such as database systems or file systems, a page fault and an object fault are handled separately. A page fault is normally handled by the kernel and administrative information regarding virtual memory page frames is hidden within the kernel. An object fault is handled by the data manager in charge of the objects. The data manager is not aware of the actual location of the data pages since paging is done transparently by the kernel. The advantage
of placing virtual memory management within the kernel is that the data manager can be independently built on top of the kernel. However, this often results in double paging. During a transaction commit, modified pages that have been securely written out on disk due to paging have to be read into memory and written out again to a different location on disk. This is due to the fact that the temporary paging storage is not organized the same way as the permanent data storage. If the data manager is solely responsible for the storage organization on disk and is involved in the handling of page faults, thus aware of the location of its virtual pages, double paging can be avoided.

Raven objects are encapsulated into object spaces and primary memory is simply used as a cache to object spaces in secondary storage. In most conventional data management systems, the logical disk page size usually differs from the virtual memory page size and the layout of data stored in disk usually differs from its layout in virtual memory. Consequently, there is an extra level of mapping required as data are read from secondary storage to user's virtual memory space. In Raven, the physical layout of an object space in virtual memory is identical to its layout on disk. This eliminates the extra mapping overheads when an object space is mapped in and out of memory. This is made possible by assigning the logical disk page size to the size of a virtual memory page and paging is handled by the same entity that manages the organization of disk storage.

All operations on Raven objects are encapsulated within a transaction mechanism. Constraints required by the transaction can be relaxed by specifying the attributes of the underlying object spaces that store Raven objects.

Object location is transparent to the users. Users can control the distribution of objects by invoking the primitives provided by the object management system.
1.1 Architectural Model

This section presents an overview of the architecture of Raven System. The architecture can be decomposed into four layers of software organizations. At the lowest layer, layer 1, is the basic Raven kernel. It is a minimal kernel that implements process management, memory management, local communication and I/O device drivers.

Communications between machines is done by processes outside the kernels of the respective machines making use of the network I/O device abstraction provided by the kernel. These processes implement the Communications Manager which is used by layer 2, 3 and 4. The Communications Manager maintains the information about the connectivity of Raven machines, monitors the 'health' of remote Raven machines, and queues outgoing and incoming messages.

Above the kernel is the Object Manager which provides distributed transaction facilities to higher layers. It is responsible for secondary storage management of objects, concurrency
CHAPTER 1. INTRODUCTION

control, transaction management, location transparency of objects, failure recovery and handling of page faults. There is one Object Manager per Raven machine. The Object Manager communicates with its peers in other machines to provide these services. At this level, a Raven object is viewed as a passive object; it is simply an organized block of data whose contents or usage is transparent to the Object Manager. The content can be operation code or actual data manipulated by operation code. However, it is of no concern to the Object Manager. To reiterate, its main responsibility is to map the physical copies of the objects to the higher layer and ensures that any changes to these objects will be consistent in the face of concurrent accesses and machine failures. Each of these services will be further described in subsequent chapters.

Occupying the next layer up are the Invocation Managers and the Class Manager. A Class Manager manages the operation code of classes. Object classes are analogous to object types in other object-oriented systems. Each Raven machine has a Class Manager associated with it. The Class Manager is responsible for the creation of classes, the removal of classes, the addition of new operations into a class, the removal of operations from a class, and the distribution of classes. An Invocation Manager is a group of execution threads provided by the Class Manager to perform the actual operations on data objects of a given class. The Class Manager maintains information regarding its locally active Invocation Managers and communicates with its remote peers regarding the distribution of classes. There may be more than one Invocation Manager per class, for example, if there are more than one way of implementing the same class. The code for the Invocation Manager can be part of the class. A class is also an object as far as the Object Manager is concerned, except its contents happen to be operation code instead of state data. Changes on the operation code are also encapsulated within the transaction mechanism provided by the Object Manager. A user could create a new class, modify the code of a given class, remove a class from the system via the primitives exported by the Class Manager. The Class Manager determines
CHAPTER 1. INTRODUCTION

the distribution and replication of code and data since it is aware of the semantics and the use of these objects. The Object Manager provides the facility to the Class Manager to do so atomically.

Raven objects are logically encapsulated into object spaces. An object space is simply an abstraction of the underlying physical storage and a means to organize Raven objects in secondary storage and in main memory. The grouping of logically related objects into an object space is an attempt to reduce the complexity of managing distributed objects and to improve the performance in general. The physical layout of objects and object spaces are only used within the Object Manager; they are not visible to the higher layers. Central to the storage organization of Raven is the identical layout of object space regardless of whether it is in primary memory or in secondary storage. This organization reduces the mapping overhead and simplifies memory allocations/deallocations. Subsequent chapters elaborate more on the physical organization of Raven objects and the motivation behind this particular organization.

Viewed from the application layer, a Raven object is associated with both state data and operation code. An object is an instance of some abstract data type with a set of exported operations. The exported operations are the only means by which other objects can access and modify the state data of the object. Unlike some object oriented systems such as Argus or Eden where object state data and operations code are managed under the same manager, Raven separates the management of object data from object code. The object data, representing the state of the object, is managed by the Object Manager. The code, which is common to all objects of a given Raven class, is managed by the Class Manager. In this respect, our design approach of separating object state data from code is similar to Clouds' approach [Pitt85]. A modification on the code is done through Class Manager which in turn makes use of the underlying transaction facility of the Object Manager to ensure that the changes to the operations code are consistent. Code for operations is also
stored in an object space. For example, a single object space may contain the code for a particular class. A modification on a class is transformed by the Class Manager into a modification on the object space that contains the operation code of that class. The code and the data on which the code operates do not have to be physically located on the same machine at any moment. Code and data are replicated independent of each other. It is the responsibility of the Class Manager to provide the availability of all code to user when some operations on the class are requested and it is the responsibility of the Object Manager to provide the availability of data needed by the operations. Figure 1.2 shows the division of tasks in a Raven machine.

1.2 Related Works

In the last several years, research in distributed systems has shown a trend towards building object-based instead of file-based distributed systems. The file-based distributed systems could be considered as the precursors to the object-based distributed systems. Many of the basic concepts such as atomicity, location transparency, caching and replication, used in the object-based distributed systems are drawn from the file-based systems. This section considers some of the related object-based distributed systems that have been built in the last few years.

1.2.1 Clouds

Clouds [Dasg85] supports the passive object model at the very low level in the system. These objects are used to build higher system layers and applications. Objects in Clouds are referenced via object capabilities. The capability contains the access rights for certain operations on the object. The physical storage abstraction of Clouds objects are segments. Segments are recoverable, non-recoverable, or volatile. The storage management in Clouds provides mechanisms for mapping segment data in and out of virtual memory, creating
Figure 1.2: Division of tasks in Raven
and destroying segments, modifying segments, and recovering segments after failures. In many respects, Clouds object storage managements are similar to Raven object storage managements. Cloud segments contain operation code, object’s persistent data, or object’s volatile data. Clouds storage manager is unconcerned with how the segments are interpreted by the higher layers. It provides recoverability on the segments through three default operations: precommit, commit and abort. Shadowing is used for recovery.

An object in Clouds is decomposed into two parts: the object type manager and the object instance. The object type manager consists of operation code to manipulate the instance object, and a template to initialize instance object. An object type manager is analogous to our Invocation Manager which is in charge of certain class of object. Object type managers are objects themselves and managed by the kernel Object Manager which is somewhat similar to our Class Manager and Object Manager combined. The kernel Object Manager is responsible for synchronizing accesses to objects, mapping objects in and out of virtual memory, and locating objects. Clouds Object Manager uses locking for synchronizations but also allows users to create their own synchronization schemes. In these respects, it is similar to Raven Object Manager.

Clouds also supports the nested transaction scheme.

1.2.2 Argus

Argus [Oki85] system model is quite different from Raven system model. An application program could view Argus as composed of a network of virtual processing nodes called guardians. A physical node could have more than one guardian. A guardian resides in a single physical node; it cannot span across several physical nodes. A guardian encapsulates and controls the access to one or more resources such as database, mailboxes, and devices. Access to these resources is provided by a set of operations called handlers. Internally, a guardian consists of a collection of data objects and execution threads to manipulate these
objects. The guardian has the sole responsibility and direct access to the objects within the guardian. External clients could only request access via the handler calls, but the actual manipulations are done by the processes inside the guardian. A guardian is also responsible for synchronization of concurrent accesses and consistency of its data in the face of machine failures. To ensure consistency, a computation in Argus is encapsulated into actions. A top-level action starts at some guardian and propagated to other guardians forming nested actions through handler calls. An action in Argus is what is normally referred to as a transaction in other systems. Synchronization is done via locking and time-out is used to resolve deadlock. Unlike Clouds, Argus is a complete object oriented system that provides linguistic constructs for application programmers to use the underlying facilities such as guardians and actions. In this respect, it is similar to Raven which also provides linguistic supports for the application programmers.

1.2.3 TABS/Camelot

TABS [Spec85] is the precursor to Camelot [Spec87]; so in many respects, they are similar systems. We will refer to them here as if they are one system. It provides a distributed transaction facility for user-defined objects. Each physical node has a Transaction Manager which is responsible for coordinating commit procedure. It implements a type-specific locking scheme which allows user to define their own lock modes and synchronization protocol. It relies on time-outs to resolve deadlock. Write-ahead logging is used for recovery instead of shadowing as in Clouds or Raven. TABS is build on top the underlying Accent kernel which provides all the necessary communications and memory mapping services. Camelot is build on top of the Mach kernel which is the successor to the Accent kernel. The transaction facilities in TABS and Camelot are also involved in virtual memory management; they implement a page replacement algorithm and maintain the page buffer pool. In this respect, it is similar to Raven Object Manager.
1.3 Thesis Summary

To summarize, the major work on this thesis is devoted to the design and implementations of the Object Manager. Several design features were implemented in the Object Manager. Notable among them are:

- the integration of virtual memory management and object space management
- the encapsulation of objects into object spaces
- the use of primary memory simply as caches to object spaces in secondary storage thus implementing the single level store concept
- location transparency of objects
- atomic operations and recovery from failures.
- Concurrency control is done via locking with a combination of time-outs and Wait-Die scheme [Rose78] to resolve deadlock.
- Shadowing is used for recovery and the tree structured two-phase commit protocol is used for transaction commits.
- The transaction facility allows a simple nested transaction where the parent transaction is blocked until the child transaction completes the operations.

Chapter 2 describes the implementation environment, and the underlying services provided by the kernel and the Communication Manager. Chapter 3 deals with the physical storage abstraction provided by the Object Manager. It introduces the concept of object spaces, describes the organization of objects into object spaces and explains the motivation behind this particular organization. Chapter 4 describes how Object Manager integrates virtual memory management with object space management and the advantages and disadvantages
of this integration. Chapter 5 is devoted to the design and implementation details of transaction management which form the major part of this thesis work. Chapter 6 deals with the recovery capability provided by the Object Manager. It describes the design and implementation of the recovery schemes. Chapter 7 summarizes the work done on this thesis, and provides some suggestions on future work.
Raven is intended to be an integrated system that provides the user with a complete environment for program development and execution. It is not a set of facilities built on top of an existing operating system although the current implementation is developed under Berkeley 4.3 UNIX. It is a complete distributed operating system that offers the programmers an object-oriented model of computation. Although it differs from the conventional operating systems such as UNIX in its computational model, the underlying services of the kernel are similar. This chapter describes the underlying services provided by Raven kernel and Raven Communication Manager and how these services are accomplished.

2.1 Raven Kernel

In a distributed operating system, resources such as memory, processors and data are distributed among a set of not necessarily identical machines interconnected with each other over a network. This distribution is normally hidden from the users, giving the impression that they are operating on a single, large machine. The major design decision faced by operating system designers has always been the degree to which distribution is to be dealt with in the kernel. A highly functional kernel such as UNIX hides much of the distribution of resources. It is intimately involved with communications protocols, input/output and
resource management. Consequently, it tends to be very complex and hard to modify. Another approach is to relieve the kernel from the knowledge of distribution altogether. The kernel only deals with the resources within a single machine such as process management, memory management and intra-machine communications. Inter-machine communications are handled by processes outside the kernel, making use of the network I/O device abstraction provided by the kernel. The consequence of moving inter-machine communications outside the kernel is that machine-boundaries are now visible outside the kernel. We have chosen the latter approach for Raven kernel. Raven kernel is a local kernel that provides memory management, process control, inter-process communications, and I/O device abstraction in a single machine.

2.1.1 Memory Management

The design of the memory model provided by the kernel is affected by the system portability requirements. Raven should be capable of running on machines without memory-management hardware, but should utilize the hardware when it is available. One of the most significant facilities provided by memory-management hardware is protection. Memory may be divided into units of pages, segments, or address spaces such that access to these units are controlled and protected. To allow for protection on hardware which supports it, we define the notion of an address space.

The memory model provided by the kernel is as follows. Memory consists of a set of address spaces. An address space is a set of pages. Each page in an address space has an associated logical page number. The pages in an address space at any given time are not necessarily contiguously numbered. Two distinct address spaces may share the same physical pages. Each Raven process is associated with an address space. Because an address space represents the boundary of protection, this has some consequences on the local IPC as we will see below.
The following primitives are provided to manipulate address spaces:

- **CreateSpace(var)SpaceId** creates a new address space containing no pages and returns the *spaceid* which uniquely identifies the address space.

- **DeleteSpace(SpaceId)** deletes the address space identified by the given SpaceId. All physical pages in the space are released unless they have been mapped into some other address space.

- **CreatePages(Spaceid, numpages, (var)startpage, (var)pagenum)** creates a logically contiguous block of pages to an address space and allocates physical pages in memory. It returns the starting address of the block and the starting logical page number relative to the address space.

- **AssociatePhysicalPages(Spaceid, numpages, logicalpage, (var)start)** associates physical pages with the logical pages already allocated in the address space. The difference between CreatePages() and AssociatePhysicalPages() is that the former creates new logical pages and allocates physical pages to an address space, while the latter only allocates physical pages and the logical pages already exist in the address space. The former expands the size of the address space while the latter does not. AssociatePhysicalPages() is only called when the current pages of an address space are brought back to RAM due to page faults.

- **DeleteAddrPages(SpaceId, numpages, start)** removes a contiguous number of pages from an address space starting from the page-aligned address value *start*. These pages are removed from the address space, and any pages that are not mapped into some other address space are released to the free pages pool.

- **CopyPages(srcSpace, destSpace, srcAddr, numpages, (var)destAddr)** copies a contiguous block of pages from the source address space to the destination address space.
space and returns the starting address of the new page block in the destination space. It basically creates a block of new pages in the destination space and copies the contents of the source pages to the newly created pages.

- SharePages(srcSpace, destSpace, srcAddr, numPages, (var)pagenum) allows pages of memory to be shared between address spaces. It adds a logically contiguous block of pages to the destination space. These logical pages point to the same physical addresses as the source pages. Multiple SharePages calls allow a given page to be shared between any number of address spaces. The memory associated with the page, including its contents, is retained until the page is removed from all spaces containing it.

- GetLogicalPageNum(SpaceId, start, (var)pagenum) returns the logical page number of the given physical page address relative to the address space.

- Pin(SpaceId, start, numPages) pins the pages with the starting page address given by start. Pinned pages cannot be paged out of memory.

- UnPin(SpaceId, start, numPages) unpins the pages addressed by start. Once the pages are unpinned, they can be paged out of memory.

2.1.2 Process Management

A Raven process is an independently-executing thread of control, which accesses code and data in a specific address space. Several processes can reside in the same address space. A Raven process is identified by process id’s (Pids). Pids have the property that they are reused very infrequently; i.e. once a process has been destroyed, its pid will not be assigned to another process until a very large number of process creations and destructions have occurred.
2.1.3 Interprocess Communication

Interprocess communication within a local machine uses the familiar \textit{Send-Receive-Reply} model. The call \texttt{Send(Pid dest, Datum message, int msglen, (var)Datum *reply)} sends a message to the process \texttt{dest}. The sending process is blocked until the destination process replies to the message; when this occurs, the reply is placed in \texttt{*reply} and \texttt{Send} returns. If the sender's space is different from the receiver's space, the kernel automatically performs \texttt{SharePages()} on the pages that contain the message so that the receiver can access the message buffer. The number of pages to be mapped is determined by the \texttt{msglen} parameter. If the message contains reference pointer to other pages of the sender's address space, the sender is responsible to explicitly do \texttt{SharePages()} on these pages to the receiver's space so that the receiver can dereference the pointer. The kernel is not aware of the contents of the message. The sender and receiver are expected to cooperate in their memory mapping if reference pointers are exchanged. The kernel can only ensure that the pages containing the message buffer are made accessible to the receiver.

