Symantec Yellow Books™

The Veritas™ Cluster File System: Technology and Usage

Optimize utilization, performance, and availability with Symantec's scalable shared file storage infrastructure

symantec™

The Veritas Cluster File System: Technology and Usage

Optimize utilization, performance, and availability with Symantec's scalable shared file storage infrastructure

Symantec Corporation 20330 Stevens Creek Blvd. Cupertino, CA 95014 USA

http://www.symantec.com

ISBN 0-321-446100

Contents

Why a Cluster File System?

Distributed applications with dynamic computing and I/O demands benefit greatly from simultaneous access to shared data from multiple servers, particularly if performance continues to scale nearly linearly when new applications and servers are added to the mix. Symantec's primary motivation in developing the Cluster File System (CFS) has been to meet the file storage requirements for this high growth market, as major applications evolve to be more distributed, taking advantage of modern datacenter "scale out" architectures.

Today, the most popular use case for CFS is active-passive clustering of single node applications. In this scenario, CFS enables failover times that are as much as 90% lower than single node file system solutions. Increasingly, however, the growth in CFS deployments is coming from applications that take advantage of the consistent simultaneous access to file data from multiple cluster nodes. These applications include clustered database management systems, messaging applications, workflow managers, video streaming, risk analysis, business intelligence, and more.

In addition, an increasing number of users are replacing NFS servers with CFS clusters that run applications directly on the servers that provide file access, for improved reliability and data consistency, as well as elimination of the network bottlenecks often seen with NFS based file storage solutions.

Other users retain the NFS file server data center architecture, but replace NAS systems with CFS-based CNFS servers to improve price-performance, scalability and flexibility. In effect, with the introduction of CNFS, CFS is commoditizing NAS while it improves scalability and quality of service.

Still other users choose CFS because it supports the top three enterprise UNIX platforms as well as Linux. This enables them to standardize data management and operating procedures across the data center, no matter which UNIX platforms they use. Finally, some users choose CFS-based solutions because they support more disk arrays than any other suite, making it possible for users to fully exploit their storage hardware investments.

While users deploy CFS for a variety of reasons, on closer inspection, there are

common file storage and management requirements that make CFS an ideal solution in a wide variety of situations.

Using clustering to scale out an application or a data center adds a dimension to the sizing and tuning complexities found in single-server environments. File sharing, distributed decision making, I/O workload asymmetry among cluster nodes, migration of applications from node to node, are among the variables that can make "getting it right" an arduous task.

Numerous interviews with CFS users have made it apparent that application specific guidance for CFS deployment and knowledge of how CFS works and interacts with other components in the storage stack are high on the list of users' concerns. Primarily for that reason, Symantec's File System Solutions team undertook the creation of this book, with the goal of putting the whole story, from technology to administration, to use cases, in one place. It is our sincere hope that the result addresses the concerns of our users, present and future.

As the product manager for CFS, my thanks go out to the author, the management team that supported this effort by granting people the flexibility to take time from their "day jobs" to participate in the project, and the individuals who patiently provided the knowledge and painstakingly reviewed the manuscript.

David Noy
Regional Product Manager, EMEA
Symantec Corporation
December 2009

About this book

This book describes Symantec's Veritas Cluster File System (CFS), an implementation of the cluster architectural model for sharing data among multiple computers. Based on Symantec's long-proven Storage Foundation File System (commonly known as VxFS), CFS is exceptionally scalable, robust, and high-performing. Building on a solid base of VxFS technology, CFS adds cluster-wide cache coherency, distributed resource control and file system transactions, and other features that enable scaling and enhance performance and load balancing across a cluster of application or database servers.

An astute reviewer noted that the book seems to address different audiences, and that is indeed the case. The Introduction sets the stage by defining the problem of sharing data among multiple application servers and contrasts the CFS approach with other commonly encountered solutions.

Part I presents an overview of CFS, with emphasis on the features that make it unique and the capabilities they enable, along with some examples of scenarios in which CFS is particularly suitable. It should be of interest to new application designers who require an appreciation of online data management in the data center environment.

Part II describes the CFS "secret sauce"-the internal architecture that gives CFS its unique combination of scalability, flexibility, robustness, and performance. It should be of interest to experienced developers and administrators who need to understand what's going on "under the hood."

Part III is a guide to installing CFS and tuning file systems. Multi-server cluster environments are inherently complex, as are their file I/O requirements. This part should be of interest to system administrators who are either installing CFS for the first time or are tasked with monitoring and managing the performance of a CFS cluster.

Most of the material is available in some form from other sources. The unique contribution of this book is to "pull it all together," and to answer the question, "why" in addition to describing the "what."

Books tend to live longer than software product versions. While every effort has been made to ensure the accuracy of the material, you should consult current product documentation and support sources before putting the principles and techniques described herein into practice.

Contributors

This book is a collaborative effort of many experts in CFS, and indeed, in the field of computer file systems at large. The following are the principal technical contributors:

Jobi Ariyamannil
Brad Boyer
Sai Chivukula
Bala Kumaresan
David Noy
Pramodh Pisupati
Hal Prince
Mike Root
Meher Shah

Reviewers

The following technical experts reviewed the manuscript intensively through the course of several drafts:

Sushil Agarwal
Jobi Ariyamannil
Brad Boyer
Grace Chen
Colin Eldridge
Bala Kumaresan
Murthy Mamidi
Hal Prince
Dilip Ranade
Karthik Ramamurthy
Mike Root
Chuck Silvers
Bhavin Thaker
Patric Uebele

File systems for data sharing

This chapter includes the following topics:

The role of file systems in information technology

Enterprises of all kinds increasingly depend on their digital data assets to operate. But the business world has moved beyond simple dependence on data availability. As information processing operations integrate across the enterprise, not only must data sets be highly available, they must also be readily accessible to multiple business applications running on different computers. Moreover, as the increasing velocity of business change is reflected in information processing, the stable, unchanging data center is becoming a thing of the past. Instant reaction to rapidly changing business conditions and computing requirements is a must. In short, the digital data that enterprises need to operate must be both highly available and simultaneously accessible to an ever-changing array of applications and servers

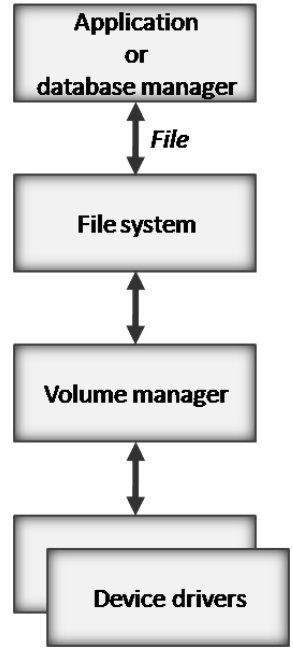
Moreover, the storage that holds business-critical data must be scalable-able to grow and shrink, both in size and accessibility-as requirements fluctuate.

From a technology standpoint, this means that key data sets must be simultaneously accessible by multiple computers and applications, in such a way that each one perceives itself as the sole user of the data.

The file system in the I/O software stack

The typical application server software "I/O stack" illustrated in Figure Intro-1 can be a useful aid to appreciating the advantages and limitations of different approaches to sharing data among computers and applications.

Figure Intro-1 The file system's position in the application server software



Nearly all applications deal with *files.* Files are a convenient representation of both business and technical objects-transactions, activity logs, reports, design documents, measurement traces, audio-visual clips, and so forth. Even the so-called "structured data" typically managed by relational database management systems is typically stored in container files.

The properties of files allow applications to deal with data much as people would deal with the business objects the data represents:

- **Flexibility.** Files are easily created, added to, truncated, moved, and deleted
- **User-friendliness.** Files' names are human and application-readable, and can be chosen for convenience
- **Hierarchical organization.** Files can be organized in hierarchies along departmental, application, project, temporal, data type, or other convenient lines for easy reference
- **Security.** Files can be protected against both tampering and physical destruction to a greater or lesser degree, depending on their value and the cost of protection

Virtualization in the I/O stack

As Figure Intro-1 suggests, applications manipulate files by requesting services from a *file system*[1]. The role of the file system is to present an application-friendly file abstraction, and to implement it using a more primitive abstraction, the virtual block storage device.

1. The term *file system* is commonly used to refer to both (a) the body of software that manages one of more block storage spaces and presents the file abstraction to clients, and (b) a block storage space managed by such software and the files it contains. The intended meaning is usually clear from the context.

Virtual block storage devices resemble disk drives. To file systems, they appear as numbered sets of fixed-size "blocks" of persistent storage capacity[2] in which data can be stored and from which it can be retrieved. Functionally, they differ from physical disk drives primarily in that the number of blocks they contain can be increased (and in some cases reduced) administratively.

Virtual block storage devices are implemented by mapping their block numbers to corresponding blocks on physical storage devices-magnetic or solid state disk drives-in some way that achieves a desirable data resiliency, I/O performance, or flexibility effect. For example, virtual blocks may be mapped to striped, mirrored, or RAID disk configurations. Mapping virtual block numbers to block addresses on physical storage devices can be performed at different locations in the hardware stack:

- **Application server.** Volume managers run in application servers and coordinate the operation of devices they perceive as disks
- **Disk array.** Disk array control firmware manages the physical storage devices in the array
- **Storage network.** Firmware in intelligent network switches that connect disk arrays to application servers manages the already-virtualized devices presented by the arrays

There may be two or even three layers of block device virtualization in an I/O path. A disk array may create a RAID group of several disks and present them as logical units (LUNs) to a network switch, which mirrors them with LUNs presented by another array, and presents the resulting virtual LUN to a volume manager, which mirrors it with a directly attached solid state disk for performance and resiliency.

What's in a file system

The file system software that implements the file abstraction using the much more primitive virtual block device abstraction for persistent storage, is a complex component of the server I/O software stack with multiple functions:

- **Storage space management.** It manages one or more "flat" (sequentially numbered) spaces of virtual device blocks, allocating them to files and other structures as needed, and keeping track of those that are free for allocation
- **Name space management.** It implements a name space, in which an application can give a file any unique syntactically valid name. In most file

2. Persistent storage is storage whose contents *persist*, or last across periods of inactivity. Information written on paper is persistent, whereas information displayed on a video screen is not. In digital data storage terms, persistence usually means retention of contents in the absence of electrical power. Thus, magnetic disk drives and solid-state flash memories are persistent. Computers' dynamic random access memories (DRAMs) are not persistent.

systems, a file's full name represents the path on which it is reached when traversing the name space hierarchy. In a UNIX file system, for example, the file **/a/b/c** would be located by starting at the top of the hierarchy (**/**), and traversing directory **a**, which contains directory **b**. File **c** is situated within directory **b**. File **/a/b/c** is distinct from file **/a/d/c**, which has the same name (**c**), but is located on a different path (**/a/d**)

- **Security.** It enforces file ownership and access rights, granting applications read and write access to files only if they present the proper credentials
- **File data mapping.** It maps file data addresses to block numbers on the underlying virtual block devices so that each file appears to applications as a stream of consecutively numbered bytes, independent of the block device locations at which it is stored

In addition to these externally visible functions, file systems perform internal functions that are transparent to applications. These include caching of file system objects ("metadata" that describes certain properties of data such as its owner, its location, and so forth) and file data to enhance performance. File system designs assume that computer main memories can be accessed four or five orders of magnitude more quickly than persistent storage. For example, scanning a directory in search of a particular file name is much faster if some or all of the directory's contents are immediately accessible in memory ("cached"), and do not have to be read from disk.

File systems perform these tasks concurrently on behalf of as many applications and other system software components as are running in the system. Each application sees itself as the only user of files in the system, except where two or more mutually aware applications explicitly share access to files. Even in this case, file systems appear to execute application requests strictly in order. For example, if one application writes data to a file, and another reads the same file blocks shortly thereafter, the data returned is that written by the preceding application.

File systems manage this complexity by a combination of:

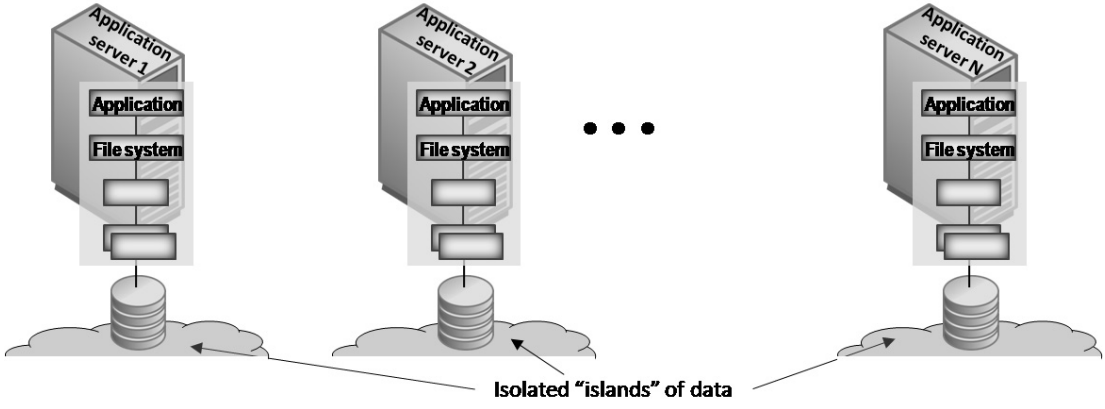
- **Careful writing.** Performing their internal operations in a strictly controlled order, particularly those that update file system metadata
- **Controlling access to resources.** Maintaining elaborate and constantly changing sets of locks that limit access to parts of the file system's data and metadata while it is in use, so that multiple accesses by uncoordinated applications do not corrupt data or deliver incorrect results

Typically, file system locks are in-memory data structures that record which data and metadata are "busy" at the moment, and which therefore cannot be perturbed by additional accesses from applications or from the file system itself.