To receive a message, a process calls the primitive

\texttt{Receive((var) Pid *sender, (var)Datum *msg, (var) int *msglen)}

If there is already a message waiting to be received by the caller, \texttt{Receive} returns immediately with the message and the sender process id; otherwise, the calling process is blocked until a message addressed to it is sent. The receiver may reply to any of its senders using

\texttt{Reply(Pid sender, Datum reply, int msglen)}

As an alternative to Replying to a message it has received, a process may \texttt{Forward} it. This is done using the call

\texttt{Forward(Pid sender, Pid dest, Datum msg, int msglen)}
which has the effect of redirecting the original Send to process dest. Process dest receives
the message just as though it had been sent to it directly from process sender. In particular,
the responsibility for responding to sender is transferred to dest.

A process may test its input message queue with

\[ \text{MsgWaits((var) Boolean *flag)} \]

which sets *flag to True or False according to whether there are any messages waiting to
be received by the calling process.

2.1.4 Disk Server

A Disk Server process resides in the kernel address space to handle read and write
requests. The disk server process is one of the many interrupt server processes in the kernel.
It accepts requests from other processes and forward the requests to the DiskReqHandler
routine. The DiskReqHandler() routine queues the requests. The process that sends a disk
request is blocked until the request is performed. When a Disk Interrupt occurs (when the
I/O device is ready for reading or writing), the Disk Interrupt Handler routine removes a
request from the queue and unblocks the sender of the request.

At present, a UNIX file is used to simulate a real disk device. The design allows more
than one disk server to be used if more than one disk is available to the kernel. The disk
server has no knowledge of the organization of disk storage. It simply views the disk as an
I/O device for reading and writing.

2.2 Communication Manager

The Communication Manager is responsible for inter-machine communications. The
Communication Manager resides outside the local kernel and makes use of the network
device abstractions provided by the kernel. The current implementation relies on the UNIX
datagram sockets as physical links to remote Raven machines. It provides basic services
sufficient to enable remote processes to communicate in the absence of failures. A very minimal failure handling facility is included. The main services are logical sequencing of messages, correct delivery of messages to the destination process, and buffering of messages if destination process is not ready to receive the message. The Client-Server model of communication is used. Each machine has a number of clients and a number of servers. In order to receive messages from remote processes, a local process must be a server process by registering itself to the Communication Manager. A local process can be both a client and a server process simultaneously. However, in most cases, it usually takes either one of the roles, but not both.

The communication services described here are barely adequate. A more comprehensive communication package needs to be built to handle all possible modes of failures.

2.2.1 Naming of Remote Processes

A remote process is uniquely identified by a pair of addresses: the MachineId and the ProcessId. MachineId is the machine name in which the remote process resides. ProcessId uniquely identifies the given process in the machine. We refer to this pair of addresses as the RPid (Remote Pid) of the process.

2.2.2 Communication Services

The following services are provided by the Communication Manager:

1. **ErrorCode = RegisterServer(char *servername)**

   To receive remote messages, a process must register to the Communication Manager as a server process by providing its symbolic name. Once a process has registered to the communication manager, all remote messages addressed to the process are recognized by the Communication Manager. If the server process is busy, the Communication Manager buffers the incoming messages addressed to the process. A remote message
destined to a local process that has not done RegisterServer() will be rejected since the Communication Manager does not know how to deliver the message.

2. ErrorCode = RemoteLookUp(MachineId destmachine,  
                               char *servername,  
                               RPid *returnedRPid)

   To communicate with a remote server, a client process has to know the RPid of the remote server. To find out the RPid of a remote server, the client invokes RemoteLookUp with the server's symbolic name. The returned RPid is guaranteed to be valid so long as the remote machine and the remote server process are alive.

3. ErrorCode = RemoteSend (RPid destprocess, Datum msg, int len,  
                           Datum *reply)

   sends a message to the remote server process identified by destprocess; msg is the message to be sent and len is the message length; *reply returns the message buffer of the reply. The client is blocked waiting for reply from the remote process and unblocked when a reply message is received. An error code is returned if the destination process does not exist or the destination machine cannot be reached.

4. ErrorCode = RemoteReceive (RPid *remoteclient, Datum *msg, int *len)

   To receive messages from remote clients, a server process invokes this primitive. The order of messages delivered to the server is in the same order as they are received by the Communication Manager. If there is a message in the buffer for the server, the call returns immediately; otherwise, the server is blocked. In order to make this call, a server process must have registered itself via RegisterServer(). This is required so that the Communication Manager knows how to buffer incoming messages addressed to the server.
5. **ErrorCode = RemoteReply (RPid remoteclient, Datum msg, int len)**

This call can only be made by a server process that has received a message from the remoteclient. The call will eventually unblock the remote client when the reply message arrives at the client's machine. The server process is temporarily blocked until an acknowledgement is received indicating that the message has been delivered to the Communication Manager in the client machine. Note that the remote client does not send the acknowledgement; the communication manager in the client machine does. The remote client may have died while waiting for remote reply but it does not affect the server process. An error code is returned if the message cannot be delivered to the remote machine in which the client process resides.

The Communication Manager also monitors the well being of all remote machines. Occasionally, the Communication Manager sends out a probe message to a remote machine to find out the status of remote processes upon which the local processes are blocked.

### 2.2.3 An Example of Client-Server Communication

The following example illustrates how a client-server model of communication can use the communication services described above. Server Beta resides in machine B and client Alpha resides in machine A.

1. Server Beta calls RegisterServer(Beta).
2. Client Alpha calls RemoteLookUp(Beta,...) which returns the RPid of server Beta.
3. Client Alpha can now do a RemoteSend(RPid of Beta, msg, len, (var) reply).
4. Server Beta can now do a RemoteReceive(). Upon receiving the message from client Alpha, the server can decide when to do RemoteReply to client Alpha. Meanwhile, client Alpha is blocked until Server Beta calls RemoteReply (RPid of Alpha, ...).
5. Upon receiving a reply from server Beta, client Alpha resumes execution.
CHAPTER 2. SYSTEM ARCHITECTURE

2.3 Development Environment

The current Raven system is implemented under the Berkeley 4.3 UNIX environment. It represents the first implementation of Raven; as such, it is regarded as a prototype implementation. Since Raven is intended to be an integrated and complete system, not an extension of some conventional operating system, we attempt to minimize the dependence on UNIX system's supports that may make it difficult for future porting to a native (bare) machine. A Raven kernel is a single UNIX process and all Raven processes are simply execution threads within this single UNIX process. The memory management, interprocess communication, and process management are all handled within this UNIX process. A UNIX file is used to simulate a disk storage. UNIX datagram sockets are used to provide links between Raven kernels. The datagram simply functions as a carrier; packet sequencing, message buffering, and addressing are handled by Raven Communication Managers.
Chapter 3

Object Storage Management

The only means by which the higher layers gain access to physical storage is through the object space abstraction provided by the Object Manager. As such, the Object Manager is responsible for storage management. Our design of storage management is driven by the following objectives:

- The number of disk operations as objects are mapped in and out of disk should be minimized. As an object space expands, the number of disk operations required to map an object space to user address space increases. We attempt to minimize the number of disk operations for average sized object spaces.

- Reorganization of physical storage on disk such as space compactions and garbage collection should be transparent to the users.

- During transaction commits, the number of disk writes should also be minimized. If possible, as few as one disk write which can be done atomically.

- The organization should be easily integrated with paging.

- The amounts of memory copy operations should also be minimized.

A three level of storage abstraction is used to organize secondary storage: *pages*, *segments* and *object spaces*. At the lowest level, the disk is simply viewed as a medium consists of
a sequence of \( N \)-byte blocks which we refer to as physical disk pages. \( N \) is normally the maximum size of data the disk device can transfer in one operation. The address of each physical page is simply an offset to a fixed area on disk.

### 3.1 Segments

The next level up, physical pages are organized into *segments* with each *segment* containing a group of logically related pages on disk. At this level, the disk is viewed as a collection of *segments*. A global *Segment Table* maintains all segments allocated in a disk. Each segment has a *segment descriptor* associated with it. A *segment descriptor* is essentially a file descriptor; however, a file descriptor perhaps conveys the notion of a more complicated structure. Therefore, we elect to use the term *segment descriptor*. The descriptor contains two pieces of information: the *segment id* to identify the segment and a *block map* to record which physical pages are allocated for the segment. The *segment id* is simply an index into the global Segment Table. The first segment, with *segment id 0* contains the Segment Table itself. When Segment Table expands, the Object Manager simply allocates additional physical pages for *segment 0*. The first few pages of *segment 0* store the *free page bitmap* in which each bit corresponds to a physical page on disk. The bitmap indicates which physical pages are free.

A *block map* has a fixed number of entries. An entry in the *block map* contains *starting physical address* of a contiguous block of pages and the *size* of the block. When the *block map* runs out of entries to allocate, the Object Manager will attempt to do space compaction by allocating larger contiguous page blocks to merge smaller blocks, thus freeing some of the entries in the *block map*. Figure 3.1 shows the partition of linear disk storage into segment pages. Our storage abstraction bears some resemblance to the *Segment Object* in Clouds [Pitt85].
CHAPTER 3. OBJECT STORAGE MANAGEMENT

Figure 3.1: Layout of segments

The following segment primitives are provided to operate on segments.

- **NewSegment((var)segmentid)** creates a new segment and returns the segmentid to the caller. A segment descriptor is allocated in the Segment Table.

- **DeleteSegment(segment id)** removes the segment descriptor from the Segment Table and releases all the physical pages allocated for the segment.

- **CreateSegmentPage(segment id, n, (var)pagenumber)** allocates n physically contiguous pages for the segment and returns the starting logical page number.

- **DeleteSegmentPage(segment id, n, pagenumber)** removes n logically contiguous
pages from the segment starting from the logical pagename.

- **ReadSegment(segment id, pagenumber, n, (var)buffer)** reads n logical pages starting from the logical pagenumber and returns the data read in buffer; pagenumber is a logical page number which provides an index into the corresponding block map entry.

- **Replace(segmentid1, segmentid2, pgnum1, pgnum2, n)** replaces n physically contiguous pages starting from pgnum1 in segmentid1 with pages starting from pgnum2 in segmentid2. Both pgnum1 and pgnum2 are logical pages relative to the respective segments. The physical pages associated with these logical pages have been previously allocated. As a result of this operation, Segmentid1 now owns the physical pages previously owned by Segmentid2. The old pages of Segmentid1 are returned back to the free page pool. This is done by simply updating the corresponding block map entries in each respective segment; it does not require any data copying. Replace operation may split a block map entry into several entries as shown in Figures 3.2 and 3.3. This operation is required during transaction commits using shadowing scheme.

- **GetSgmDesc(segmentid)** returns the page that contains the segment descriptor of the given segment id.

### 3.2 Object Spaces

We introduce the concept of object space as a storage abstraction for Raven objects. Object instances which share common attributes are stored in an object space. It is worth noting that the Object Manager does not determine how object instances should be grouped in an object space. The higher layers who are aware of the semantics of objects determine the use of the object space. Each object space has an Object Table associated with it.
CHAPTER 3. OBJECT STORAGE MANAGEMENT

Figure 3.2: Layout prior to calling Replace()

Figure 3.3: Layout after calling Replace()
An Object Table maintains the status information of object instances in the object space. Viewed from the Object Manager, an object instance is simply a variable sized data with some attributes associated with it. It has no knowledge of the contents of the instance and its relation with other instances in the same object space. The attributes are used by the Class Manager and the Invocation Manager to reconstruct Raven object in memory. When a new object instance is created, an object descriptor entry is allocated in the Object Table. An object descriptor contains sufficient information about the instance for the Object Manager to maintain its storage and to locate the instance on disk. Object Table and object descriptors are not visible to higher layers.

An object space is identified by a globally unique identifier called OSID. An OSID has two components:

- machineid to identify the host machine where the object space is created.
- serial number: to uniquely identify the object space relative to other object spaces created on the same machine. The serial number is assigned to the new OSID when the object space is created. It is an increasing sequence number that survive reboots. The current implementation uses a time stamp of the object space creation as its serial number.

Although there is a one-to-one correspondent between a segment and an object space, segment id alone cannot be used to identify an object space because of object space migration. An object space may be migrated to a different machine from its original host machine. When it is migrated to a new host machine, a new segment with a new segmentid is allocated for the object space; however, the object space still retains the original OSID assigned to it although its host machine and its segment storage have changed. The original segment storage vacated by the object space may be allocated to a different object space. Consequently, each Object Manager must maintain a directory that maps an OSID to its
current segment. The directory is implemented as one of the special object spaces. Section 3.4 describes the implementation of the directory.

An object instance has a globally unique identifier called ObjRef associated with it. An ObjRef has two fields: an OSID to identifies its object space and an index to index its object descriptor in the Object Table.

An object descriptor contains the following information:

- a Map bit indicates if the object instance has been mapped to memory or not. Map bit is set if the instance has been mapped in.

- a Modified bit indicates if the instance has been modified by the client that maps the object space. Modified bit is set if the instance has been modified.

- the logical disk address gives the location of the instance on disk.

- the RAM address is the address of the instance in memory. It is a virtual memory address in a machine with virtual memory management; otherwise, it is a physical memory address.

- a length field records the size of the instance.

3.2.1 Physical Layout of Object Space

One of the most common performance bottleneck in transaction-oriented data management systems is in accessing data on disk. Consequently, careful thought should be given to minimize the number of disk accesses. Often, disk storage space can be traded off for performance gains. In conventional file systems such as the UNIX file system, the file header or commonly referred to as an inode is placed in a separate disk block from the file data. Typically, there are a number of inodes placed in the same disk block. Consequently,
to access the data, a minimum of two disk accesses are required: one to fetch the *inode* block and another one to fetch the data block. A performance study done on the UNIX file system [Mull84] reveals that about 60 per cent of the UNIX files are less than 2048 bytes. It was suggested that if the data can be placed in the same disk block as its *inode*, then it only requires one disk access to read a small file. This implies that the file system allocates one disk block per *inode* and leaves the remaining space in the disk block to store the data. For larger files exceeding the disk block size, the conventional organization is used. Raven disk storage management follows a similar scheme as suggested in [Mull84].

Central to the storage organization of Raven is the identical layout of object space regardless of whether it is in primary memory or in secondary storage. This organization reduces the mapping overheads and simplifies storage allocations/deallocations. Initially, when an object space is created, it does not contain any instances. A new segment is allocated in the *Segment Table* for the object space. A physical page is allocated for the new segment to store its segment descriptor. For discussion purpose, we refer to the page that stores the segment descriptor as a *segment page*. Recall from Section 3.1 that a number of physically contiguous pages are preallocated for the Segment Table. When `NewSegment()` is invoked, the Object Manager looks for a free physical page for the new segment from these preallocated pages in the Segment Table. If none exists, the Segment Table is extended by allocating additional number of physically contiguous pages.

The segment descriptor does not have a *block map* yet since no data page has been allocated for the segment descriptor except the page to store the segment descriptor itself. And this page belongs to *Segment 0*. We deliberately allocate a disk page to store a segment descriptor although a segment descriptor without the block map only occupies a few bytes of space. This leaves a large free space in the segment page. This remaining space is used to store the object space itself. A logical disk page size is typically 1K to 4K bytes long. The logical disk page size is in turn assigned to match the virtual memory page size so that
there is no reshuffling required when an object space is read from disk. The page is mapped directly to the user's address space.

We expect that on average, the object space can fit into the remaining space of the segment page. This arrangement certainly minimizes the number of disk operations. For example, a transaction mapping an object space that fits into a single segment page only requires one disk access. If the transaction is local and only maps this single object space, no commit record is required during transaction commit since the object space can be written out in place assuming that one disk write can be done atomically. We refer to the object space that fits into the segment page as object space of case 1 layout. Figure 3.4 shows the layout of a case 1 object space.

When an object space expands such that its object instances no longer fit into the segment page or when its Object Table runs out of object descriptors, the whole object space is moved out of the segment page with the exception of the headers as explained below. For simplicity, we refer to this layout as case 2 layout. In case 2 layout, a number of physically contiguous pages are allocated for the Object Table and other pages are allocated for instances. The space vacated in the segment page is now used to store a block map. A page hint table is created; similar to Object Table, a page block is allocated to store a page hint table. The use of the page hint table is described in the next section. The segment descriptor page now contains a block map, an Object Space header, an Object Table header and a page hint table header. The Object Space header contains administrative information such as the attributes of the object space, the date of creation, and the current mapping control mode. The Object Table header contains information about the size of the Object Table and the physical pages that store the table. Page hint table header contains information about the size of the page hint table and the physical pages that store the table. Figure 3.5 shows the layout of a case 2 object space.
3.2.2 Page Hint Table

When a new object instance is created in an object space, one of the parameters supplied to the Object Manager is the context of the instance. A context is simply an attempt to provide locality of reference within an object space. Instances of the same context are grouped together into the same disk page. For the current implementation of Raven, context is simply an integer number to distinguish it from other context numbers in the same object space. The page hint table enables the Object Manager to locate the corresponding page with the right context when a new instance is created. It reduces the number of disk accesses during the creations of new instances. The Object Manager does not need to read in a particular disk page to find out if the page has the right context and enough free space to store a new instance. The entries in the page hint table provide exact information as to which page has the right context and enough free space for a new instance. An entry in the page hint table contains the following information:

- the logical disk page number of the page (4 bytes) which is used as an index into a block map entry,
- the context of the page (2 bytes),
- and the total free space left in the page (2 bytes).