The problem with files: data islands

While files are an extremely useful abstraction, implementing the abstraction in a file system that runs on a single application server ultimately creates an inherent limitation that is suggested by the graphic of Figure Intro-2. In Figure Intro-2, each set of virtual block storage devices is managed by a file system that runs in an application server. Each file system controls its storage devices, and presents a name space to all applications running in its server. There is no connection between the file systems on different servers, however. Each one presents a separate name space to the applications ("clients") that use it, creating uncorrelated "islands" of storage and data.

Figure Intro-2 "Islands" of data



Islands of storage and data are costly in several dimensions:

- **Storage cost.** Excess storage capacity connected to one application server is not readily available to other servers that might require it. In most cases, each application manager provisions storage for the worst case, resulting in overall waste
- **Bandwidth and time.** Data required by two or more applications must be copied between originating and consuming servers. Copying uses network bandwidth, and perhaps more important, takes time. The degree to which applications can be synchronized with each other is limited by the time required to copy data from one to another
- **Potential for error.** Keeping two or more copies of data on different "islands" synchronized is a fragile, error prone process. Moreover, each time applications, data structures, system or network configurations, or operating procedures change, data copying and other management procedures must be reviewed and changed as well. Infrequent, non-routine operations are generally the most error-prone, occasionally with catastrophic consequences

As suggested earlier, applications that process digital data are becoming increasingly integrated-from the point of origin of data, through editing and modification, analysis, reporting, action, and finally archiving. The “islands of data” scenario represented by Figure 2 is becoming correspondingly less acceptable. Enterprises need technology that enables many applications (or many cooperating instances of the same application) to access the data in a single file system, even if they are not all running on the same application server.

There are several architectural solutions to this problem of shared access to file systems. The most frequently encountered are:

- **Network Attached Storage (NAS)**
- **File Area Networks**
- **SAN file systems**
- **Cluster file systems**

Each solution has its strong and weak points, and consequently, classes of application for which it is more or less optimal. The sections that follow describe these file sharing solutions and enumerate their strengths and weaknesses.

Shared data file system architectures

File systems that enable data sharing by multiple client applications running on interconnected computers can generally be characterized by:

- **Functions performed by the client.** The file system tasks performed by the application server
- **Location of data access software.** The location of the file system software that accesses and transfers file data
- **Location of metadata management software.** The location of the file system software that manages file system metadata

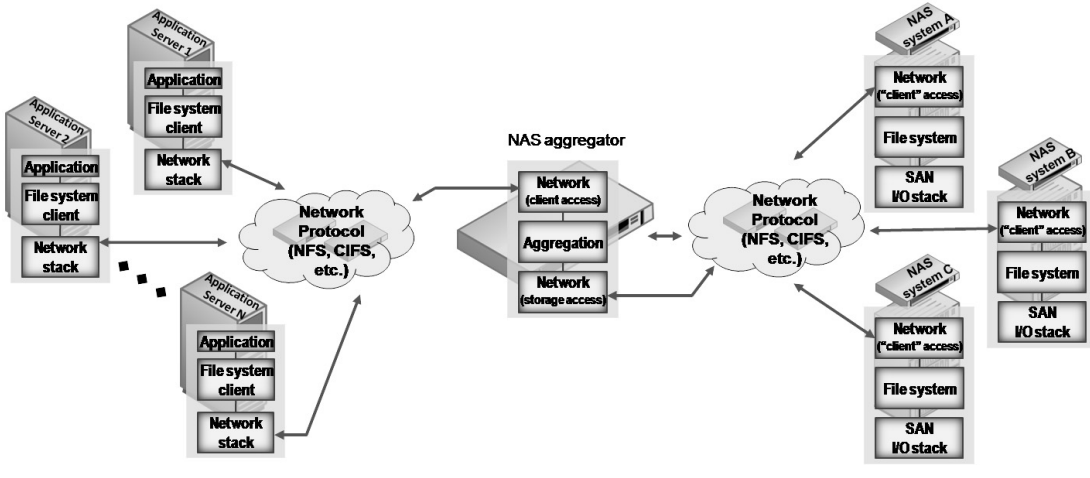
Shared data file system models have other characteristic properties, such as the network technologies they use, but these are generally either historical artifacts (for example Fibre Channel with SAN file systems or Ethernet with NAS systems) or market-driven implementation choices (for example, Infiniband for high-performance computing), rather than being inherent in the architectures. This section discusses the four most frequently-encountered shared data file system architectural models, including the cluster file system model implemented by Symantec’s Veritas Cluster File System (CFS).

Model 1: network-attached storage (NAS)

Perhaps the most frequently encountered shared data file system architecture is the network-attached storage (NAS) model depicted in Figure Intro-3. In a NAS system, a single computer (the NAS "head" in Figure Intro-3) manages file system metadata and accesses file data on behalf of clients. The NAS head, which in smaller configurations is often integrated with the disk drives it controls, communicates with client computers over a network (usually TCP/IP) using a file access protocol such as Network File System (NFS) or Common Internet File System (CIFS). To applications running on client computers, NAS-hosted file systems are essentially indistinguishable from local ones.

A file access client software component running in the client computer translates applications' file access requests expressed by operating system APIs such as POSIX into CIFS or NFS protocol messages and transmits them to the NAS system.

Figure Intro-3 Network-attached storage model



CIFS, NFS, and other network file access protocols express file access and metadata manipulation requests, which the NAS head executes on behalf of

clients. Table Intro-1 summarizes the characteristics of the NAS model.

Table Intro-1 Properties of the NAS model

Tasks performed by application server	Location of file data access software	Location of metadata management software
Translate applications' file access requests to the NAS protocol (typically NFS, CIFS, http, DAV, or similar)	NAS head	NAS head

Advantages and limitations of the NAS architectural model

The NAS model has become very popular for use with both UNIX and Windows application servers. Popularity has resulted in a high level of maturity (performance and robustness) and integration, for example, with network management and security tools. NAS systems are widely deployed as second-tier data storage, and are increasingly coming into use in more performance-critical applications.

As may be apparent from Figure Intro-3, however, the NAS model has certain inherent limitations:

- **Protocol overhead.** Applications running on NAS client computers use operating system APIs (POSIX or WIN32) to express file access requests. File access client software translates these into CIFS or NFS messages and sends them to the NAS head. The NAS head executes the requests using its own operating system APIs to access file system data and metadata. Thus, client requests are translated twice before being executed. Similarly, the NAS system's responses are translated twice as well.

  A more pronounced effect in some applications is data movement overhead. The CIFS and NFS protocols do not lend themselves to so-called "direct," or "zero-copy" file I/O, in which data is written directly from or read directly into application buffers. Typically, CIFS and NFS client software moves data between user buffers and the kernel operating system buffers from which I/O is done. Particularly for streaming applications, which read and write large blocks of data, the overhead of copying data to and from application buffers can be substantial.

- **Bottlenecking.** A NAS head is the single access point for the file systems it hosts. Thus, a NAS system's total performance is limited by the ability of the head to process requests and to absorb and deliver data. Data centers with more clients or greater I/O demands than a NAS system can satisfy must divide their workloads among multiple NAS systems, creating "islands" of unconnected data. Chapter 3 describes a novel capability of Symantec's

Veritas Cluster File System that relieves this limitation, the clustered NFS server, or CNFS.