For each new page allocated to store an instance object that can fit into a single page, an entry is created in the page hint table. For instance, an object whose length exceeds one disk page, no entry is created in the page hint table since these pages are owned by a particular instance only.

We trade off storage space for performance gain in the case where instances are created frequently within an object space or when the object space is very large that to find a page with the right context requires many disk accesses. Typically, one disk page is enough to
store a page hint table since each entry only occupies 8 bytes of storage space. A page of size 2048 bytes can accommodate up to 256 instance pages.

3.2.3 Logical Address of an Instance

In case 1 layout, the logical address of an object instance is an offset relative to the beginning of the segment descriptor page. In case 2 layout, the logical address of an instance has two fields: a logical page number and an index. To read an instance of case 2 layout from disk, the Object Manager has to find the corresponding block map entry which contains the physical address of the instance page. The index is then used to find the offset of the instance within the page.
CHAPTER 3. OBJECT STORAGE MANAGEMENT

Global Segment Table

<table>
<thead>
<tr>
<th>Segment Descriptor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved for Segment Table</td>
</tr>
<tr>
<td>1</td>
<td>reserved for Shadow Segment</td>
</tr>
<tr>
<td>2</td>
<td>reserved for Commit Records</td>
</tr>
<tr>
<td>3</td>
<td>reserved for Directory</td>
</tr>
<tr>
<td>4</td>
<td>object space of case 1</td>
</tr>
</tbody>
</table>

Figure 3.4: Layout of a case 1 object space on disk
CHAPTER 3. OBJECT STORAGE MANAGEMENT

3.2.4 Physical Page Layout

This section describes the physical page layout of an object space in case 2 layout. The physical page layout applies to both the virtual memory page and the disk page since an instance page is read directly from disk into virtual memory page without rearrangement. Each instance page has a page header associated with it. A page header contains information to manage the free spaces of the page. It also contains redundant information that enables
The page header contains information such as the OSID that owns the page, the logical disk page number, the context number, the total amount of free space left, the largest size of free block in the page, and the relative address to the first free space block within the page.

If a page contains more than one instance, there is an index area next to the page header. An index area is initially allocated with 16 index entries. Each entry is 4 bytes long and is divided into:
• 2 bytes for offset relative to the beginning of the page,

• and 2 bytes for a reverse pointer which is simply an index to the corresponding object descriptor in the Object Table.

Recall that the logical address of an instance contains an index used to lookup an entry in the index area. Actually, the sole purpose of an index area is to facilitate compaction within a page when there are many "holes" in a page due to object deletions. The Object Manager can do space compaction within the page without changing the logical disk address by simply updating the index area. Thus, only the affected page needs to be updated on disk. The ease of space compaction comes at the costs of some space overhead and an extra level of indirection. Figure 3.6 shows the organization of instances within a page. Figure 3.7 shows how an object instance residing in a case 2 object space is located given its ObjRef.

If an object instance is larger than a page, then the first page of the instance has a page header. The remaining pages only contain instance data with no page header. The pages allocated for the instance are physically contiguous in memory, and logically contiguous on disk. In this case, a page header contains four pieces of information: the OSID that owns the page, the disk page number, the reverse pointer and the instance length. Figure 3.8 shows how an instance exceeding a page is stored in the object space.

Our storage organization trades off storage space for simplicity of mapping operations. We expect that in most cases the average instance length is much less than a page size.

3.3 Object Space Primitives

Object Manager provides a set of primitives to the higher layer. The services are provided following the client server model. The Object Manager functions as a server and the higher layer is the client sending requests to the server through the stub routines. The
CHAPTER 3. OBJECT STORAGE MANAGEMENT

via Directory map

Segment X

OSID index

ObjRef

indexing an entry in the OT

block map entry maps logical disk page number to physical location on disk

point to index entry

physical instance page

point to the actual physical instance on disk

Figure 3.7: Locating an instance in an object space of case 2 layout

n physically contiguous pages

page header

OSID | logical disk page number | reverse ptr | instance length

Figure 3.8: Layout of an object instance that is larger than a page
client is blocked until the service is performed or an error occurs. The primitives must be invoked within a transaction. The transaction simply encapsulates and identifies a series of operations done on behalf of a particular client. It does not necessarily enforce all the consistency constraints, namely, *serializability, failure atomicity and permanence* of a transaction. The Object Manager decides what services are required within a transaction according to the attributes of the object spaces manipulated by the transaction. Chapter 5 elaborates more on how these services are provided and implemented inside Object Manager.

The following primitives are provided for object space operations:

- **CreateObjSpace**(transId, attributes, (var)OSID, (var)osidHandler) creates a new object space with the given attributes and returns the new OSID and handler to the client. The new object space created is automatically mapped to client’s address space and the handler is used as an object space volatile identifier for further operations on the object space in virtual memory. See Chapter 5 for discussion on transId.

- **MapObjSpace**(transId, OSID, accessMode, (var) osidHandler) maps the object space identified by OSID to a client’s address space. The access mode is for synchronization with respect to other concurrently executing clients. The access mode can be either Read or Write. If access mode is Read, the object space may be read shared with other concurrently active clients. If the access mode is Write, the client has an exclusive access to the object space until the transaction under which this mapping is done is committed or aborted. Conceptually, a MapObjSpace operation is like an Open File operation.

- **DeleteObjSpace**(transId, OSID) permanently removes the object space from the system.
• ControlObjSpace(transId, OSID, controlMode) to set control modes on the object space with regards to the distribution of the object space. The control mode determines if the object space can be migrated, remote-mapped, or remote-invoked.

The following primitives are provided for individual instance operations. To invoke these primitives, an object space must have already been mapped into the client’s address space via MapObjSpace() above. As will be shown in Chapter 5, these instance operations do not always have to be performed by the Object Manager. In most cases, they can be executed under the client’s execution thread without communicating with the Object Manager. This reduces the amount of messages exchanged between the clients and the Object Manager's server.

• CreateInstance(transId, osidHandler, context,
       (var) objRef, (var) memoryAddress)
creates a new object instance in the object space identified by osidHandler and returns the new objRef of the instance and the virtual address of the instance in memory.

• DeleteInstance(transId, osidHandler, objRef) permanently removes the object instance from the object space.

• GetInstance(transId, osidHandler, objRef, accessMode,
       (var) memoryAddress)
returns the memory address of the instance. The address is a virtual address for a machine with virtual memory management; otherwise, it is the actual physical address of the instance in RAM. The accessMode could be read or write. A write access mode indicates that the particular instance will be modified, i.e., the Modified
bit in the instance's Object Descriptor is set. This individual access mode for each
GetInstance operation on an object space should not be in conflict with the general
map mode on the object space which was set when the object space was mapped to
the client's space via MapObjectSpace( OSID, mapmode,...) invocation. For example,
a client that maps object space for READ only could not invoke a GetInstance(...,
write) on the object space; it could only invoke a GetInstance(..., read) on the
object space.

3.4 Locating Raven Objects

Location of object spaces are transparent to the higher layers. The Object Manager
is responsible for locating an object space given its OSID. To accomplish that, each Raven
Object Manager maintains a cache table of OSIDs and their corresponding locations. We
refer to this table as a directory. It is a cache table instead of a complete naming system
since it may not contain the most up-to-date information about the location of the object
spaces. In Raven, there is no centralized name server. The Object Managers cooperate
with each others to provide distributed name servers.

The directory is currently implemented as one of the special object spaces with segment
id 3. A directory entry corresponds to an object instance in the object space. In principle,
the directory could be implemented without making use of the object space storage mech­
nanism. However, in that case, the Object Manager then has to implement an alternative
mechanism to maintain storage for the directory. In terms of performance, it is probably
more efficient to implement a separate storage scheme to store the directory since each entry
in the directory is a fixed size object. Each disk page size could contain a fixed number of
objects without having to use the indirection of Object Table and Index Area. Without
the Object Table, writing out a new directory entry to disk requires less number of disk
operations. However, for the current implementation, we elect to use the existing storage
mechanism that comes with the object space abstractions for simplicity of implementation.

An entry in the directory contains the following information:

- an index into the Object Table corresponding to the object instance that provides physical storage for the entry.
- OSID of the object space,
- current site id of the object space,
- mapping control mode: indicates if the object space could be migrated, remote mapped or remote invoked. The control mode of the object space is set when the object space is created and could be changed via Control Object Space invocation.
- the current segment id of the object space. This field is relevant only if the object space currently resides in the local machine. Then the segment id points to the local segment on disk that contains the object space. Without object space migration, this field is not necessary since the OSID could contain the segment id of the object space which is unique at all times.
- the status of this object space. The status field is used to ensure consistency across machine failures when a committing transaction creates and deletes object spaces. The status field indicates two things: the state of the object space and the state of the transaction that creates or deletes this object space. A crucial piece of data that shows the existence or non-existence of an object space is the marking on the corresponding Segment Table entry as occupied or free. Section 6.5 elaborates more on the use of this status field to manage object space creations/deletions across machine failures.

When a transaction creating a new object space is committed, a new entry is created in the directory. Similarly, when a transaction deleting an object space is committed,
 CHAPTER 3. OBJECT STORAGE MANAGEMENT

Figure 3.9: Locating an instance in an object space of case 1 layout
the corresponding entry in the directory is deleted. Periodically, each Object Manager broadcasts its changes in the local directory to its peers in remote machines. Upon receiving the broadcast message, a remote Object Manager updates its own directory on disk. If for some reasons due to communications or machine failures, a remote Object Manager does not receive the broadcast message, it may not be aware of the creations or deletions of some object spaces.

When an Object Manager does not know the location of an object space, i.e., its local directory does not contain an entry of the object space, it broadcasts a look-up message on the OSID of the object space. Because our Communication Manager does not provide broadcast services, the Object Manager has to improvise its own ad-hoc message broadcasting scheme by using a group of worker processes. For each peer machine to which this Object Manager is connected, a worker process is used to do a RemoteSend() to the Object Manager of the peer machine. The peer Object Manager that currently owns the object space sends a positive response to the corresponding worker process. Object Managers in other machines that do not own the object space send negative responses back. Upon receiving this information, the Object Manager could update its own cache directory. At best, the directory represents a hint on the current location of an object space.
Chapter 4

Page Memory Management

In Raven, the Object Manager cooperates with the kernel for virtual memory management. The kernel deals with the lower level of memory management such as creating address spaces, mapping pages to address spaces, allocating stack and heap memory, and reclaiming memory from user's address space. Raven kernel memory management hides the machine-dependent portion of the virtual memory management. It contains machine-dependent code to deal with memory management hardware in the machine. In a machine with no memory management hardware, Raven kernel simply views the primary memory as a single linear memory space. The Object Manager, on the other hand, is independent of any memory management hardware. It records page frames owned by each transaction and the object spaces within these page frames. As briefly discussed in the introduction chapter, one of the main reason for involving Object Manager in virtual memory management was to avoid double paging. The other rationale for involving Object Manager is that the Object Manager is the only entity that maps object spaces to a client's address space. It has complete information about the pages owned by an object space and which of these pages are mapped into a client's address space. During a transaction commit, the Object Manager needs not communicate with any other entities for information regarding the pages modified by the transaction.
Central to our design for virtual memory management is the ability to handle page fault and page-out requests outside the kernel. In this respect, our virtual memory design resembles Mach virtual memory management [Rash87] which also moves page fault handling outside the kernel. In Mach, for each object mapped into virtual space, there is a pager which is a process thread to maintain the status information of all the virtual memory pages of the object. In Raven, instead of associating a pager for every object, the status information of all objects' memory pages are maintained by the Object Manager through its Paging Module. The Paging Module is responsible for handling page fault requests and page-out requests from the kernel. The kernel page server sends these requests to a process thread implementing the Paging Module. When a modified page is paged out of memory, it is written to a common Shadow Segment. Similarly, a page written out during a transaction commit is also placed in the Shadow Segment. The Object Manager allocates segment with id 1 as the Shadow Segment.

In a machine with memory management hardware, when a virtual memory page with no corresponding physical page in RAM is referenced, a page fault interrupt is generated by the hardware. The kernel page server intercepts this interrupt and sends a page fault request to the Paging Module to map in the faulted page from secondary storage. Before sending a page fault request, the kernel may need to page out some unpinned pages to reclaim physical memory pages to store the faulted page. Note that pinned pages cannot be paged out from memory. Consequently, the kernel may send a pair of page fault and page out requests to the Paging Module.

In a machine with no memory management hardware, virtual memory cannot be supported. All memory addresses are actual physical addresses. A page allocated for an address space remains in memory until it is explicitly deleted from the address space. When a page is deleted from an address space, its physical page frame is returned to the free page pool if no other address space is sharing the page. Since a page of an address space remains in
memory at all times until deleted, there is no page fault request generated by the kernel. A page-out request may still get generated by the kernel when the kernel runs out of free physical pages to allocate to an address space. In fact, in a machine with no paging hardware, the Object Manager must be involved in paging out object space pages from memory to guarantee the validity of instances’ memory addresses. The remainder of this chapter describes the implementation details of the memory management within the Object Manager. Our description pertains to memory management without hardware support since our current implementation does not make use of paging hardware; all memory addresses are physical addresses. Nevertheless, our implementation can be easily integrated with paging hardware.

4.1 Managing Resident Objects’ Pages

As mentioned in the previous chapter, physical memory in Raven is primarily used to cache object spaces in secondary storage. The Object Manager maintains information about all object spaces in resident memory. Some of the object spaces may be currently mapped into some client’s address space within a transaction. Other object spaces may not be currently mapped into any client’s address space. They remain in resident memory as caches in the Object Manager’s own address space, referred to as obmspace. In fact, when an object space is mapped in from disk, it is mapped into the obmspace first. The Object Manager then maps the object space to client’s address space via kernel page memory primitives. When the client’s transaction commits, the object space is removed from the client’s address space; however, it remains as caches in the obmspace. These caches can be removed from the obmspace if the Object Manager runs out of memory pages.

For a large object space, i.e., object spaces of case 2 layout, not all the pages need to be mapped into memory. Initially, when an object space of case 2 layout is mapped into
memory, only its segment descriptor page, its Object Table’s pages and its Page Hint Table’s pages are read into memory. After an object space is mapped into the client’s address space, GetInstance() invoked by the client is executed under the client’s process thread. When an instance referenced by a client via GetInstance() primitive has not been mapped into memory, an object fault occurs. Recall that the map bit field of an object descriptor in the Object Table indicates whether the corresponding instance has been mapped into memory or not. When an object fault occurs, the client’s process sends a message to the Object Manager’s server process to map the instance page into client’s address space. As a side effect of an object fault, all the instances that happen to be in the same page as the instance being referenced are also mapped into memory. Therefore, their corresponding bit maps are also updated. Recall that an instance page contains an index area which has reverse pointers to the object descriptors in the Object Table. These pointers are used to find the corresponding object descriptors and to set their map bits.

For object spaces of case 1 layout, all instances are mapped into memory when the segment descriptor page which contain the instances is read into memory. So there is no object fault for object spaces of case 1 layout. Figure 4.1 shows how pages are shared between the Object Manager’s address space and the clients’ address space.

4.2 Data Structures for Memory Management

This section describes the data structures used to maintain the information about the memory pages owned by resident object spaces. The Object Manager maintains an Address Space Table to record all the address spaces of currently running transactions. Each client’s address space has an entry in the table. An entry in the table contains the following information:

- space id of the address space,
CHAPTER 4. PAGE MEMORY MANAGEMENT

Use1's address space           User2's address space

- instance pages shared

Object Table shared mapped into user's address space

Object Manager's Address Space

- Object Tables maintained as caches
- Pinned pages
- Pages that store Object Manager's codes

These pages can be pinned and unpinned

Pages of object spaces in RAM

Figure 4.1: Mapping Object Spaces' Pages
• a pointer to the Logical Page Table,

• the current size of the Logical Page Table,

• and a pointer to a linked list of transaction records. The transaction record maintains
  the information about the object spaces mapped in by the transaction. The Object
  Manager places no restriction on the number of active transactions in an address
  space or the number of object spaces a transaction can map in. Although in most
  cases, there will probably be just one transaction, corresponds to a particular client,
  executing in an address space.

Each client's address space has a Logical Page Table associated with it. The Object
Manager derives information about the current location of a logical page from this table;
a logical page could be in secondary storage or in memory. An entry in the Logical Page
Table referred to as a page record, contains the following information:

• logical memory page number relative to the address space.

• logical disk page number of the original page on disk. The page number is relative to
  the segment that contains the object space. This field is NULL if the page is a newly
  created page of the object space; it does not have an original copy of the page on disk
  yet until the transaction that creates this page is committed.

• logical disk page number of the page relative to the Shadow Segment. This field is
  NULL if there is no copy of the page yet in the Shadow Segment; the page has never
  been paged out before.

An object space mapped to an address space has a Page Frame Map associated with it.
The Page Frame Map records which logical pages in the address space belong to the object
space since there may be a number of object spaces mapped by different transactions, all
residing in the same address space. An entry in the Page Frame Map is an address pointer
to the corresponding page record in the Logical Page Table. This information is created dynamically as clients submit mapping requests to the Object Manager. Figure 4.2 shows the relations among these data structures.

The following example shows how the records in the tables are created:

1. A client invokes MapObjSpace() within a transaction.

2. The Object Manager creates an entry in the Address Space Table if it does not already exist. When a new entry is created, a Logical Page Table is allocated for the entry.

3. The Object Manager then creates a transaction record of the client if it does not already exist.

4. It then allocates a Page Frame Table for the object space to records which pages of the object space are mapped into the client’s address space.

4.3 Memory Management without Paging Hardware

Our current implementation does not make use of memory management hardware. This has some consequences on our design approach on memory management. First, memory addresses are physical addresses. Pages of object spaces that are mapped into some client’s space within a transaction cannot be removed from memory until the transaction is committed or aborted. These pages are automatically pinned by the kernel when they are created. They can be unpinned via a call to UnPinPages() to the kernel or when they are deleted from the address space. After the transaction commits, these pages are deleted from the client’s address space, but they are still retained in the obmspace as caches. The cached pages are unpinned by the Object Manager via a call to UnPinPages() to the kernel and these are the only unpinned pages in the whole Raven memory space. These pages
CHAPTER 4. PAGE MEMORY MANAGEMENT

An entry in the Address Space Table

| Addr Space ID | Page table size |

Linked list of Transactions Records

| Tid |

Linked list of OSIDs

| OSID |

An entry in the Logical Page Table

| Logical Page Number relative to the Address space | Original Disk Page Number | Disk Page Number in Shadow Segment | Page address in memory |

Figure 4.2: Memory Management Tables
are subject to be removed by the kernel if the kernel runs out of free physical pages to allocate. When the kernel has to remove these *unpinned* pages, it must inform the Object Manager about the validity of these physical pages. In fact, the Object Manager is involved in paging out the cached pages from memory. To remove unpinned pages from memory and reclaim the physical pages occupied by the unpinned pages, the kernel sends a page-out request to the Paging Module process within the Object Manager. The kernel supplies the *address space id*, *starting logical page number* relative to the address space, *physical address* of the starting page, and the number of physically contiguous pages. The Paging Module uses the *address space id* to find the corresponding entry in the *Address Space Table*, and the *logical page number* to find the corresponding *page record* in the *Logical Page Table*. It then updates the *page record* by invalidating the physical page address. The Paging Module need not write out the physical pages since the physical pages must have been written out when the transaction is committed. After servicing the page-out request, the Paging Module sends a reply to the kernel, thus unblocking the *page server*. This service must be performed as efficiently as possible since page-out requests may have to be performed quite frequently. Two look-up operations are required to service the request: one to find the corresponding entry in the *Address Space Table* and one more to find the corresponding entry in the *Logical Page Table*.

A page of a cached object space in the *obmspace* may be removed from memory if the kernel runs out of physical pages. Consequently, physical addresses of instances in a cached object space may become invalid if the corresponding physical page is reused for other purposes. The Object Manager has to ensure the validity of these addresses when a cached object space is remapped into a client's address space. To ensure this, the Object Manager invalidates the *map bit* field of each object descriptor whose instance is in the page that is unpinned. By invalidating the map bits when an instance page is unpinned, we reduce the overhead involved in servicing a page-out request. Later if the unpinned page gets removed.
from memory, the Object Manager does not have to go through the Object Table again and invalidates the map bit fields. However, when a cached object space is mapped back into a new client's address space, the Object Manager has to check whether the instance pages of this object space (which were unpinned) were removed from memory while they were in unpinned state. If they were not removed from memory, the Object Manager requests the kernel to pin the pages again and the corresponding map bits in the Object Table is set. If they were removed from memory, then nothing needs to be done since the map bits in the Object Table was already invalidated. The Object Manager determines whether an instance page was removed from memory by checking the corresponding entry in the Logical Page Table which, among other thing, contains the physical RAM address of the page. If this field is NULL, then the instance page was removed from memory. See Figure 4.2. Note that the pages that contain the Object Table, the Page Hint Table, and the segment descriptor of an object space are still pinned. These pages are freed when the Object Manager decides to remove the whole object space from memory. Only the pages that contain object instances are unpinned. There is a limit as to the number of object spaces that can be cached in memory.

4.4 Memory Management with Paging Hardware

If hardware support for memory management is available, less work is required inside the Object Manager. An instance memory address is a virtual address, so there is no need to invalidate the address when its corresponding physical page is paged out of memory. Furthermore, there is no need to pin an instance page for data consistency since the instance page is a virtual page. If a referenced virtual page is not in memory, a page fault occurs to bring the virtual page to memory. When a page fault occurs, the kernel looks for some unused or least recently used page to be removed from memory and its corresponding
physical page is used to store the new page. Consequently, the kernel may send a page fault and a page-out requests to the Paging Module. The information supplied by the kernel is similar to the one without memory management hardware. However, the Paging Module handles the requests differently.

When a page fault request is received, the Paging Module finds the corresponding page record in the Logical Page Table based on the information supplied. If a shadow copy exists in the Shadow Segment, the Paging Module reads the page from the Shadow Segment; otherwise, it reads the page from its original segment on disk.

In most cases, a page fault will be accompanied by a page-out request. When page-out request is received, the Paging Module creates pages in the Shadow Segment to temporarily store the paged-out pages. It also updates the corresponding page records in the Logical Page Table to record the location of the pages in the Shadow Segment.
Central to any design of fault-tolerant systems involving multiple active processes is the concept of transaction. A transaction is a series of operations marked by a pair of delimiters, BeginTransaction and EndTransaction. A transaction typically has three special properties: serializability, failure atomicity, and permanence. Serializability ensures that intermediate states and possibly inconsistent states of data accessed by a transaction are not visible to other active transactions. This property requires a transaction facility to provide some form of concurrency control. Failure atomicity ensures that either all or none of the transaction's operations are performed. This property implies the existence of some recovery scheme that could revert an object to its consistent state when failures occur. Permanence ensures that if a transaction completes successfully, the modifications on the object's states will never be lost unless some catastrophic event occurs. Permanence requires that objects are secured on secondary storage that persists across machine failures. Unfortunately, these desirable properties guaranteed by a transaction come at a penalty of performance overhead. They require rather heavy weight mechanisms such as locks to ensure serializability, two-phase commit protocols, logging or shadowing on disk to ensure recoverability and persistence. To mitigate this penalty, distributed transaction facilities often allow some relaxation on the constraints of these properties for certain applications. The relaxation is based on the
fact that some applications do not really need such strong constraints to ensure consistency of their data or they can afford to have certain degree of inconsistencies on their data. However, this relaxation is often done on an ad-hoc basis to improve performance by allowing users to bypass some of the transaction facilities and replace it with their own application tailored mechanism. Raven transaction facility attempts to relax these constraints in a more structured manner by associating three basic attributes to a Raven object space. An object space can have any combinations of these three attributes:

- **persistent/volatile:**
  A *persistent* object space always has a permanent copy on secondary storage (long lived objects). It can survive machine failures and certain degree of media (disk) failure.
  A *volatile* object space is short lived. It does not exist across reboots. Examples of volatile object spaces are stacks, Unix c-shell. A *volatile* object space can be promoted to a *persistent* object space at any time during a session.

- **recoverable/nonrecoverable:**
  A *recoverable* object space has failure atomicity property attached to it. A *recoverable* object space is always consistent with respect to the series of operations applied to it and its intermediate states are not visible to other concurrently executing clients.
  A recoverable object space implies the existence of a back-up copy either in memory or in secondary storage. When a failure occurs, the Recovery Manager performs recovery to reset the object space to its previously consistent state by setting the back-up copy as the current copy of the object space. For example, modifications on a persistent and recoverable object space are written out to a temporary copy on disk, not on the original copy on disk. The original copy functions as a back-up copy. A *nonrecoverable* object space may be left in an inconsistent state when a failure occurs.
since any modification on the object space is applied directly to the original copy. A recoverable object space requires shadowing scheme while nonrecoverable object space does not.

- **sequential / concurrent**

A *sequential* object space does not require concurrency control from the object manager. The higher layers that define the object space as *sequential* provides their own concurrency control, perhaps tailored to the application associated with the object space. A *concurrent* object space requires concurrency control from the Object Manager. Several processes may concurrently access the object space. Object Manager must synchronize these concurrent accesses.

The following are some examples of the possible combination of attributes for an object space:

- **persistent - recoverable - concurrent:**
  Most object spaces will probably have these attributes. It requires the complete transaction mechanisms provided by the Object Manager since it demands all three properties of transaction.

- **persistent - recoverable - sequential:**
  Concurrency control is left to the users, but the Object Manager still ensures the recoverability of the object space across machine failures by using recovery mechanisms such as two-phase commits, logging or shadowing and commit records.

- **persistent - nonrecoverable - sequential:**
  The object space is long lived (persistent), but the object space may contain inconsistent or unfinished modifications if some machine failure occurs since modifications are done directly on the original copies on disk.
• volatile-recoverable-concurrent:

The object space does not survive across machine restarts since it does not have a permanent copy in secondary storage although it may have a temporary copy in secondary storage due to paging. It is recoverable in the sense that if the client that modifies the object space fails in the middle of the transaction, the state of the object space can be reverted back to its state before the start of the transaction. This implies the existence of a back-up copy in memory. Access to the object space is synchronized by the Object Manager.

This chapter and the next chapter represent the major portion of this thesis. This chapter is devoted to the concurrency control and nested transaction problems in the design of Object Manager. It describes how these problems are solved and the implementation details of transaction management. Next chapter deals with the recovery problems and the recovery procedures implemented in Raven.

5.1 Modules within the Object Manager

Conceptually, there are five modules or submanagers within the Object Manager. Each is responsible for a particular task. It is important to emphasize that this is only a logical division of tasks for software organization purposes: each module does not necessarily represent a distinct execution thread. In fact, as we will see below, the process structures that implement these modules are somewhat different from its logical division. The following modules or submanagers divide the tasks of the Object Manager:

1. Storage Module: is responsible for the physical storage of objects. It organizes the disk storage, provides storage abstractions from the linear disk storage space, and locates Raven objects through unique object identifiers. These were discussed in Chapter three.
CHAPTER 5. TRANSACTION MANAGEMENT

2. **Paging Module**: is responsible for the management of objects in virtual memory. It maintains the status information of all virtual memory pages occupied by Raven objects, and handles page faults and page-out requests from the kernel. These were described in Chapter four.

3. **Transaction Module**: is responsible for the management of transaction. In particular, it is responsible for transaction commit protocols, and the transaction trees of nested transactions.

4. **Lock Module**: is responsible for synchronizing access to objects from concurrently executing clients. In particular, it implements a locking scheme and a deadlock resolution protocol.

5. **Recovery Module**: is responsible for recovery after failures. It organizes commit records of all transactions and executes recovery protocol based on the information stored in the commit records.

5.2 **Synchronization and Process Structuring**

To realize the services of the above modules, the Object Manager is configured into a set of Raven processes. One of the main concern in structuring processes is how to synchronize access to critical sections or shared data within the Object Manager. Our objective is to allow a fast and efficient access to the shared data without jeopardizing the consistency of the shared data. Consistency of the data may be compromised if context switching occurs to a process executing in the critical section and another process who is going to execute in the same critical section is scheduled. There are several ways to solve this synchronization problem.
CHAPTER 5. TRANSACTION MANAGEMENT

One way is to implement semaphores [Dijk65] or a monitor [Hoar74] for each critical section. This approach is more suitable for procedure oriented systems that depend on some form of shared memory accessed through procedure calls.

The second approach is to encapsulate the critical section into a single process thread who is in charge of the critical section. Other processes that need to access to the critical section must send a message to the process in charge of the critical section. Since the critical section is only executed under one process thread, if the process is switched out while executing in the critical section, the inconsistent state is not visible to other processes since other processes cannot execute in the critical section. This approach resembles the client-server model of computation where the server is responsible for the critical section. It is more suitable for message-oriented system where sending and receiving messages are common operations. The setback of the second approach is that the overhead of message exchanges may become prohibitively high if the shared data are accessed frequently.

The third approach is to identify a group of processes that are going to access the shared data. These processes belong to a group process or a team as in V [Cher84]. The most important characteristic of the processes in a group is that if a process in the group is switched out due to time-quantum expiration, no other process in the same group will be scheduled. The next time the group acquires the CPU time, the original process is rescheduled to execute from where it left off. This can be done quite simply by inserting the original process in the ready queue ahead of all of its peer processes in the same group. Other processes in the same group are scheduled to execute only if the original process in the same group voluntarily give up control of the processor. Control is usually relinquished whenever a process performs a kernel call such as invoking a Send() operation. This approach does not require any message passing as in the client-server model, or setting any shared integer values as in semaphore approach, to access shared data. However, it requires the kernel process management to identify processes that belong to a group and to perform process scheduling.
within a group according to the synchronization requirement above. Furthermore, it requires
the programmer to know beforehand that each process in the group will eventually relinquish
control of the processor. This is crucial in order for other processes in the same group to
be scheduled for execution.

Our synchronization approach to shared data in the Object Manager is to use a com-
bination of the second and third approach. Some critical sections such as acquiring and
releasing locks on object spaces are implemented using the client-server model. Conceptu-
ally, it is easier to understand since there is a single process in charge of handing out locks
and reclaiming locks from clients. Furthermore, locks given out to clients require some
kind of monitoring on the health of the clients. The server is responsible for monitoring
the health of its clients. So the client-server approach is more appropriate. On the other
hand, activities such as access to the Global Segment Table, access to the Address Space
Table, and access to the disk free page pool should be synchronized following the process
group approach since these are very frequent activities in the Object Manager. Further-
more, they do not require some kind of monitoring because the processes that access the
critical sections belong to the group within the Object Manager, thus their health are not
jeopardized.

Process configuration in the Object Manager is based on the synchronization require-
ments discussed above. All processes implementing the Object Manager reside in the same
address space, obmspace; therefore, they have access to all shared data within the Object
Manager. The Object Manager is implemented by the following processes (see Figure 5.1):

- an Object Server process: is the most important process within the Object Man-
ger. It is in charge of receiving requests from client processes and replying to the
client processes; as such, it cannot block. Depending on the type of requests, the
Object Server may elect to fulfill the request itself or forward the request to one of
its worker processes. Requests and tasks such as reading from disk, writing onto disk,
Figure 5.1: Process Configuration in Raven
sending and replying to remote processes that cause the Object Server to block must be handled by worker processes.

- **a set of worker processes**: are created by the Object Server process to handle requests that may cause the executing process to block. To avoid the overhead of creating and destroying processes, a fixed number of worker processes are created during initialization of the Object Manager. If all the worker processes are busy, a request from a client is queued in the Object Server; otherwise, the request is forwarded to one of the idle worker processes. After a worker has processed a request, it does a Send() to the Object Server returning the results of the operation and also indicating that it is ready to perform the next request. The Object Server does a Reply() to the worker process and puts the next request in the reply buffer. The set of worker processes together with a page faults server process belong to the same group. Synchronization for shared data among these processes follows the group process approach as described above. A worker process is an I/O bound process since it mostly performs the tasks such as reading and writing to disks, and sending messages to remote processes. Consequently, in most cases, it will release its processor time before its allotted time expired. Our group synchronization approach therefore ensures that each worker process get a 'fair' share of the CPU time allocated for the Object Manager. The current implementation of the kernel does not use a time-sliced process scheduling. A process relinquishes control when it performs local local IPC, Sleep() or Suspend() operations. Nonetheless, our process structuring is designed with a time-sliced process scheduling in mind.

- **a page fault server process**: services page faults and page-out request from the kernel paging process. The kernel paging process is blocked while the request is being processed by the page fault server. This ensures that at any time, there can only
be one page fault or page-out request.

- a Recovery Manager process: is responsible for managing transaction commit records and executes recovery procedures after machine failures. To execute the recovery procedures, it creates a number of worker processes. The design and implementation details of the Recovery Manager are described in the next chapter.

5.2.1 Data Structures Accessed by Object Manager's Processes

The Object Server process is solely responsible for the management of the lock table and the queues for locks. To acquire or release a lock, a process sends a request to the Object Server process. The Object Server process is also responsible for caching object spaces in memory. To manage memory resident object spaces, the Object Server uses a hash-table of Osid Control Blocks. The Osid Control Blocks are described in more details in Section 5.5.1. The Object Server also functions as a transaction manager; therefore, it also maintains a hash-table of Transaction Control Blocks. A Transaction Control Block contains all the relevant information about a particular transaction. Section 5.5.2 elaborates on the management of transactions. The Object Server process has exclusive access to these three tables above, the lock table, the Osid Control Block table, and the Transaction Control Block table. These tables are volatile.