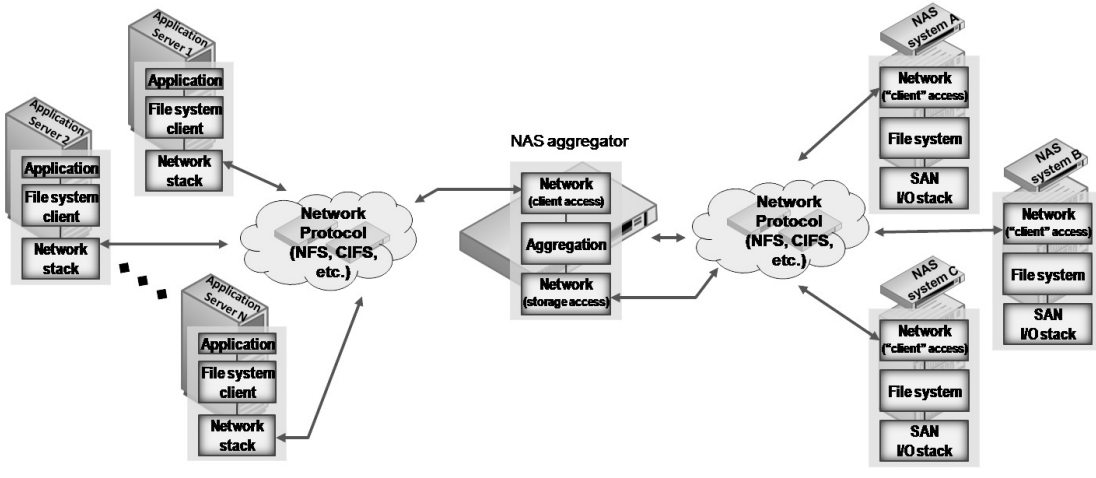
In addition to the inherent limitations of the model, NAS implementations have historically lacked certain features that are important for critical enterprise data, notably high availability (the capability of a system to sustain single component failures and continue to perform its function), and disaster protection (typically in the form of replication of data to a remote location). In recent years, these features have been introduced in NAS systems, but they remain in the minority of deployments.

NAS systems are generally most suitable for storing large amounts of file data with modest individual client access performance requirements, and least suitable in applications such as business transaction processing, for which the lowest possible latency (end-to-end I/O request execution time) is the defining performance criterion. Historically, they have been perceived as simple to administer relative to systems that implement other shared data file system architectural models.

Model 2: The file area network (FAN)

The popularity of NAS systems, and the consequent growth of "data islands" (Figure Intro-2) has led to another shared data file system model that is sometimes called the *file area network* (FAN). File area networks bring multiple NAS systems together into a single logical name space that is presented to clients. Figure Intro-4 illustrates the FAN model.

Figure Intro-4 File area network model



In a FAN file system, multiple NAS systems connect to a file system aggregator, usually a highly specialized network router with large amounts of internal bandwidth, network connectivity, and cache memory. As its name implies, the file system aggregator combines the file systems presented to it by two or more NAS into a single name space hierarchy, which it in turn presents to clients.

A FAN file system appears to clients as a single name space. The aggregator keeps track of the locations of sub-trees within the aggregated name space and relays each client request to the NAS system that holds the referenced data. Table Intro-2 summarizes the distinguishing properties of the file area network architectural model.

Table Intro-2 Properties of the file area network model

Tasks performed by application server	Location of file data access software	Location of metadata management software
Translate I/O requests to NAS protocol (typically NFS, CIFS, http, DAV, or similar)	NAS aggregator (client request distribution) Subordinate file servers (data access)	NAS aggregator (data location mapping) Subordinate file servers (metadata access)

Advantages and limitations of the FAN architectural model

FAN file systems have two important advantages over other shared data file system technologies:

- **Name space consolidation.** They combine isolated “islands” of data stored on NAS systems into larger, more flexibly accessed and more easily managed collections. For data centers with dozens of NAS systems to administer, this is particularly important
- **Advanced functions.** They enable advanced functions, principally those based on copying data from one NAS system to another under the control of the file system aggregator. This facility has several applications; three important ones are NAS system-based backup, load balancing across NAS servers, and migration of data from older NAS servers to their replacements. Copying is generally transparent to client computers, so FAN aggregators can generally migrate data while applications are accessing it

The limitations of the FAN architectural model are similar to those of the NAS model that underlies it, long I/O paths (even longer than with direct NAS access), protocol translation overhead, and the bottleneck represented by the file system aggregator itself. Thus, FAN technology is most suited for “tier 2" applications, for which simple administration of large amounts of data stored on

multiple NAS systems has a higher priority than the low-latency access by client computers that is characteristic of business transaction processing.

Variation on the file area network theme: distributed file systems

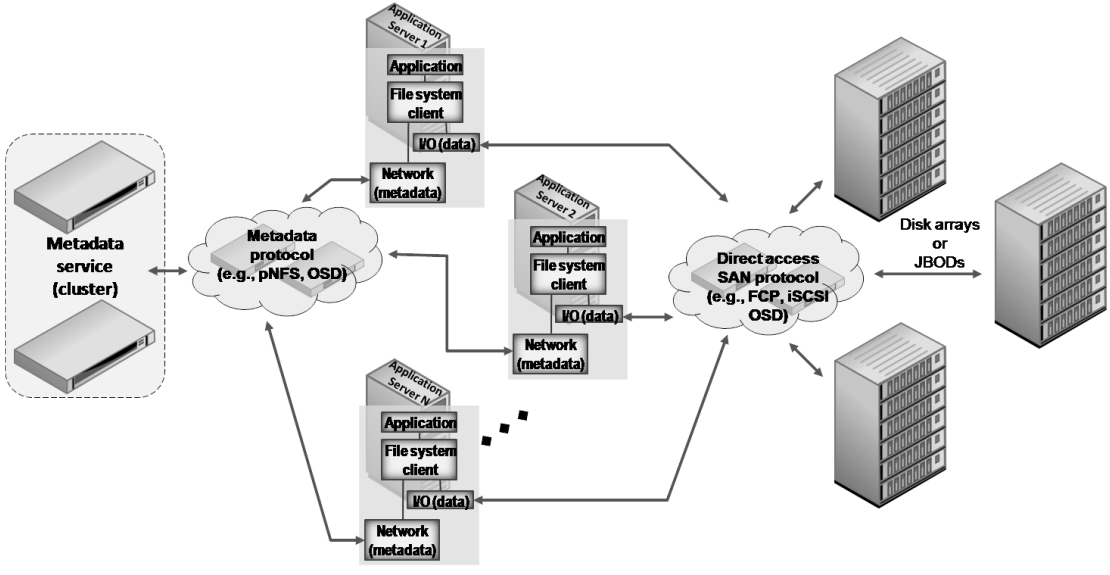
Microsoft Corporation's Distributed File System (DFS) implements a model similar but not identical to the FAN model illustrated in Figure Intro-4. In addition to the files it hosts, a DFS file server stores referrals-links to directory trees hosted by other servers. When a client requests access to a file or directory represented by a referral, the DFS server responds with the hosting server's network name. The client establishes a connection with the server that hosts its data, and communicates directly with it to manipulate files. The DFS referral architecture is sometimes referred to as *out-of-band* file area networking, because client access to referred files is not in the communication path between the client and the server containing the referral.