The sole purpose of the existence of the worker processes is to perform I/O tasks that could not be performed by the Object Server since the tasks cause the Object Server to block. Blocking tasks such as sending messages to a remote process, disk read/write operations, allocating and deleting pages on disk are performed by the worker processes. Disk page allocation or deletion requires modifications on the global Segment Table and the Free Page bitmap. The worker processes have shared access to these global data. When object space pages are read into memory by a worker process, a worker process must also update the
CHAPTER 5. TRANSACTION MANAGEMENT

Page Frame Map and Logical Page Table associated with the transaction. The worker processes share the Page Frame Maps and Logical Page Tables with the page-fault server process. Alternatively, we could assign the page-fault server process to be the sole manager of the Page Frame Maps and the Logical Page Tables. When a worker process maps in an object space, it will then have to inform the page-fault server process to update these tables. To reduce the communications between the worker processes and the page-fault server, we allow the worker processes to manipulate these memory management tables directly. This is realized by including the page-fault server process as a member of the worker process group. Processes that belong to the same group synchronize accesses to their shared data based on the non-preemptive scheduling described in Section 5.2 above.

The Recovery Manager process is responsible for the management of commit records. It has exclusive access to the commit records table. To create/remove a commit record and to update a commit record, a process must send a request to the Recovery Manager process.

5.3 Concurrency Control

Data management systems must synchronize access to the data from concurrently executing clients. Synchronization of access to data objects is often referred to as concurrency control. Literature on concurrency control problems abound [Bern81, Reed83, Kung81]. Typically there are three approaches for concurrency control: the locking approach [Eswa76, Gray78], the time-stamp based or multi-version approach [Reed83], and the validation approach [Kung81]. There is no strong argument or evidence in the literature showing that one approach is better than the others under all circumstances. The performance of a concurrency control scheme to some extent depends on the types of applications and also on the organization of data. Nevertheless, there are some general guidelines to evaluate a good concurrency control mechanism in distributed systems. In general, a good concurrency
control should [Kohl81]

- be resilient to node and communication network failure,
- permit parallelism to the extent necessary to satisfy system performance requirements,
- incur modest computational and storage overhead,
- place few constraints on the structure of the atomic actions.

5.3.1 Choosing a Concurrency Control Scheme for Raven

Several factors influenced our decision in choosing a concurrency control scheme. First, in the present implementation, the Object Manager has no knowledge of how object spaces are going to be used by the higher layers. An object space represents a physical storage abstraction for the higher layers. The storage for a Raven object such as a mailbox or an employee record as viewed by a Raven programmer may be a single object space or a single instance in an object space. Because of this lack of knowledge of how object space storage are typically used by the higher layers, we adopt a more conservative approach that will allow future modifications on the concurrency control or future additions of different concurrency control scheme. Secondly, the limited time and human resources imposed on the first implementation of the Raven system motivate us to choose a scheme that is well understood, easier to implement, and has been successfully used in many other distributed systems. These factors lead us to choose a locking scheme over multi-versions scheme or some optimistic validation scheme. Locking is a well understood synchronization mechanism that has been successfully used in other data management systems such as TABS [Spec85], Argus [Lisk82], QuickSilver [Hask88], and R [Gray81]. It is known to perform reasonably well for most applications. It is comparatively easier to implement locking than to implement multi-version or validation schemes. There are also some disadvantages to locking schemes.
First, locking schemes synchronize access to shared data by delaying other interested clients. Delaying clients in some situations may lead to deadlock. Therefore, all locking schemes have to be complemented by some deadlock resolution scheme. Secondly, locking requires some kind of monitoring on the health of current lock holders. This monitoring is complicated in a decentralized system with the possibility of communication and machine failures.

5.3.2 Granularity of Locks

One of the crucial factors in the performance of locking scheme is the granularity of the data locked. A large granularity may unnecessarily reduce concurrency. However, lock management also becomes simpler compared to those of finer granularity by the simple fact that fewer locks are given out to clients. Finer granularity does not necessarily increase concurrency since the overhead of managing large number of locks and the overhead of communicating with the lock server also grow substantially. In the Raven Object Manager, the unit of storage and memory mapping is the object space. Consequently, it is natural to apply locks on an object space basis. Locking on an object space has some advantages. Once an object space is mapped into a client's address space, i.e. the lock on the object space has been granted, the client need not communicate with the Object Manager until the next object fault occurs. It can access any instances in the object space without consulting the Object Manager. If locking is done on an instance basis, the client has to communicate with the Object Manager to acquire a lock for every instance in the object space that it wants to access. Communication is reduced even further if the object space is of case 1 layout. In this case, the client only communicates with the Object Manager once when it invokes MapObjSpace(); afterwards, the client can access all the instances in the object space since all the instances are automatically mapped into the client's address space. Request for a lock is implicit when the client invokes MapObjSpace() with Read or Write access mode. Recall
that an object space can have the attribute: sequential or concurrent. A sequential object space has no lock associated with it. Any client requesting access for a sequential object space is granted access without delay. The Object Manager assumes that clients implement their own concurrency control on the sequential object spaces. The main setback of locking on an object space basis is that concurrency may be reduced if the majority of object spaces are relatively large. Two clients cannot modify an object space concurrently although each of them may modify distinct instances in the object space.

If the majority of the object spaces are relatively small, then locking on an object space basis may not affect the overall system performance since it saves in communication and lock management overhead at the expense of some reduced concurrency. We envisioned that there is a one to one correspondent between a Raven object as viewed by the programmer and an object space as viewed by the Raven Object Manager. The instances in an object space can be used to store the components of a Raven object. In this case, the majority of object spaces will be relatively small, possibly of case 1 layout and locking on the whole object space is better than locking on each instance of the object space.

5.3.3 Lock Management

Raven Object Manager implements the conventional two-phase locking scheme to ensure serializability. There are two types of locks: Read lock and Write lock corresponding to the Read access mode and Write access mode of MapObjSpace operation. A Read mode allows a client to read an object space without preventing other clients from reading the same object space. A Write mode provides a client with exclusive access to an object space.

There are two ways to acquire a lock on an object space: implicitly by invoking MapObjSpace or explicitly by sending LockObjSpace request. The latter operation is mostly used by the Recovery Manager's processes to lock object spaces while doing recovery on the object
spaces. An object space modified by a transaction may be in an undetermined state when a machine failure occurs. The Recovery Manager has to lock the object space until the state of the object space is determined. Unlike the MapObjSpace operation, the LockObjSpace operation does not map the object space into the requestor’s address space; it simply prevents the object space from being accessed by other clients. In most cases, a client outside the Object Manager invokes MapObjSpace to map an object space to the client’s address space, thus implicitly acquires the appropriate lock on the object space. The exception is for the DeleteObjSpace operation. This operation does not require the Object Manager to map the object space to the client’s address space. The client only needs to have an exclusive access to the object space. DeleteObjSpace implicitly requests a write lock on the object space without mapping the object space into the client’s address space.

A client of the Object Manager can be a local or remote client. The Object Manager must monitor the health of lock holders to prevent locks from being held indefinitely. The Object Manager can forcibly release locks held by a client if the Object Manager detects some failure on the part of the client. When and how an Object Manager can forcibly release locks hold by a client is discussed in the implementation section.

5.3.4 Deadlock Resolution

The use of locking for concurrency control introduces the possibility of one transaction being suspended waiting for a lock held by another transaction. When one transaction is suspended waiting for a second transaction, and the second transaction is waiting either directly or indirectly for the first transaction, the result is a circular wait condition and hence the possibility of deadlock. Algorithms to resolve deadlock abound in the literature [Ober82, Knap87, Rose78]. We use a combination of time-out and deadlock prevention called the Wait-Die scheme [Rose78]. Most experimental distributed systems such as TABS
[Spec85], Argus [Lisk82], Eden [Jess82] rely on time-outs to prevent deadlocks. A time-out scheme is based on the assumption that if a time-out occurs (providing a reasonably long time-out period) on a transaction waiting for a lock, chances are the transaction is involved in a deadlock situation. Normally, the system chooses to abort the waiting transaction over the current transaction holding the lock. Relying on time-outs to resolve deadlock is the simplest of all the mechanisms since it essentially does not require any extra works on the management of locks except setting a time-out period. Some systems such as TABS let the clients set their own time-out periods. Unfortunately, relying on time-outs alone to resolve deadlock may result in unnecessary transaction aborts. A time-out on a waiting transaction does not necessarily mean that a deadlock has actually occurred. The current holder of the lock may be a long running transaction or the holder may have crashed or the holder's machine may have lost connection with the lock manager; therefore, the lock may not get released before the time-out expires.

**Wait-Die** scheme is a conservative deadlock prevention scheme that allows a transaction to wait for the lock held by another transaction only if such a wait never causes a deadlock; otherwise, the transaction is aborted. The scheme assumes the existence of a *unique number* which is assigned to each transaction. The number is chosen from a monotonically increasing sequence which is often a function of the time of day. A *time-stamp* is normally used for such a unique number by concatenating the local clock time with the local machine identifier. In **Wait-Die** scheme, the requestor transaction is allowed to wait for the current holder if it is older (has a smaller or earlier timestamp) than the current holder; otherwise, the transaction is aborted and restarted again at a later time. The protocol guarantees that a cycle or a deadlock can never occur since a younger transaction is not allowed to wait for an older transaction. Unfortunately, the scheme may unnecessarily abort transactions since a younger transaction requesting a lock held by an older transaction must be aborted although no deadlock occurred.
The only other way to resolve deadlock without unnecessarily aborting a transaction is to fully implement a distributed deadlock detection scheme by constructing a global wait for graph [Ober82]. However, the scheme increases the complexity and the costs of managing locks substantially. Constructing a global wait for graph requires constant communications among the lock managers in the distributed system. The performance gained in reducing the number of transaction aborts must be weighed against the overhead of constructing wait for graphs.

Our combination of time-out and Wait Die scheme represents a compromise to improve concurrency without increasing implementation complexity. Initially, if a transaction requests a lock that is in conflict with the current lock on the object space, the transaction is allowed to wait and a time-out period is set by the Object Manager. If a time-out occurs, chances are that a deadlock may have occurred. However, instead of immediately aborting the transaction as in the pure time-out scheme, the Object Manager checks if the transaction can still wait with a new time-out period. When a time-out occurs, the Object Manager checks the health of the current lock holder(s). If the current holder happens to be in a remote machine, a probe is sent to the remote machine. The way the probe is sent is described in the subsequent implementation section. If the current holder is alive, then the Object Manager compares the timed-out transaction’s timestamp with the current holder(s)’ timestamps according to the Wait-Die scheme. If there is no potential deadlock, the transaction is allowed to wait again with a new time-out period; otherwise, the waiting transaction is aborted. By first allowing a transaction to wait until a time-out period expires, we attempt to reduce the number of unnecessary transaction aborts as in the case of pure Wait-Die scheme. Even after a time-out period expires, the transaction may still be allowed to wait if such a wait conforms to the Wait-Die scheme. In doing so, we attempt to reduce unnecessary transaction aborts when an older transaction happens to wait for a long running younger transaction.
5.3.5 Scheduling Waiting Requests

When a lock is released by the current holder, the Object Manager must decide who gets the lock next among the waiting transactions. Our deadlock resolution scheme affects the scheduling of the next holder(s). The scheduling cannot use a First Come First Served basis since it may violates the Wait-Die scheme by allowing a younger transaction to wait for an older transaction. Note that because of the existence of time-out period, violating the Wait-Die scheme will not cause a deadlock. Using a better scheduling mechanism, we can avoid unnecessary aborts. The approach is to grant the lock to the youngest transaction among the waiting transactions. If the youngest waiting transaction happens to request a Read lock, then the Object Manager may also grant Read locks to all other Read waiting transactions that are younger than the youngest Write waiting transaction.

By allowing an older transaction to wait for a younger transaction (Wait-Die scheme), we ensure that a long running transaction (older) which may have done substantial operations are not aborted by a younger transaction, thus reducing the cost of transaction aborts. The drawback of the Wait-Die scheme is that an older transaction may not be able to complete at all if younger transactions keep on coming into the front of the queue while the older transaction is waiting in the queue. To prevent this, the Object Manager sets a limit to the number of younger transactions that can overtake an older waiting transaction in the queue. For each waiting transaction, it keeps a count of the number of younger transactions that has jumped in front of the queue. When this count reaches a limit, no new younger transaction is allowed to queue in front of this waiting transaction. The following example (see Figure 5.2) illustrates the scheduling algorithm. A transaction with timestamp n is denoted by $T_n$. Transaction $T_3$ is currently holding a write lock on an object space $X$. Transactions $T_1$ is waiting for Read, $T_2$ is waiting for Write, $T_4$ and $T_5$ are waiting for Read. It is immaterial as to which of these waiting transactions submits its request first to the Object Manager. The Object Manager queues the requests based on the time-stamps of the
CHAPTER 5. TRANSACTION MANAGEMENT

READ Waiting List
T1  T4  T5
WRITE Waiting List
T2

Current Holder: T3
Write Lock

T3 releases the lock

READ Waiting List
T1
WRITE Waiting List
T2

Current Holders: T4 and T5
Read Lock

T4 and T5 release locks

READ Waiting List
T1
Current Holder: T2
Write Lock

Figure 5.2: A Lock Scheduling Example
transactions with the youngest transaction (larger time stamp) in front of the queue. After $T_3$ releases the lock, providing no time-out occurs yet, both $T_4$ and $T_5$ gets the Read lock. However, $T_1$ cannot be granted a Read lock at this moment since it may unnecessarily abort $T_2$ when $T_2$ times out. After $T_4$ and $T_5$ release their locks, then $T_2$ is scheduled followed by $T_1$.

5.4 Transaction Invocation and Completion

As mentioned in the early part of this chapter, Raven transaction facility allows some relaxation on the constraints imposed by an atomic transaction. Such relaxation is accomplished by associating atomicity attributes to an object space. Our discussion of transaction management is mostly devoted to the strictest notion of transaction mechanism that provides serializability, recoverability and permanence. We expect that the majority of object spaces do require the complete transaction facility provided by the Object Manager. Whenever appropriate, we will mention how the transaction mechanism handles object spaces that do not require the whole transaction facility.

In Raven, a transaction can be composed of other transactions or normally referred to in the literature as nested transactions. The Object Manager implements a subset of the nested transaction model as described in [Moss81]. In our implementation, the parent transaction waits for the child transaction to complete before continuing with its own execution. A child transaction is normally referred to as a subtransaction of its parent transaction. A child transaction can in turn creates its own substractions, thus forming a transaction invocation tree. Nested transactions have all the properties of traditional transactions. An additional property of nested transactions is that subtransactions of a given parent transaction fail independently of one another. The parent transaction can create an alternate subtransaction in place of a failed subtransaction in order to accomplish the same task.
In a distributed environment, nested transaction executions can be distributed across many machines. Managing the transaction invocation trees across the network while ensuring atomicity and recoverability require a well defined set of protocols. These protocols enforce atomicity and recoverability in the face of communications and machine failures. One of the most common of protocols is the *two-phase commit* protocol to ensure atomicity of distributed transactions. Raven also implements a variation of the two-phase commit protocol, the tree-structured two-phase commit protocol.

Briefly, a two-phase commit protocol executes as follows when the coordinator wants to commit the transaction:

1. The coordinator sends out *prepare to commit* messages to all the participants of the transaction. The coordinator itself writes a record in secondary storage to record its state as *prepared* along with other information such as its participants' transaction ids. This stable record is normally referred to as a *commit record*. The coordinator then waits for replies from its participants.

2. Upon receiving a *prepare to commit* message from its coordinator or parent transaction, a participant flushes out its modified data to a temporary location on disk if it has not already flushed out the data. Once the modified data is securely written on disk with the proper commit record associated with it, the participant sends a positive reply to the coordinator indicating that the participant is already in *prepared* state and ready to commit. Steps 1 and 2 form the *first phase* of the protocol.

3. Upon receiving replies from all participants, the coordinator can decide to commit or to abort the transaction. In either case, the decision made by the coordinator must be securely recorded in secondary storage via the commit record. If all participants send positive replies, the coordinator commits and sends out a *commit* message to all its participants to make the changes permanent; otherwise, the coordinator aborts
and sends out an *abort* message to all its participants.

4. Upon receiving a *commit* message from the coordinator, a participant makes the changes to data permanent through some underlying facilities and sends back an acknowledgement to the coordinator. The acknowledgement is used by the coordinator to perform garbage collection on its commit record.

5. If all participants acknowledge the *commit* message, the coordinator can free its commit record storage; otherwise, the commit record must be retained since some participant has not yet learnt the final state of the coordinator. Steps 3, 4 and 5 form the second phase of the protocol.

During recovery, another set of protocols is run to complete the transaction.

### 5.4.1 Distributed Transactions

The current implementation of Raven enables a transaction to access a remote object space in two ways: one is via *a Remote Mapping* and another one is via *a Remote Invocation*. In *a Remote Mapping*, the Object Manager communicates with its remote peer that currently owns the object space to map a copy of the object space to the local machine. By contrast, in *a Remote Invocation*, the Invocation Manager (the higher layer) communicates with its peer in the remote machine to perform an operation on the object space. To do a remote invocation on an object space, the Invocation Manager must somehow know the location of the object space, presumably, via some naming services. When a transaction accesses a remote object space through one of the two mechanisms above, the transaction becomes distributed and executes a two-phase commit protocol when it commits.

The following example illustrates how a subtransaction gets created as part of a distributed transaction:
1. Transaction $T_1$ at site A sends a `MapObjSpace(Tid1, X,..)` request to the local Object Manager.

2. The Object Manager performs a lookup on the object space Directory to find out the current location of OSID X. If there is no entry of OSID X in the Directory, the Object Manager broadcasts a lookup on OSID X to its peers in remote machines. The remote peer in which the object space is currently located sends a positive response containing the location of X and the control modes of X. Upon receiving the response, the Object Manager updates its own Directory to record the location of X.

3. If X is local, then the Object Manager can map the object space to $T_1$'s address space. If X is not local, the Object Manager has three choices depending on the control modes of object space X previously set via `ControlObjSpace` invocation. The Object Manager can:
   
   (a) replies back to $T_1$ and let $T_1$ sends a Remote Invocation to the site that currently owns X, for example, site B.

   (b) or requests the remote Object Manager in charge of X to map a copy of X to the local machine (Remote Mapping).

   (c) or requests the remote Object Manager to migrate the whole object space X to the local machine.

5.4.2 Remote Invocation

From the example above, if $T_1$ sends a Remote Invocation to an Invocation Manager at site B, the Invocation Manager at site B then starts a subtransaction, $T_2$. When $T_2$ issues a `CommitTransaction` request to the local Object Manager, the Object Manager at site 2 is aware that $T_2$ is a subtransaction of some transaction $T_1$ at site A; therefore, it creates
temporary copies of the object spaces modified by \( T_2 \) in the Shadow Segment on disk and marks \( T_2 \) as in prepared state. The temporary copies will become permanent when the parent transaction \( T_1 \) commits. The Invocation Manager sends back the result to \( T_1 \). If \( T_2 \) is aborted for some reason, a negative result is sent back to \( T_1 \) and \( T_1 \) can then decide how to proceed, perhaps sends a new remote invocation to site B. If a positive result is returned, \( T_1 \) records \( T_2 \) as its participant by recording \( T_2 \) identifier in \( T_1 \)’s participant list. Notice that \( T_2 \) is already in its prepared state when control is returned back to \( T_1 \). Consequently, if all participants of \( T_1 \) are generated due to remote invocations, by the time \( T_1 \) decide to commit, all its participants are already in prepared state. In this case, \( T_1 \) can skip the first phase of the two phase commit protocol since the first phase has been done when a remote invocation returns with a positive result. This is due to the fact that in our simple nested transaction scheme, a parent transaction is blocked waiting for its child transaction to finish.

5.4.3 Remote Mapping

If the control mode of an object space allows a remote mapping on the object space, the Object Manager could remotely map the object space transparent to the invoking client. The following example illustrates a typical case when a remote mapping could arise:

1. Transaction \( T_1 \) at site A requests to map object space \( X \) which happens to be located at site B.

2. Object Manager at site A then sends a remote mapping request to its peer at site B.

3. Upon receiving a remote mapping request from the Object Manager at A on behalf of transaction \( T_1 \), the Object Manager at B sends a copy of the object space \( X \) to site A if no one is currently holding a conflicting lock on \( X \); otherwise, the request is
queued in the lock waiting list. If X is of case 1 layout, then the Object Manager at B sends the whole object space to A since it only occupies one page size. If X is of case 2 layout, i.e., the object space is large, only the segment descriptor page of X and the Object Table of X are sent to A. When an object fault occurs at A, the Object Manager at A sends a request to B to map the corresponding page that contains the object instance to A. We do not wish to send the whole copy of a large object space to A since perhaps only a small portion of the object space will be referenced by \( T_1 \).

4. Object Manager at B notes that \( T_1 \) is currently holding a lock on the object space X. It may have to monitor the health of \( T_1 \) if there are other outstanding requests for object space X.

5. When \( T_1 \) decides to commit the changes, Object Manager at A sends a prepare message along with the changes done on object space X to the manager at site B. Only the modified pages of object space X are sent back to site B.

6. The manager at site B installs the changes on X to a temporary copy in the Shadow Segment on disk, creates a commit record to record that a subtransaction is prepared, and sends back a ready to commit message to A. This corresponds to the first phase of the two phase commit protocol.

7. Upon receiving an acknowledgement from B, Object Manager at A can then proceed with the second phase of the protocol.

5.4.4 Object Space Migration

If the control mode of an object space indicates that migration is desired, the Object Manager requests the remote Object Manager to migrate the object space X. If object space X is currently mapped by other clients, migration is delayed until all the clients release X. Any new request to map object space X is delayed until X is migrated to the new location.
When an object space is migrated to a new location, a new segment is allocated for the object space and the directory is updated. The previous segment occupied by X in the old site is freed. Note that object space migration has not been fully implemented at the time of this writing.

5.5 Implementation Details

5.5.1 Managing Cached Object Spaces

A local object space mapped to main memory by a transaction remains in main memory as a cache after the transaction is committed. The maximum number of object spaces that can be retained in memory is a system parameter that is dependent on the size of the main memory. To maintain the status information on all of the object spaces mapped into main memory, the Object Manager implements a hash-table of Osid Control Blocks. An Osid Control Block contains the following information (see Figure 5.3):

- the OSID of the object space.

- the memory address of the object space header, obsheader. An obsheader is like a handler for the object space in memory. Knowing the handler, all instances in the object space can be located in memory or in secondary storage.

- the identifier of the site that is currently responsible for the permanent copy of the object space on disk.

- current map mode or current lock mode held by the client. It can be Read or Write or None. Current map mode None means that no one is currently holding a lock on the object space.
• the list of active transactions that are currently holding locks on the object space. There can only be one transaction in the list if current map mode is Write. The list is empty if current map mode is None.

• the list of transactions waiting for Read locks on the object space.

• the list of transactions waiting for Write locks on the object space.

• a boolean flag indicating if the object space is in the process of being mapped into main memory by one of the worker processes. The flag is used to avoid double mappings of the same object space into memory by two worker processes. For example, if two clients A and B happen to map a local object space X concurrently for Read. The request from client A is received by the Object Server first and is dispatched to a worker process. A moment later the request from client B is received by the Object Server process. By checking the flag, the Object Server does not have to dispatch the request from B to a worker process since another worker is currently mapping the same object space into memory.

• a boolean flag indicating if an instance page of an object space is currently being mapped in by a worker process. The flag is particularly useful to reduce the number of messages sent in the case of remote mapping by two or more clients. For example, if two clients A and B happen to map a remote object space X of case 2 layout. If an object fault occurs on client A, a worker process sends a message to the remote Object Manager requesting the faulted instance page. If a moment later client B also generates an object fault on the same instance page, the Object Manager need not send another message to the remote Object Manager since one message has been sent out to map the same page.
• the list of clients waiting for the mapping to be processed by the worker processes. This list applies to a client waiting for a local or remote page mapping due to object faults. When an instance page is mapped into memory by a worker, the Object Server checks the list if any other clients are also waiting for the same page. If so, it unblocks the clients.

For a remote object space mapped into a local machine, the Object Manager does not retain the object space as a cache after it is committed. The reason being that the cache may become inconsistent with its original copy and the cost of maintaining a consistent cache can be quite high. Even if a cache is maintained, a new client that wishes to map the cache must first acquire the lock from its original Object Manager at the remote site. At that point, the remote Object Manager can reply with an up to date copy of the object.
space. Maintaining a cache on a remote object space may become worthwhile if the object space is large such that once a client acquires a lock and providing the cache is still valid, the client does not have to communicate with the remote server when an object fault occurs since the local cache exists. However, maintaining a cache on a large remote object space also consumes substantial memory space.

5.5.2 Managing Transactions

Raven Object Manager exports the following primitives for transaction operations:

- **BeginTrans((var) tid)** begins a new top level transaction and returns the transaction id, \(tid\) to the client. The transaction id is used for subsequent object space invocations by the client. A transaction id contains two fields: the machine id where it is created and time stamp when it is created. A timestamp is a monotonically increasing sequence numbers that uniquely identifies a transaction across machine failures.

- **BeginSubTrans( parent tid, (var) child tid)** begins a subtransaction and returns the child transaction id. The client must submit the parent tid so that the Object Manager can register the subtransaction as a child of the parent transaction. Invocation Managers calls this primitive when it receives a Remote Invocation request from the parent transaction in the remote machine.

- **CommitTrans(tid, list of child transaction ids)** commits the transaction. When a client commits or aborts a transaction, it also submits a list of child transactions ids that have been created in remote machines due to remote invocations. From this list, the Object Manager learns the number of child transactions in remote machines and starts the commit or abort protocols. Because remote invocations are
application specific operations, the client process communicates with its peer process in the remote machine to perform the remote invocation without involving the Object Manager. Therefore, the Object Manager is not aware of any child transactions created due to remote invocations until the transaction commits or aborts. An indication is returned to the client if the commit succeeds or fails.

- **AbortTrans**(tid, list of child transaction ids) aborts the changes done by the transaction in the local machine and aborts all its child transactions in remote machines.

When a client process sends a **BeginTrans** request to the **Object Server** process, the **Object Server** process assigns a new transaction id for the client and creates a Transaction Control Block called *tblock* to record the activity of the transaction. Each transaction has a *tblock* associated with it in the Object Manager space and a hash table of *tblocks* is maintained in the Object Manager space. A *tblock* contains the following information (see Figure 5.4):

- the current state of the transaction which can be *active*, *prepared*, *committed*, or *aborted*.

- the type of the transaction which can be one of the following:
  - **Coordinator**: if the transaction is a top level transaction,
  - **RMapParticipant**: if it is a subtransaction created due to a remote mapping,
  - **RInvParticipant**: if it is a subtransaction created due to a remote invocation and it does not have its own child transactions.
  - **Coordinator-Participant**: if the transaction is both a child of some parent transaction and a coordinator to its own child transactions. A transaction of type
### Current State
- Type of Transaction
- PID of process thread
- Address Space Id
- Parent Transaction Id
- Transaction Id
- # of object spaces mapped
- list of objspace handlers
- Number of Subtransactions
- Subtransaction ids list
- Commit Record Id

![Figure 5.4: A Transaction Control Block](image-url)
*R InvParticipant* becomes a *Coordinator Participant* when it creates child transactions through remote invocations and remote mappings.

- PID of the process that initiated the transaction. The field is used by the Object Server to monitor the health of the process that has acquired some locks under the transaction.
- the address space id in which the transaction process's thread is currently executing.
- transaction id of the parent if the transaction is not a top-level transaction.
- its own transaction id.
- the number of object spaces mapped under the transaction.
- the list of object space handlers corresponding to the number of object spaces mapped.

An object space handler contains the following information about the object space:

- OSID of the object space,
- *memory address* of the object space header.
- the current size of the object table. This is useful if the object table is expanded within the transaction.
- current map mode of the object space by this transaction.
- the current site id in which the object space has a permanent copy on disk. The site identifier distinguishes a local object space from a remote one. If it is remote, then when the transaction commits, the changes on the object space are sent back to the site that owns the object space on disk.
- the number of subtransactions created under the transaction.
• the list of the subtransactions' ids.

• the commit record id associated with the transaction. A commit record is only created when a transaction commits. In some cases, a transaction may not need a commit record to ensure atomicity as we will see in subsequent sections.

5.5.3 Committing Transaction

This section describes the actions taken inside the Object Manager when a client sends a CommitTrans request to the server process. The actions taken depend on whether the transaction is a top-level transaction or a subtransaction and on the types of mappings done under the transaction. We begin by discussing how a subtransaction commits. There are three types of subtransaction as mentioned above: RInvParticipant, RMapParticipant and Coordinator-Participant. When the Object Server receives the request to commit a transaction, the Object Server forwards the request to a worker process.

Committing RInvParticipant Transaction

If the transaction's type is RInvParticipant, the worker executes the following protocol:

1. If an object space has been modified by the transaction, the worker creates temporary pages in the Shadow Segment to store the modified pages. Note that only the modified pages of an object space are written out to the Shadow Segment; the rest of the pages are not written out. In a machine with paging hardware, a dirty page can be detected by the hardware. In a machine with no paging hardware, the worker must scan through the Object Table to check which instances have been modified by looking at their modified bits. If a modified bit is turned-on, then the corresponding instance has been modified. Recall that a modified bit is turned on when the client invokes GetInstance with write mode as one of the parameters. The pages that contain these modified instances are written out to the Shadow Segment.
2. The worker then creates a commit record and sets the state of the transaction to 
*prepared*. The commit record also contains other information such as the transaction 
id of the parent, the page numbers of the modified pages in the Shadow Segment and 
in the original segments. The worker communicates with the Recovery Manager to 
create the commit record.

3. Once the commit record is secured on disk, the participant is *prepared* waiting for the 
final commit message or abort message from its parent transaction.

4. The worker returns an indication to the client. The client in this case is most prob­
ably the Invocation Manager which then informs its peer of the result of the remote 
invocation.

### Committing a Coordinator Transaction

If the transaction type is a top-level *coordinator*, the worker executes the following 
protocol. To clarify the illustration, we use the following transaction invocation tree. A is 
the top-level coordinator at site 1; it has an *RInvParticipant* at site 2 and an *RInvPartic­i­
ipant* at site 3; and also maps a remote object space from site 3 and a remote object space 
from site 4. Figure 5.5 shows the communications among the processes that execute the 
distributed transaction.

1. The worker process examines the list of object spaces’ *handlers* held by this trans­
action. If any of the object spaces are remote object spaces, then the worker must 
go through the first phase of the two-phase commit protocol. At this point, all the 
subtransactions recorded by the *tblock* are subtransactions due to *remote invocations*. 
These subtransactions are already in *prepared* state because of the *RInvParticipant* 
commit protocol above. For a *remote mapping*, a subtransaction is not created at the 
remote site until the coordinator sends back the changes on the object space in the
Figure 5.5: Process interactions during transaction commits
first phase of the protocol.

2. For each remote machine that needs to be involved in the execution of the protocol, a slave process is created by the worker. The set of slave processes are used to reduce the time to execute the protocol. If the worker process itself has to send to each participant in the remote machine, it can only send to one at a time instead of sending to all participants concurrently. From our example, there will be three slave processes correspond to site 2, site 3, and site 4; we call these processes $p_2$, $p_3$ and $p_4$. Process $p_2$ does not participate in the first phase of the protocol since there is no message to send to site 2 at this stage. Processes $p_3$ and $p_4$ send the changes on the object spaces to the Object Servers of the corresponding sites. This corresponds to the first phase of the protocol.

3. Upon receiving a prepare to commit message from $p_3$, the Object Server at site 3 creates a subtransaction of type $RMapParticipant$ to identify the remote mapping operation. The subtransaction created is simply for transaction management purposes to identify the remote mapping operation as one of the participants of $A$; it does not have a separate process. The Object Server at site 3 then modifies the object space according to the changes made by $A$ and writes the modified pages to the $Shadow$ $Segment$. It also creates a commit record and sets the $RMapParticipant$ transaction state to prepared. The Object Server at site 3 replies to $p_3$ the subtransaction id just created. Similar procedure occurs at site 4.

4. While $p_3$ and $p_4$ send out prepare messages to site 3 and site 4, the worker process at site 1 creates a commit record to record the coordinator state and its participant list. The coordinator state is set to prepared. The participant list is not filled up yet since $p_3$ and $p_4$ have not received back the subtransactions' ids. At this point, the participant list in the commit record only contains the subtransaction ids due to
CHAPTER 5. TRANSACTION MANAGEMENT

remote invocations.

5. When $p_3$ and $p_4$ receive back positive acknowledgements, the worker process includes the returned subtransaction ids into the participant list of the commit record. At this point, the worker decides to commit or to abort. If the worker decides to commit, it sets the transaction state in the commit record to committed state and writes out the commit record to disk. At this point, transaction A is committed. If for some reason, $p_3$ or $p_4$ does not return a positive acknowledgment, the worker must abort the whole transaction. In this case, the worker sets the transaction state to aborted in the commit record and sends out an abort message to its participants through its slave processes.

6. If the worker decides to commit, it requests processes $p_2$, $p_3$ and $p_4$ to send a commit message to the corresponding participant machines. The worker process waits for positive responses from all the participants acknowledging the receipt of the commit message.

7. If all the participants send back acknowledgements, the worker requests the Recovery Manager to delete the commit record; otherwise, the commit record must remain until all the participants are informed about the final state of the transaction. A participant may not receive the message because the participant's machine has failed or has been partitioned.