DFS minimizes the bottleneck inherent in the in-band FAN model illustrated in Figure Intro-4, because data transfer is distributed between clients and the servers that hold their data, rather than being funneled through a single point. Similarly to the in-band FAN model, it enables advanced features based on copying data transparently to clients. It poses challenges, however, synchronizing the copies, and in keeping referrals current as files are created and deleted, and as directory structures change.

Model 3: The SAN (direct data access) file system

The storage area network (SAN) file system is a more recent shared data file system architectural model. SAN file systems can also be termed *parallel*, or *direct data access*, because they enable client computers (usually application servers; SAN file systems are rarely used with single-user desktop computers) to access file data directly from block storage devices, without funneling it through an intermediate stage, as is the case with NAS systems. Figure Intro-5 illustrates a parallel data access shared data file system.

Figure Intro-5 Direct access model



In the model illustrated in Figure Intro-5, client computers running applications communicate with a cluster of metadata servers to authenticate themselves and gain authorization to access file systems. Once authenticated, clients access files by requesting from the metadata service a map of the files' data locations. They use this map to form read and write requests which they communicate directly to storage devices, as the right side of the figure suggests. Table Intro-3 summarizes the properties of the direct data access model.

Table Intro-3 Properties of the direct data access file system model

Tasks performed by application server	Location of file data access software	Location of metadata management software
Request metadata from metadata server Access to file data is direct from data servers	Application server	Metadata server

Advantages and limitations of the SAN file system architectural model

The name “SAN file system” derives from the storage network that connects client computers directly to storage devices (typically a Fibre Channel-based storage network, although iSCSI is also used). The SAN file system architectural model has three important advantages:

- **Bottleneck elimination.** It eliminates the bottleneck represented by the NAS head) (Figure 3)
- **Independent scaling.** It enables storage capacity, metadata processing power, and I/O bandwidth to scale independently as required by applications
- **Low protocol overhead.** It eliminates the double protocol conversion overhead, and for data transfer, increases the potential for zero-copy I/O, at least with Fibre Channel

All of these advantages tend to promote scaling, particularly of I/O performance. In fact, the SAN file system architectural model has been most successful in the high-performance computing sector-simulation, experimental data reduction, and similar applications.

With a SAN file system, once application servers have block storage devices’ network addresses, they can access data on the devices directly. This is a strength of the direct data access model, because it shortens the path between application and data. But it is also the weakness, because block-oriented storage network protocols are typically designed to respond to any command from an authenticated initiator. (This property is sometimes colorfully referred to as *promiscuity*.)

While the danger of a “rogue” application server in a controlled data center environment is minimal, there is no protection against a software error causing data, or worse yet metadata, to be corrupted. Viewed from another perspective, for a SAN file system to function correctly, all metadata servers *and all client application servers* have to function perfectly all the time. Arguably, this risk has inhibited broader commercial adoption of SAN file systems.

Variation on the direct data access theme: object-based file systems

Object-based storage devices (OSDs) are a relatively new arrival on the file storage scene. Based loosely on the file system paradigm, OSDs and the file systems that utilize them eliminate much of the security risk inherent in the SAN file system architectural model.

In essence, OSDs are persistent storage devices that manage their block storage internally and present “objects” that resemble files to clients. Clients’ read and write requests specify block ranges within objects; they do not have access to an OSD’s entire range of block storage.

The OSD-based file system model resembles the metadata server-data server model illustrated in Figure Intro-5. It differs in that file data locations are expressed in terms of object identifiers and block ranges within objects rather than in terms of ranges of raw disk blocks.

Standards-based OSD access protocols include security mechanisms that allow metadata servers to limit client access to specific objects, and to revoke it if necessary, for example if a client times out or unmounts a file system. Thus, clients can only access objects that a file system's metadata servers have authorized them to access. Malfunctioning clients have little or no capacity to damage file system data integrity. Moreover, metadata servers typically detect failing or misbehaving clients immediately and block them from accessing the file system entirely.

Advantages and limitations of OSD-based file systems

While they are technically attractive, two factors limit the adoption of OSD-based direct data access file systems:

- **Limited availability.** OSD technology is relatively new, and no OSD devices are available on the market. Consequently, file system developers have little motivation to create file systems for OSD devices. Generally, the object-based file systems that have been developed are part of complete storage systems that include both software-based OSDs and metadata servers
- **Inherent access latency.** Because opening a file requires access to both a metadata server and an OSD, OSD-based file systems have inherently higher "time to first byte" than block storage-based ones. Thus, they are not particularly well-suited to applications such as desktop file serving in which files are opened and closed frequently

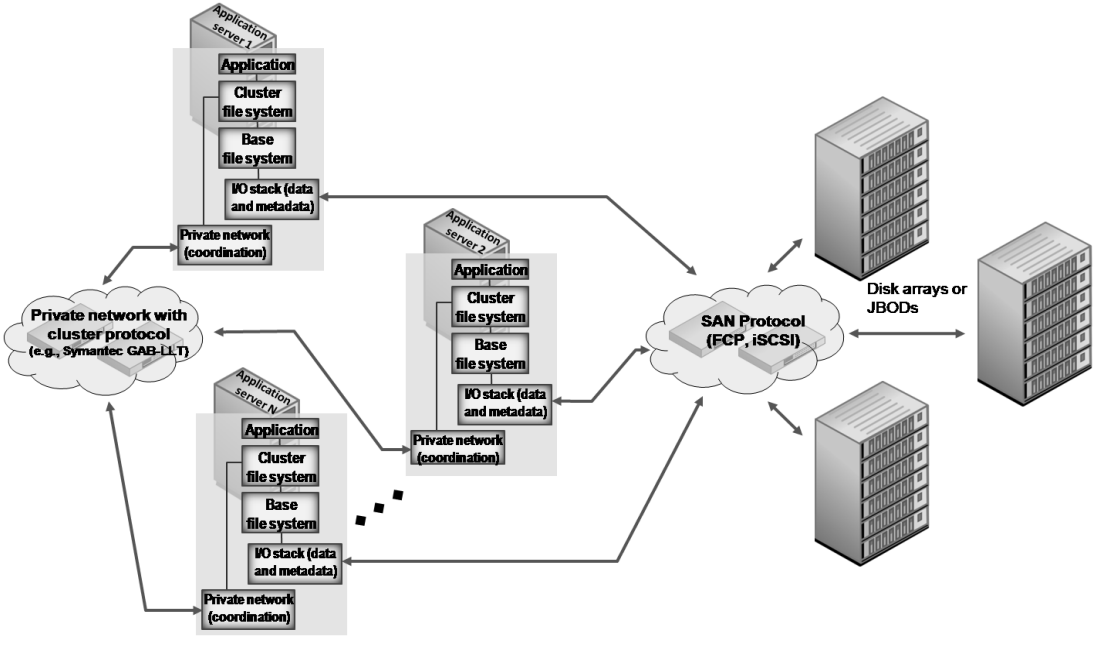
As mentioned earlier, direct data access file systems have gained popularity in high-performance computing, where the scale is large (petabytes), the computing environments are generally trusted, and the magnitude of the data storage and access problems is such that they simply cannot be solved any other way.