Processes $p_2$, $p_3$ and $p_4$ are simply used to send messages to remote machines and to improve concurrency in the executions of the commit protocol. They terminate themselves when they have accomplished the task.
Committing a Coordinator-Participant Transaction

If the transaction type is Coordinator-Participant, the worker only executes the first phase of the commit protocol. The procedure is identical to the case of a coordinator transaction. The worker does not proceed with the second phase since the transaction must wait for its parent before continuing with the second phase of the protocol. After finishing the first phase, the transaction is in prepared state and waits for the parent transaction to issue a commit or an abort message. If the first phase fails, the worker aborts the transaction and returns a failure indication to the parent transaction. When the parent transaction issues a commit or abort message, the participant transaction executes the second phase of the coordinator protocol, i.e. informing its own child transactions of the final state of the transaction.

5.5.4 Managing Locks

Lock management is implemented within the Object Server process thread. To map an object space into the client's address space or to lock an object space, the client's process sends a request to the ObjectServer. A process sending a request to the Object Server is blocked until the Object Server replies to the process by granting the lock or returning an error code indicating that lock cannot be granted. In the latter case, the client may decide to try again, or abort itself.

To manage locks, Object Server maintains a table, called Lock Table (see Figure 5.6). The table records locks held by active transactions which can be local or remote. An entry in the table is allocated for an active holder of locks. An entry contains the following information about the active holder:

- transaction id of the holder.
• current state of the transaction. This information is used by Object Server to determine when it could forcibly release locks held by a transaction in case of failure. For example, if the current state of a remote transaction is active and the Object Server detects that the remote machine has crashed or the remote transaction no longer exists, the Object Server could forcibly release the lock held by the transaction.

• list of osid locks held.

Recall that the waiting lists for an osid lock are maintained in an Osid Control Block. For each transaction in the waiting list, there is a time-out associated with it. The Object Server maintains a single list of time-outs in which each node in the list is associated with a waiting transaction. The node records the time left for the waiting transaction before a
time-out occurs on the transaction. The nodes are ordered with the smallest time left in front of the list. The list is a delta list where the time-left of a node in the list is a sum of all the values of nodes preceding this node in the list. For each clock tick, only the value of the first node in the list needs to be decremented. As an example (see Figure 5.7), the time left for $T_1$ is 8, for $T_2$ is 10, for $T_3$ and $T_4$ are 12. When a clock tick occurs, the value of $T_1$'s mode is decremented to 7.

When a time-out occurs on a transaction waiting for an object space $X$, the Object Server checks the health of the current holder(s). If the current holder is local and its status is active, the Object Server requests the kernel to check if the process thread executing the transaction still exists. If it does not exist, then the process must have failed and the Object Server can forcibly release the locks held by the deceased transaction. If the current holder is remote due to remote mapping and its state is active, the Object Server sends a probe message through its worker to the remote Object Server requesting the status of the remote holder. If the remote machine has crashed or the remote transaction no longer exists, then the Object Server forcibly releases the lock held by the remote transaction. If the state of the remote holder is prepared, the Object Server cannot release the lock even though the remote machine may have crashed. The remote holder may have decided to commit right
before it crashed. When the remote machine recovers, its Recovery Manager will release the locks held. There may be more than one transaction time-out at the same clock tick since the time-out period is constant and set by the Object Server. To reduce the number of probe messages sent out, only one probe message is sent out per machine. A probe message contains the active transaction ids whose status are inquired by the Object Server. Note that the Object Server does not monitor the health of its current holders constantly. The Object Server only checks the health of the current lock holder if there is a waiting client time-out.

Occasionally, the Object Server may detect that a remote machine has crashed through its Communication Manager; for example, when it tries to do a RemoteSend() to the failed machine, the Communication Manager returns an error code indicating that the remote machine has failed or connection has been severed. Learning this condition, the Object Server forcibly releases all locks held by the active transactions executing in the remote machine. Only the locks held by remote transactions whose state are active could be released. The locks held by remote transactions whose states are prepared can not be removed since the remote machine may have been partitioned and the remote transactions may have decided to commit during the partition. Locks held by an active remote transaction can be forcibly released because an active remote transaction must send a prepare message to the Object Server to become prepared. When an active remote transaction sends a prepare message to the Object Server, the Object Server first checks if the remote transaction still holds the lock on the object space that is being prepared. The Object Server may have forcibly released the lock on the object space and given the lock to some other transaction during a partition since the Object Server assumed that the remote machine has crashed. In this case, the Object Server sends a negative response to the prepared message and the remote transaction will have to abort because it no longer holds the lock on the object space.
5.5.5 Managing Object Spaces of Various Attributes

Our discussions above mainly pertain to **concurrent**, **recoverable**, and **persistent** object spaces which require a complete transaction facility. We expect the majority of the object spaces in Raven fall into this case. This section describes how the Object Manager handles object spaces of other attributes. Recall that the attributes of an object space are recorded in the **object space header**.

For a **sequential** object space, the Object Manager does not synchronize access to the object space; there is no lock associated with the object space. Every request to map the Object Space to a client's address space is granted without delay. The higher layer is expected to synchronize the access to the object space.

For a **non-recoverable** and **persistent** object space, the Object Manager does not create a temporary copy in the **Shadow Segment** when the transaction commits the changes on the object space. The Object Manager writes directly to the original copy on disk. If the machine fails during the execution of the commit procedure, some modified data may get written out while others may not, thus the object space may become inconsistent with respect to the changes made within the transaction.

For a **non-recoverable** and **volatile** object space, the Object Manager does not create a permanent copy of the object space on disk since it is a volatile object space. In fact, there is only one single copy in memory manipulated by a transaction. If the transaction aborts half way through the operations on the object space, the changes on the object space remain since it is **non-recoverable**.

For a **recoverable** and **volatile** object space, the Object Manager maintains an original copy in memory in the **obmspace**. When a transaction maps for **Write** on the object space, a separate copy in memory is given to the transaction. This way guarantees that if the transaction crashes half way through its execution, the object space is reverted to its original state before the transaction. The original copy of the object space in **obmspace** remains in
tact. After the transaction commits, the updated copy of the object space replaces the original copy. Recall that for a recoverable and persistent object space, the Object Manager does not create a separate copy in memory when an object space is mapped by a transaction. The original copy in the obmspace is physically shared mapped into the transaction's address space. If the transaction crashes half way through its execution or the transaction aborts, the Object Manager must get a new original copy from disk. The Object Manager need not create a separate copy in memory since there is an original copy on disk.

A transaction may map a number of object spaces; some of which may be recoverable and some may be non-recoverable. A commit record is still needed to ensure the recoverability of the persistent and recoverable object spaces. If a transaction only makes changes to non-recoverable object spaces, it does not need a commit record.
Chapter 6

Recovery Management

As the number of components in a distributed system increases, the possibility of failures also increases. A fault-tolerant distributed system must deal with site failures and communication failures. An essential property of a fault-tolerant distributed operating system is its ability to confine the effects of failure and to allow the rest of the system to function normally. Raven accomplishes fault-tolerance by implementing a uniform, system-wide recovery management system.

Raven recovery management is implemented by a Recovery Manager inside the Object Manager. The Recovery Manager implements a transaction-based recovery using shadowing instead of logging schemes. Shadowing is deemed more suitable to our recovery scheme since the Object Manager also handles virtual memory management. Paging out due to transaction commits and paging out due to page-faults are handled by the same mechanism within the Object Manager and the dirty pages are written to the same paging storage, the Shadow Segment. Committing a transaction is simpler using shadowing scheme since the Object Manager only needs to check which pages of the object spaces are dirtied by a transaction and flushed out these pages to the Shadow Segment. Checking dirty pages is made simpler if paging hardware is available. When a transaction is committed, the Object Manager simply updates the blockmap of the segment descriptor that contains the object
space. If \textit{logging} is used instead, committing a transaction requires the Object Manager to check which instances have been modified through the Object Table, and write out the new values of the instances in a log on disk perhaps together with a commit record in the same log. When the transaction is finally committed, the new values of the instances are written back to its original locations on disk. Using \textit{logging}, the Object Manager could not take advantage of the fact that some modified instances have been written out to disk due to \textit{paging} since the paging storage is normally different from the logging storage on disk. In fact, most database systems that implement logging are not aware of virtual memory management. In these systems, paging are hidden inside the kernel.

During recovery after failures, less work is done by the recovery process if \textit{shadowing} is used. The recovery process only needs to read in the commit record associated with the transaction from disk and redo the updates on the blockmaps of the affected segments. Updating blockmaps must be an idempotent operation. If \textit{value-logging} is used, the recovery process must read in the modified values in the log from disk and redo the updates on the original copies on disk. Updating blockmaps located in segment descriptor pages require less operations than updating original instances scattered across many disk pages.

The major argument against using \textit{shadowing} scheme for recovery is that shadowing destroys the locality of reference while \textit{logging} does not [Gray81]. Logging typically requires some form of log file to record the changes made on data records. When the transaction is committed, the changes are copied back to the original records on disk. Therefore, the location of original records on disk is not affected by a transaction. However, it requires disk copy operations from the log file to the original records. Our shadowing scheme does not require any copy operations after a transaction is committed. The Object Manager simply updates the block map of a segment to point to the new instance pages and frees the original pages.

The argument in favor of logging is usually made for database management systems
where *locality of reference* among the related database records is crucial to the overall performance of the system. Furthermore, most database management systems do not take into accounts virtual memory management so *shadowing* yield no significant benefits over *logging*. For Raven, locality of reference is less important than the database systems. The performance advantages gained from integrating *shadowing* scheme and virtual memory management are expected to outweigh the lack of *locality of reference*. How important *locality of reference* is in Raven may not be known until some data on the usage patterns of object spaces are available.

### 6.1 Management of Commit Records

Central to failure recovery is the existence of some records on secondary storage that provide information about the system's state prior to failure. The recovery protocol makes use of this information to bring the system into a consistent state during recovery. Such stable records are normally referred to as *commit records* when describing failure recovery for transactions. A *commit record* contains sufficient information about a committing transaction such that failures during the transaction commit or abort will not jeopardize the atomicity property of the transaction. To reiterate from previous chapter, not all transactions require commit records. For example, a transaction that only modifies a single disk page does not need a commit record assuming a single disk page can be written out to disk atomically; a transaction that only modifies *non-recoverable* object spaces does not need a commit record.

In Raven, commit records are maintained by the Recovery Manager inside the Object Manager. The commit records are stored in a special object space with *segment id 2* where each instance in the object space is a commit record. Literature describing transaction management systems usually assumes that commit records are stored in *stable storage* [Lamp79] if such storage is implemented. *Stable storage* is implemented such that the probability that
A Raven commit record contains the following information about the committing transaction (see Figure 6.1):

- a commit id: to identify the commit record. It is simply an index number into the Object Table for the corresponding object instance.
• the *state* of transaction.

• the *type* of transaction.

• the *transaction id*

• the *parent transaction id* if the transaction is a child transaction.

• the number of object spaces modified by the transaction.

• the list of *osidpagesets* where each *osidpageset* contains the following information about a modified object space:

  – OSID of the object space,

  – the number of modified page records,

  – the list of modified page records. Each modified page record contains

    1. the starting disk page number of page block in the original segment,

    2. the starting disk page number of the temporary page block in the Shadow Segment,

    3. and the number of pages in the block.

the number of subtransactions. If the transaction is a *coordinator* transaction, then it has a number of subtransactions.

the list of subtransactions’ ids.

the number of participant machines that have not been informed about the latest state of the transaction. We refer to this field as the *mcount* field hereon. The field indicates when the Recovery Manager could free the commit record. When a coordinator transaction commits, it broadcasts *commit* messages to all the participant machines. The Object Servers of the participant machines send acknowledgements back to the coordinator. For each participant
machine that sends back an acknowledgement, the transaction decrements \textit{mcount} by one. When \textit{mcount} drops to 0, that means all the participant machines have learned about the latest state of the transaction; therefore, the Recovery Manager can delete the commit record.

### 6.2 Recovery Manager's Services

Aside from performing recovery after failure, the Recovery Manager provides the following primitives during normal transaction commits. Worker processes send requests to the Recovery Manager via these stub primitives.

- **CommitId**  
  
  \texttt{CreateCommitRecord (TransControlBlock *tblock, OsidPageSet *listofosidpageset, int numofpageset, TransId *listofsubtrans, int numofsubtrans, TransType type of transaction, TransState stateoftransaction)}

requests the Recovery Manager to create a commit record for the given transaction. The Recovery Manager returns the \textit{commit id} to the committing transaction. Only the Recovery Manager is aware of the internal structure of a commit record. Recovery Manager exports the \textit{commit id} to the external world to identify a particular commit record. Based on the \textit{listofsubtrans}, the Recovery Manager could determine the number of participant machines involved in the transaction; therefore, it could set the \textit{mcount} value in the commit record. The newly created commit record is secured on disk before control is returned to the requestor.
CHAPTER 6. RECOVERY MANAGEMENT

- **DeleteCommitRecord** (*CommitId*)
  explicitly requests the Recovery Manager to delete the commit record, allowing the Recovery Manager to claim back the storage allocated for the commit record. The coordinator process in charge of the committing procedure knows when it could delete the commit record associated with the committing transaction. A Commit Record could also be implicitly deleted by the Recovery Manager when it finds that the *mcount* of a commit record has dropped to zero.

- **DecrementMCount** (*CommitId*, number of informed machines)
  requests the Recovery Manager to decrement the *mcount* value by the number of machines already informed. When a transaction commits, the coordinator process keeps count of the number of participant machines that have learnt the final state of the transaction. If the coordinator learns that at least one of the participant machines does not return acknowledgements, it could not delete the commit record. Instead, it decrements the *mcount* so that the Recovery Manager knows the number of still uninformed participant machines. The Recovery Manager is now responsible for deleting the commit record when the *mcount* value drops to zero. How the *mcount* value of a commit record drops to zero is described in the next section.

- **IncludeSubTransList** (*CommitId*, subtransidlist, numofsubtrans)
  includes the additional subtransaction ids into the commit record's subtransaction ids list. When a commit record is created by a coordinator transaction, the subtransaction ids list passed to the Recovery Manager only contains the *tids* of subtransactions generated due to *remote invocations*. The coordinator process at this moment has not learnt about the *tids* of subtransactions due to *remote mappings*. Recall that subtransactions for remote mappings are only created when the coordinator process sends back changes on the object space to the remote server during the first phase of the two-phase
commit protocol. When the coordinator receives back all the *tids* of subtransactions due to *remote mappings*, it requests the Recovery Manager to include the subtransaction ids into the commit record. Consequently, when a commit record is created, the Recovery Manager must allocate enough space to store these extra subtransaction ids that come later. The parameter *numofsubtrans* passed to CreateCommitRecord is the maximum subtransactions expected including the subtransactions due to *remote mappings*. The request is normally followed by SetCommitRecord which will flush the commit record to disk.

- **SetCommitRecord (CommitId, final state of transaction)**
  sets the final state of the transaction in the commit record and writes out the commit record to disk. The coordinator process invokes this routine when it decides to commit or to abort the transaction. Once the commit record is secured on disk, control is returned back to the coordinator process and the transaction state is permanent. When a commit record is created, the transaction state is normally set to *prepared* and the commit record is flushed out to disk.

### 6.3 Recovering Transactions after a Machine’s Failure

There are many possible modes of failures during a transaction commit. This section describes how the Recovery Manager recovers from each failure case. One of the salient features of the Object Manager's design is that recovery could be done in the background while Object Manager proceeds with the normal operations after failures. The Recovery Manager process runs concurrently with other client processes. When a machine restarts, the Object Server process creates a set of worker processes and a *Recovery Manager* process. The Recovery Manager process itself creates a number of child processes to perform recovery. Figure 6.2 shows the creation of these child processes. The Recovery Manager process
Figure 6.2: Recovery Processes
executes the following protocol after it is created:

1. Read the object space that contains the commit records from disk.

2. For each commit record left in the object space, there may be unfinished work on the committing transaction and the Recovery Manager must complete the work to ensure atomicity of the transaction.

3. For each commit record, the Recovery Manager process creates a recovery process to performs recovery on the transaction associated with that particular commit record. This improves the concurrency of recovery processing and frees the Recovery Manager process to perform other normal functions such as creating commit records, writing commit records etc.

4. The first thing a recovery process does is to request the Object Server process to lock all local object spaces modified by the unfinished transaction. Note that all the OSIDs recorded in the commit record are local object spaces. The locks are released once the committing process is done. This prevents other clients from mapping the object spaces until the object spaces are brought to a consistent state. Consequently, there must be a moment lapse before an Object Server could start receiving requests from clients outside the Object Manager. The Recovery Manager sends an indication message to the Object Server once all the object spaces in questioned have been locked so the Object Server could start perform regular operations.

5. Once all the object spaces in questioned have been locked, recovery could run in the background while the Object Server receives requests from external clients.