Direct data access file system architectural models are evolving rapidly, at least by data access protocol standards. The parallel network file system (pNFS) is a proposed standard for direct client access to file data stored either on block storage devices, OSDs, or NAS systems. Like the OSD technology after which it is patterned, pNFS is a functionally rich model that is finding early success in the high-performance computing sector. As the protocol itself matures, and more storage devices and systems support it, its acceptance in the commercial sector can be expected to broaden.

Model 4: The cluster file system

Figure 6 illustrates the *cluster file system* architectural model for shared access to files. Cluster file systems provide low-latency access and a near linear capacity and performance scaling for a moderate-size (typically 32-64) cluster of nodes (application servers) accessing one or more common sets of files in different file systems.

Figure Intro-6 Cluster file system model



With the cluster file system model, each node runs a complete instance of file system software that is functionally equivalent to the single-server local file system illustrated in Figure Intro-1 in addition to its applications. Each cluster file system instance is aware of the others, and they all cooperate by continuously exchanging state and resource control information over a private network to provide coordinated file system access to their respective applications.

In a cluster file system, all file system instances operate on the same logical images of data. Multiple physical copies of data may exist, for example where disk-level mirroring is used to enhance performance and resiliency, but file system instances perceive a single data image). Table Intro-4 lists the defining characteristics of the cluster file system model.

Table Intro-4 Properties of the cluster file system model

Tasks performed by application server	Location of file data access software	Location of metadata management software
Full file system functionality	Application server (cluster node)	Application server (cluster node)

In a cluster file system, all file system instances manipulate file system metadata directly, after first coordinating with their peer instances in other nodes via network messages to "lock" access to the metadata they are manipulating (the icons labeled "Cluster file system" in Figure Intro-6). Likewise, each file system instance accesses data directly, again, after locking access to it to prevent inadvertent corruption by peer instances running elsewhere in the cluster.

Advantages and limitations of the cluster file system architectural model

The advantages of the cluster file system model are:

- **Low latency.** Direct application server access to data with minimal protocol translation minimizes I/O request latency
- **Resource scaling.** Metadata processing power, cache memory available to file systems, and storage access bandwidth all increase as nodes are added to a cluster. Storage capacity increases independently as disk arrays are added to the storage network
- **Cache coherency.** Each node's updates to cached data are instantly available to all nodes in a cluster so that all have an up-to-date picture of data at all times
- **Load balancing.** When integrated with application clustering, cluster file systems make it possible to redistribute application and file system workload when a cluster node fails or when workload requirements change
- **Rapid recovery.** Because cluster file systems are mounted on all cluster nodes, restart of failed applications on alternate cluster nodes (called failover) tends to be faster than with other approaches, because file systems need not perform full checking and restart. Typically, only replay of the failed node's file system log by a surviving node is required.

Cluster file systems offer low latency data transfer, and so are suitable for business transaction processing as well as other workloads. With short data paths and no protocol conversion during data transfer, cluster file systems are also suitable for high-bandwidth streaming workloads.

The number of nodes in a typical cluster is relatively modest–common upper limits are between 32 and 64. Cluster file systems are therefore not well-suited for personal computer data sharing, where there might be hundreds, or even thousands, of client computers, many of which are mobile. The NAS and FAN architectural models are usually preferable for large numbers of personal computer clients. Chapter 3 describes a novel cluster file system capability, a *clustered NFS server* (CNFS) that enables a cluster of up to 32 nodes running Symantec's Veritas Cluster File System to be deployed as a scalable, resilient NFS server suitable for providing file access services to dozens of application servers or thousands of personal computer clients.

Part I

Understanding and using CFS

- What makes CFS unique
- Using CFS: application scenarios
- Using CFS: scalable NFS file serving

Chapter 1

What makes CFS unique

This chapter includes the following topics:

CFS foundation and platforms

The cluster file system component of Symantec's Veritas Storage Foundation Cluster File System package (SFCFS), known informally as CFS, evolved from Symantec's proven Storage Foundation File System (commonly called VxFS), originally developed by VERITAS Software Corporation. CFS is a 64-bit fully POSIX-compliant cluster file system available for Sun Microsystems' Solaris (SPARC and 64-bit Intel), Hewlett-Packard's HP-UX, IBM's AIX, and Linux distributions offered by RedHat, SuSe, IBM, and Oracle.

CFS can host multiple file systems[3], each encompassing up to 256 terabytes of storage and containing up to a billion files. It includes several advanced capabilities that make it the file system of choice in complex production data center environments. The "top 10 list" of CFS advanced capabilities are described later in this chapter.

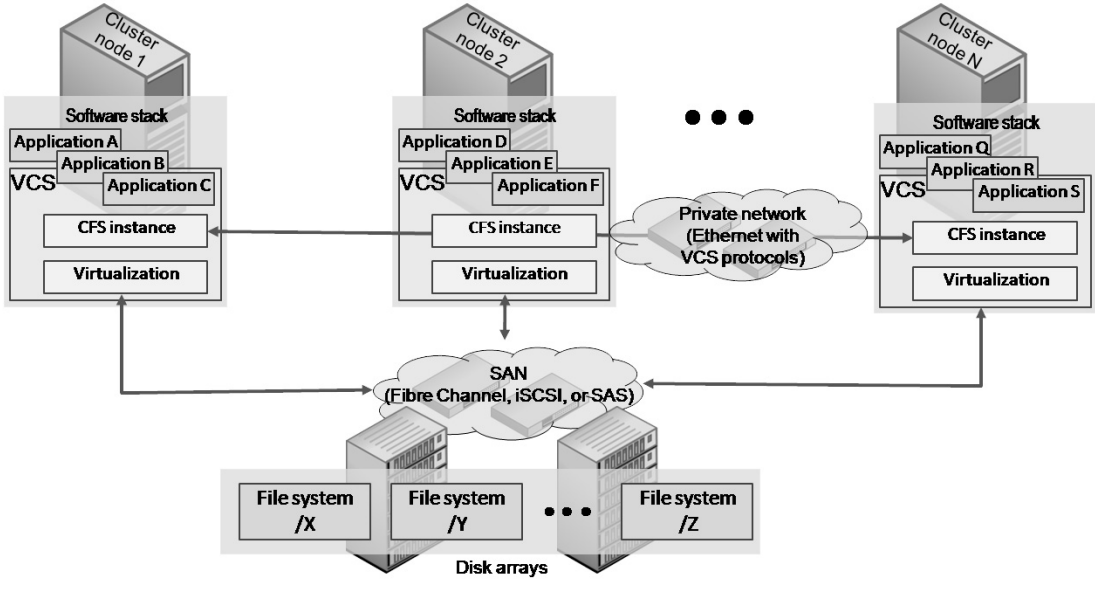
CFS implements the cluster file system architectural model described in "Model 4: The cluster file system" on page 21 in UNIX and Linux-based VERITAS Cluster Server (VCS) computer clusters, as well as Hewlett-Packard MC-Service Guard

3. The term *file system* is commonly used to refer to both (a) the body of software that manages one of more block storage spaces and presents the file abstraction to clients, and (b) a block storage space managed by such software and the files it contains. The intended meaning is usually clear from the context.

clusters on the HP-UX operating system. CFS instances in a VCS cluster of as many as 32 *nodes*[4] (application servers) cooperate to provide simultaneous shared access to file systems for applications running on some or all of the cluster's nodes. Figure 1-1 illustrates the software topology of a cluster in which CFS provides shared file system access.

Each CFS instance provides file access services to applications running on its node. For example, in Figure 1-1, the CFS instance on Cluster node 1 can mount file systems **/X**, **/Y**, and **/Z**, and make them available to Applications **A**, **B**, and **C**. The instance on Cluster node 2 can mount the same three file systems and make them available to Applications **D**, **E**, and **F**, and so forth. CFS instances coordinate file access across the entire cluster so that all applications have the same view of the file system state and file contents at all times, and so that potentially conflicting updates do not interfere with each other or with other file accesses.

Figure 1-1 CFS topology



In the scenario illustrated in Figure 1-1, the applications might be completely independent of each other, or they might themselves be structured as *high availability cluster services*. For example, Application **A** might be configured as a cluster service that would "fail over" to (automatically restart on) Cluster node

4. Symantec supports CFS in clusters of up to 32 nodes at the time of publication. Readers should consult the most recent product documentation for up-to-date product parameters.

2 if Cluster node 1 were to fail. The surviving nodes discover the failure, and *reconfigure* themselves into a new cluster consisting of nodes 2-N. After the reconfiguration, Application **A** automatically restarts on node 2. Since the CFS instance on node 2 serves the same file systems to applications running there, it seamlessly provides file system access to the restarted Application **A**.

Alternatively, applications may be structured as *parallel cluster services*. Instances of parallel services run concurrently on multiple cluster nodes, sharing access to CFS files and serving the same or different clients. If a node fails, its clients are shifted to an instance on a surviving node, and service continues uninterrupted.

CFS applications

With the ability to support up to 256 terabytes of storage and up to a billion files per name space, multi-volume file systems, and individual sparse files as large as 8 exabytes[5], CFS file systems can be sized to meet demanding enterprise requirements.

Supporting large file systems is much more than data structures. CFS is designed for reliable high performance file data access across a broad spectrum of enterprise applications, including:

- **Media.** CFS efficiently maps multi-gigabyte media files for fast access
- **Science and engineering.** CFS is an ideal repository for large data sets collected during experiments or generated by modeling applications, and reread piecemeal for analysis
- **Commercial.** CFS is versatile enough to support databases accessed randomly by business transaction processing applications as well as those accessed sequentially by decision support systems
- **Unstructured.** CFS file systems make high-performance platforms for file serving where users are constantly creating, deleting, opening, and closing large numbers of files of varying size and composition. The Clustered NFS (CNFS) feature described in Chapter 3 extends scalable file serving to thousands of users through the Network File System (NFS) protocol

CFS uses an efficient *extent-based* scheme to map blocks of file data to file system block locations. CFS data mapping is particularly concise for large contiguous files: in principle, it can describe the largest file that CFS supports with a single descriptor. Concise data block mapping is an important factor in I/O performance. Not only does it minimize the amount of metadata required to map a file's data, it simplifies application read and write request execution

5. Because it supports "sparse" files (files for which no storage is allocated to block ranges until data is written to them), CFS can create file address spaces that are larger than the amount of block storage allotted to a file system.

because it minimizes the number of file block ranges that must be read or written with separate disk I/O commands.

CFS prerequisites

In order for CFS instances to share access to file systems, the storage devices that hold the file systems must be directly accessible by all cluster nodes; that is, storage devices and the cluster nodes must all be interconnected by a storage network as Figure 1-1 suggests. CFS stores metadata and data on virtual storage devices called *volumes* that are managed by the Symantec Cluster Volume Manager (CVM) component of the Storage Foundation. CVM configures volumes by combining disks and disk array logical units (LUNs) connected to the cluster's nodes by Fibre Channel, iSCSI, or Serial Attached SCSI (SAS) storage networks.

In addition to their connections to shared storage devices, the nodes in a VCS cluster must be interconnected directly to each other via an Ethernet-based *private network*, as illustrated in Figure 1-1. VCS uses the private network to transmit *heartbeat* messages among nodes and to coordinate cluster reconfigurations and application failovers. CFS uses the VCS private network to coordinate access to shared file system resources such as file and free space metadata.

CFS packages

Four Symantec products include CFS technology:

- **Storage Foundation CFS (SFCFS).** In addition to CFS and CVM, the SFCFS package includes the VCS components and configuration tools that enable a group of computers to form a cluster, create volumes, and create, mount, and access shared file systems. It does *not* include the VCS facilities required to structure arbitrary applications for automatic failover
- **Storage Foundation CFS for high availability (SFCFS-HA).** The SFCFS-HA package includes CFS, CVM, and full-function VCS. With SFCFS-HA, most applications can be configured as highly available VCS service groups that store their data in CFS shared file systems mounted on CVM volumes
- **Storage Foundation for Oracle Real Application Cluster (SFRAC and SFCFSRAC).** The SFRAC package, and the companion SFCFSRAC (for Linux platforms) include CFS, CVM, and VCS, along with additional software components that facilitate high availability for Oracle's *Real Application Cluster* (RAC) clustered database management system software
- **Storage Foundation for Sybase Cluster Edition (SFSYBCE).** The SFSCE package includes CFS, CVM, and VCS, along with additional software components that facilitate high availability for *Sybase Cluster Edition* clustered database management system software

Any of these can be installed on a supported UNIX or Linux cluster. In addition,

CFS is the file system component of file storage systems that are based on Symantec's *FileStore* NFS and CIFS file server technology.

What makes CFS unique

The remainder of this chapter describes the CFS "top ten list"—ten features that differentiate CFS from other cluster file systems for UNIX platforms and from other approaches to file data sharing. The features are listed alphabetically because it is impossible to assign relative importance to them:

- **Feature 1: Cluster and data disk fencing.** Any cluster runs the risk of partitioning into two groups of nodes that cannot communicate with each other. To avoid incorrect operation and data corruption, there must be a foolproof algorithm for dealing with partitioning that always results in one partition continuing to operate as a reduced cluster and the other shutting down. VCS uses *coordinator disks* to resolve cluster partitions that result from failures of the private network. (Alternatively, a *coordinator server* can be configured in place of one or more of the disks.) In addition, if CFS file systems reside on volumes configured from PGR-capable disks, CVM uses *data disk fencing* to protect against data corruption when a cluster partitions (see "Feature 1: Cluster and data disk fencing" on page 33)

- **Feature 2: Database management system I/O accelerators.** CFS includes three different database acceleration options that make it possible for database management systems and other I/O intensive applications to get the administrative convenience of using files as data containers without incurring the performance penalty typical of this approach. (see "Feature 2: Database management system I/O accelerators" on page 36 and Chapter 11)