6.3.1 Recovering Coordinator Transactions

If the unfinished transaction is a Coordinator, there are three cases to consider based on the state's of the transaction:
• Case 1:
If it is in prepared state, the recovery process must abort the transaction since the transaction may have remote-mapped some object space and the state’s of the object space may have been lost before it reaches the remote Object Server during the first-phase of the commit protocol. It executes the following protocol:

1. The recovery process sets the transaction state’s in the commit record to aborted
2. and performs the second-phase of the protocol by sending abort messages to each participant machine. The execution of the second phase is identical to the execution during normal operation. The recovery process also creates a set of slave processes where each slave process corresponds to each participant machine to be informed. The commit record is retained until all participant machines are informed about the final state of the transaction.
3. If there are pages allocated for the transaction in the Shadow Segment, the recovery process frees the pages while its slave processes are sending out abort messages to the participants.
4. The recovery process terminates after executing the second phase of the protocol.

• case 2:
If the transaction is in committed state, the coordinator has been committed before the machine crashed. The recovery process repeats the execution of the second phase of the protocol by informing the transaction’s participants of the final state of the transaction. The way it is done is identical to the above protocol. It also repeats the updates on the blockmaps of the segments modified by the transaction based on the osidpagesets recorded in the commit record.

• case 3:
If the transaction is in aborted state, the coordinator executes the same protocol as
the \textit{prepared} case with the exception that it does not have to set the transaction state in the commit record to \textit{aborted}.

\section*{6.3.2 Recovering Participant Transactions}

A slightly more complicated recovery protocol is run if the transaction is a participant of some parent transaction. A participant machine could not be informed about the final state of the transaction if the machine has crashed or has been partitioned from the coordinator machine. There are three cases to consider as well:

- \textbf{case 1:}

If the participant is in \textit{prepared} state, the recovery process executes the following protocol:

1. It sends an \textit{inquire parent status} message to the parent transaction (coordinator). Recall that one of the field in the commit record stores the parent transaction id.

2. When the Object Server in the parent's machine receives an \textit{inquire parent status} message, it checks if it still has the \textit{transaction control block} associated with the inquired transaction. If so, it replies directly to the inquiring process; otherwise, it consults the Recovery Manager inquiring the status of the transaction.

3. If the Recovery Manager has a commit record associated with the transaction, it returns the state of the transaction to the Object Server; otherwise, it returns an indication that there is no commit record associated with the transaction. If there is no commit record associated with the transaction, the Object Server could safely conclude that the transaction must have crashed before committing; therefore, it is an aborted transaction. The Object Server then replies to the inquiring process.
4. Upon learning the final state of its parent transaction, the recovery process could proceed with the second phase of the commit protocol for the participant. For example, if the final state of the parent transaction is committed, then recovery process could commit the participant transaction. If the participant transaction happens to have a number of subtransactions as well, it will then inform its own subtransactions according to the second phase of the protocol.

5. The recovery process also sends back an acknowledgement to the parent transaction machine indicating that it has learnt the final state of the parent transaction.

6. Upon receiving the acknowledgement, the Object Server in the parent machine could then informs its Recovery Manager to decrement the \textit{mcount} value. When the value becomes zero, the Recovery Manager automatically deletes the commit record.

If the recovery process could not reach the parent's machine, it could not proceed with the recovery of the participant at this moment. It has to wait for the parent machine to come up again and resends the \textit{inquire parent status} message to the parent machine. In principle, if the recovery process could not contact the parent transaction, it could broadcast an inquire message to the peer participants in case the peer participants know about the state of the parent. However, this assumes that participants must retain the information about their parent long after they are committed or aborted just in case other participants have not learnt about the final state of the parent. Maintaining and knowing when to delete these information could be quite complicated.

- \textbf{case 2:}

  If the participant is in \textit{committed} state, the recovery process simply redoes the committing procedure by updating the blockmaps of the affected segments based on the \textit{osidpagesets} recorded in the commit record. If the participant has a number of sub-
transactions, the recovery process executes the second phase of the protocol, informing its child transactions of the final state of the participant.

- case 3:

If the participant is in aborted state, the recovery process simply redoes the aborting procedure according to the second phase of the commit protocol.

Occasionally, when the Recovery Manager receives an inquire message about a transaction state, it could not immediately reply to the inquiring process if the transaction in questioned is also going through its own recovery procedure. The inquiring process is blocked until the inquired transaction's state is determined. A typical case is if the transaction is of type CoordinatorParticipant and its machine has crashed while the transaction is in prepared state, and at the same time some of its participant machines also crash. While the recovery process for the CoordinatorParticipant transaction is sending an inquire message to the parent transaction, its own child transaction may send an inquire message to the CoordinatorParticipant transaction. In this case, the Recovery Manager must wait until the recovery process has finalized the state of the CoordinatorParticipant before replying to the child transaction.

Figure 6.3 illustrates the communications involved in recovering a committing transaction after a site's crashes and reboots.

6.4 Updating Segments' BlockMaps During a Transaction Commit

After a transaction is committed, i.e., its commit record has been written out to disk, the committing process replaces the new pages in the Shadow Segment with the pages in the original segments. Recall that these new pages have been secured on disk. To replace these pages, the committing process updates the corresponding blockmaps of the original
Site #1

Object Server

Recovery Manager

Inquire transaction status

Site #2 crashes and recovers

Object Server

Recovery Manager

Inquire Parent state

3. Send back the status

a recovery process

1. Lock Osids

p1 p2

Site #3

Object Server

Site #4

Object Server

4. Commit or Abort the child trans

Transaction Tree

transaction at site 1

transaction at site 2

transaction at site 3

transaction at site 4

Figure 6.3: Recovering a Participant Transaction
segments to point to the new pages. This update is done in such a way that when a failure occurs while updating the blockmaps of the segments, the consistency of the data on disk is not compromised. The following steps are executed when updating the block maps on disk assuming the block maps have been updated in memory:

1. Write out the segment descriptor page that contains the block map which now points to some new pages. If there is more than one object space modified, step 1 is repeated until all block maps are written out.

2. Write out the segment descriptor page of the Shadow Segment. Once the page is written out, the Shadow Segment no longer owns the new pages.

3. Write out the page(s) that contain the free page bitmap. The bitmap pages need to be written out since the old pages of the segment descriptors are freed by the Replace() primitive described in Section 3.1. After step 3 is executed, the old pages of the segment descriptors are permanently returned back to free page pool.

If a failure occurs during the execution of step 1, a segment descriptor's block map has either pointed to all new pages or not at all since the block map is in one disk page which could be written out atomically. When the recovery process calls Replace for the segment descriptor, the Replace primitive checks if the corresponding block map entries already point to the new pages. To check this, Replace primitive simply checks the physical disk addresses contained in the block map entries. If these physical addresses happen to be the same as the ones in the Shadow Segment, then the block map entries already point to the new pages; therefore, Replace does not update the segment descriptor block map again, but it still needs to update the Shadow Segment's block map since the Shadow Segment descriptor page did not get written out when the failure occurs. Note that the Shadow Segment descriptor page is written out only after the other segment descriptor pages are written out. This is to ensure that if failure occurs while writing out the other segment
descriptor pages, the new pages are still retained in the Shadow Segment. By the time the Shadow Segment descriptor page is written out, other segment descriptor pages have pointed to the new pages. Therefore, the new pages are not lost even if failure occurs after the Shadow Segment descriptor page is written out.

If a failure occurs anytime before step 3 is completed, a number of old pages that are freed by Replace primitive may not get returned to the free page pool since the free page bitmap pages were not written out to disk yet when the failure occurs. The fact that the old pages have been freed was not recorded on disk. This problem does not cause data inconsistency, but it does require the Storage Module to occasionally perform garbage collections on some of these orphan pages.

One way to reclaim back these orphan pages is to create a new free page bitmap and scan through the Segment Table to check which disk pages are currently allocated to segments. For those pages that have been allocated, their corresponding bits in the bitmap are turned off. The disk pages whose corresponding bits in the bitmap are On belong to the free page pool.

Figures 6.4 to 6.7 show the updates done on the blockmaps when a transaction is committed.

6.5 Managing Object Space Creations and Deletions Across Failures

The previous section describes how the blockmap of an object space is updated when its instance pages have been modified by a transaction. The mechanism to ensure consistency of updates across machine failures is simpler in the case of updating instance pages only than in the case of creating and deleting object spaces. This section elaborates how the Object Manager ensures data consistency across failures when object spaces are created or deleted by a transaction.
CHAPTER 6. RECOVERY MANAGEMENT

Figure 6.4: Layout of blockmaps on disk before the update

Figure 6.5: Layout of blockmaps on disk after step 1
CHAPTER 6. RECOVERY MANAGEMENT

Figure 6.6: Layout of blockmaps on disk after step 2

Figure 6.7: Layout of blockmaps on disk after step 3
Since we allow recovery processing to run concurrently with normal operations, we have to ensure that an object space that was deleted by a committing transaction before the failure does not get mapped into a new client’s space after the machine reboots. We also have to ensure that a segment entry in the Segment Table allocated to a new object space does not get allocated again to another client because of machine failures. We achieve this using the status field of the Directory.

Recall from Section 3.4, one of the fields in a directory entry is the status field which indicates the latest status of the object space. An object space could assume one of the following paired states only:

1. **Old-Committed**: *Old* indicates that the necessary update with regards to the existence of this object space, namely the marking of the corresponding Segment Table entry as *occupied* has been secured on disk. Since it is marked *Old*, the object space is probably created long before the machine crashed. *Committed* indicates that the transaction that creates this object space has been committed. Actually, the *Committed* is redundant since the transaction that created this object space must have committed because the object space is marked *Old*. However, we include them since the rest of the combinations are paired.

2. **New-Prepared**: indicates that the object space has just been created by a transaction and the transaction was in *prepared* state when the machine crashed. Depending on the final outcome of the transaction during recovery, the next possible state of this object space is either *Old-Committed* or gets removed from the object space.

3. **New-Committed**: indicates that the object space is a newly created object space and the transaction that creates the object space has been committed. New-Committed state tells the Object Manager that it has not broadcasted the newly created object space’s OSID to other machines.
4. **Deleted-Prepared**: indicates that the object space is to be deleted by a transaction, but the transaction itself is still in *prepared* state when the machine crashed. Depending on the final outcome of the transaction after a recovery is performed, the next possible state for this object space is either *Old-Committed* or *Deleted-Committed*. If the recovery process aborts the transaction, the object space is not deleted. The state of the object space is set back to *Old-Committed*. Note that when an object space is at this state, its corresponding entry in the Segment Table is still marked as *occupied*. This will ensure that the Object Manager does not allocate this segment entry to another client. For deleting an object space, the marking on the Segment Table is only done when the object space is in Deleted-Committed state. For creating a new object space, the marking on the Segment Table is done when the object space is in New-Prepared state. In fact, the marking of the Object Table as *occupied* is written out to disk before the directory entry is written out to disk. Since there may be a machine failure between the transaction prepared and committed state, the marking of the Segment Table ensures that the Object Manager does not allocate the same entry to another client after the machine reboots. If the recovery process commits the transaction, the object space is deleted and its entry is removed. To ensure that a machine crash during the execution of the recovery process does not jeopardize the consistency of the directory or the Segment Table, the next logical state for the object space in this case is *Deleted-Committed* before it is actually removed from disk.

5. **Deleted-Committed**: indicates that the object space is deleted and the transaction that deletes the object space has been committed. The recovery process proceeds to unmark the Segment Table (if it is not already unmarked before the machine crashed) and remove the entry from the directory.

To map an object space to a client's space, the Object Manager first checks the cor-
responding OSID entry in the directory. If the entry is marked as Old-Committed, or Deleted-Aborted, the Object Manager could safely map the object space to the client's space. If the entry is Deleted-Committed, the Object Manager could inform the client that object space has been deleted. If the object space is in other states, the client must wait until the final state of the object space is determined by the recovery process. This will ensure that an object space deleted by a committed transaction does not get mapped into another client's address space after a machine failure. The recovery process knows which object spaces have been created or deleted by a transaction since the commit record contains the list of newly created OSIDs and deleted OSIDs (see Figure 6.1). Based on this information, it requests the Object Manager to update the directory and the Segment Table accordingly.

The following example shows the steps taken to commit a transaction that creates object spaces:

1. Transaction A creates two object spaces and the Object Manager allocates Segment 5 and Segment 6 for these object spaces. The Object Manager simply marks the corresponding Segment Table entries as \textit{occupied}. Note that this marking is done in memory (no disk write yet).

2. Transaction A prepares to commit by executing the following steps in sequence:
   
   (a) write out the corresponding segment pages (entry 5 and 6 of the Segment Table) to disk.
   
   (b) set the corresponding directory entries as New-Prepared and write out these entries.
   
   (c) write out a commit record which, among other things, contains the OSIDs of the newly created object spaces. At this point, the transaction is in prepared state.

3. If transaction A decides to commit, it executes the following steps in sequence:
(a) set the transaction state in the commit record on disk as COMMITTED.

(b) set the corresponding directory entries as New-Committed and write out these entries. Actually, we could have set the state to Old-Committed and write out these entries instead. The only purpose to distinguish between New-Committed and Old-Committed is for the Object Manager to know if it has broadcasted the newly created object space to other machines. Once this newly created object space is broadcasted, its entry is set to Old-Committed.

4. If transaction A decides to abort, it executes the following steps in sequence:

(a) set the transaction state in the commit record on disk as ABORTED.

(b) remove the entries from the directory.

(c) free the corresponding Segment Table entries.

Similar steps are taken for a transaction that deletes an object space. The only difference is that the Segment Table entry is not freed until the transaction is in Committed state (no marking on Segment Table on disk until A is in committed state). This ensures that the Segment Table entry is not allocated to other clients during recovery of a prepared transaction. Because the directory entry records the state of an object space as Deleted, other client cannot map this object space during recovery processing.
Chapter 7

Concluding Remarks

This thesis work represents the first design and implementations of Raven’s underlying system. As such, it should be viewed as a prototype system developed to gain some experiences and feedback on the overall Raven design. Our implementation of the Object Manager was somewhat hampered by the lack of supports from the Raven kernel and communication manager. Consequently, substantial implementation work was first devoted to providing a working kernel and minimal communication services for remote IPC before the design and implementation of Object Manager could begin. The Object Managers have been implemented and are running on the UNIX 4.3 BSD system.

7.1 Summary

In this thesis, we introduce the notion of object spaces as physical storage abstraction for objects. The use of object spaces as a unit of storage and recovery for objects simplifies implementation on the transaction facilities that support these objects.

Our approach to divide an object space into two layouts is based on the expectation that most object spaces should fit into one disk page. By grouping data together with the directory that maintains the data into one disk page, we expect to achieve better
CHAPTER 7. CONCLUDING REMARKS

performance results than separating the directory from data. Performance measurements on UNIX files support our claims [Mull84]. Whether this approach will truly yield better performance results depends much on how the object spaces are used by higher layers that ultimately determines the average size of object spaces.

The identical layout of an object space regardless of whether it is in memory or in secondary storage and the involvement of Object Manager in paging decisions simplify object mapping operations. Furthermore, it also reduces the number of disk writes during a transaction commit.

By attaching atomicity attributes to object spaces, we reduce the overheads incurred by the heavy-weight transaction mechanism.

The design allows recovery to run concurrently with normal operations to hide effects of recovery processing from users.

7.2 Future Work

We have designed and implemented the underlying layers of Raven up to the Object Manager level. The next logical step is to build Class Manager and Invocation Managers on top of the Object Manager. Once the Class Manager and Invocation Managers are implemented, we could evaluate the performance of Object Manager and gain some feedbacks on the design decisions made on the Object Manager, especially with regards to the layout of object spaces and the effects of shadowing with respect to locality of reference of objects.

To accurately evaluate the performance of Raven system, Raven should be ported to a native machine instead of running on top of UNIX BSD 4.3. Modifications on the kernel are needed when porting to a native machine. We identify three main areas of work needed when porting to a native machine:

• The kernel memory management has to be rewritten to take advantage of any memory
management hardware available in the native machine. The interface to the users could remain the same. The current implementation does not make use of any memory management hardware. Consequently, as such, all memory pages are physical pages and they are not paged out until explicitly released. This severely affects our testing of virtual memory management in the Object Manager since the current implementation does not generate page faults. The virtual memory management code in the Object Manager remain the same regardless of whether the kernel memory management has hardware supports or not.

- The current Communication Manager's services are minimal at best although it seems to be sufficient for Object Managers to communicate with each others. It relies on the UNIX datagram sockets to provide physical links among the Raven machines. If Raven is ported to a native machine, a new communications package need to be written. However, writing a complete communications package requires considerable amount of work that may hamper the research work in other areas of Raven system.

- The disk server on the kernel needs to be rewritten since the current implementation uses a UNIX file to simulate a disk storage.

Object space migration is not implemented yet although we suspect it will be quite simple since the mechanisms to update object space directory and to send object spaces across machines are already in place.

Once the Class Manager and Invocation Managers are in place, Raven can serve as a basis for further researchs in other areas of distributed systems such as system configurations, load balancing and resilience of the system. One final note, we have attached object spaces with three separate attributes: sequential or concurrent, persistent or non-persistent, and recoverable or non-recoverable to attempt to identify its data consistency requirements. How useful these attributes are to the the higher layers and how the separation of these properties
improves the system performance remain to be an interesting area to be explored.
Bibliography


BIBLIOGRAPHY