- **Feature 3: The File Change Log (FCL).** A CFS file system can be configured to maintain a circular *File Change Log* (FCL) in which it records descriptions of all changes to files in the file system. Incremental backup, auditing, and similar applications can use APIs supplied with CFS to determine which files in a file system were changed during a given period. (see "Feature 3: The File Change Log (FCL)" on page 37)

- **Feature 4: The file system history log.** CFS permanently logs all maintenance performed on a file system in the file system itself. The file system history log gives support engineers instant access to reliable, up-to-date information about the state of a file system for faster problem diagnosis and resolution. (see "Feature 4: The file system history log" on page 39)

- **Feature 5: Flexible snapshots and clones.** Products that include CFS support both snapshots of sets of CVM volumes and snapshots of file systems (called *Storage Checkpoints*). Administrators can choose between full-size volume snapshots that can be taken off-host for separate processing and space-optimized snapshots of either volumes or file systems that occupy space in

proportion to amount of changed data in a data set, rather than the data set's size. All Storage Foundation snapshot technologies can be configured for use either as read-only point-in-time images of data, or as writable clones of their parent data sets.
(see "Feature 5: Flexible snapshots and clones" on page 40)

- **Feature 6: Named data streams.** Named data streams make it possible for applications to attach virtually unlimited custom metadata to files. Structured as hidden files, named data streams can contain anything from a single byte to a video clip.
(see "Feature 6: Named data streams" on page 42)
- **Feature 7: Portable Data Containers (PDC).** Storage Foundation Portable Data Container (PDC) technology makes *cross-platform data sharing* possible. CFS file systems produced by one type of VCS cluster platform (for example, AIX) can be converted for use on a system or cluster of a different type (for example Linux), even if the two platforms use different memory addressing.
(see "Feature 7: Portable Data Containers (PDC)" on page 43)
- **Feature 8: User and group quotas.** CFS enforces both hard (non-exceedable) and soft (exceedable for a limited time) quotas that limit the file system storage space that individual users and groups of users are permitted to consume.
(see "Feature 8: User and group quotas" on page 47)
- **Feature 9: Sparse files.** CFS files are inherently sparse. By default, the file system only allocates storage for file blocks to which applications actually write data. No storage is allocated for file blocks that have never been written. Sparse files simplify indexing when index spaces are large, without exacting a toll in overprovisioned storage.
(see "Feature 9: Sparse files" on page 48)
- **Feature 10: Storage tiers.** CFS helps minimize blended storage cost in large file systems that contain combinations of active and inactive or critical and non-critical data by automatically placing files on the "right" type of storage throughout their lifetimes.
(see "Feature 10: Storage tiers" on page 50)

In addition to these ten, all of which are shared with the single-instance VxFS file system, many other CFS features, including its "friendliness" to thin provisioned underlying storage devices, support for reclamation of unused storage capacity, and SmartMove technology for data migration, make it the file system of choice for critical enterprise applications in complex, rapidly changing data center environments.

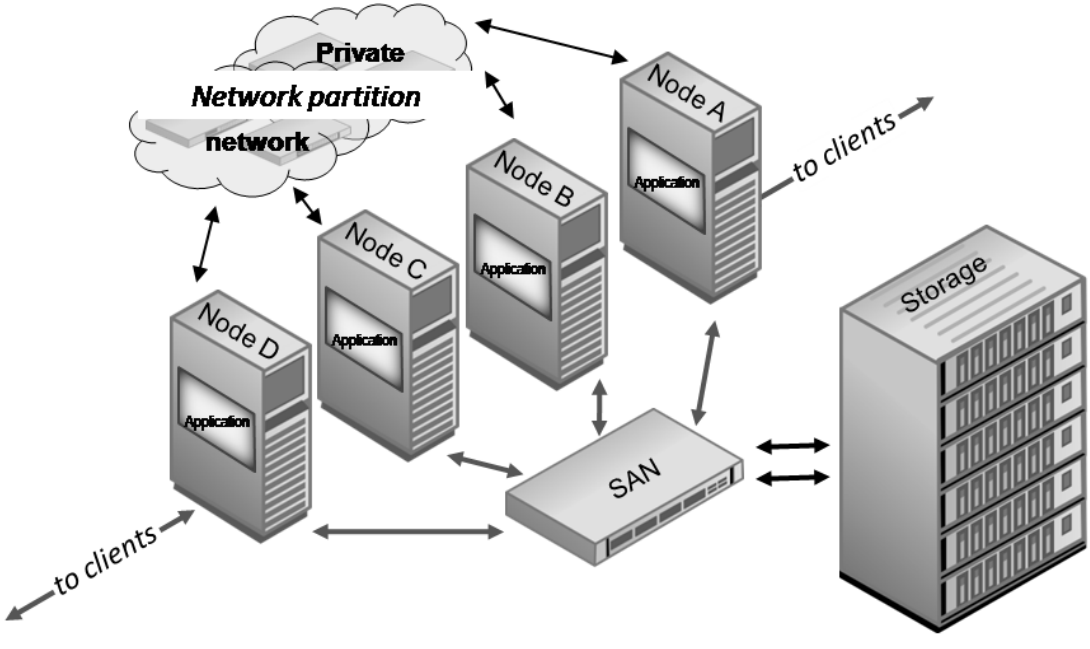
The top 10 in depth

The sections that follow describe the “top 10” list of CFS differentiating features in depth.

Feature 1: Cluster and data disk fencing

Like any shared data cluster whose nodes and storage are interconnected by separate networks, a CFS cluster must avoid data corruption due to *partitioning* (sometimes informally called a *split brain* condition). If a cluster’s private network fails in such a way that two or more disjoint groups of nodes cannot communicate with each other, one group of nodes can continue to act as the cluster, but the remaining groups must shut down to avert data corruption. Figure 1-2 illustrates partitioning in a four-node cluster.[6]

Figure 1-2 Cluster partitioning



In Figure 1-2, the private network has become partitioned so that Nodes A and B can communicate with each other, as can Nodes C and D. But the two halves of

6. Partitioning is usually described prominently in cluster-related literature, but is in fact a fairly rare condition, particularly in VCS clusters, because VCS requires redundant private networks, and in addition, supports private networks that are doubly redundant, and therefore capable of sustaining two link failures without partitioning.

the cluster cannot intercommunicate on the private network. Both halves of the cluster *can* communicate with storage devices, however, creating the potential for corrupting data and file system structures.

Partitioning of a VCS cluster's private network is difficult to diagnose because the nodes within each partition (Nodes A and B and Nodes C and D respectively in Figure 1-2) cannot distinguish between:

- **Node failure.** Failure of the nodes in other partitions. In this case, the failed nodes are shut down by definition; the surviving nodes should become the cluster and continue to provide services to clients
- **Private network failure.** Failure of communication links between the partitions. In this case, there is no automatic "right" answer to which partition should continue to function as the cluster. Both partitions are functioning properly, but since they cannot intercommunicate, they cannot coordinate access to shared storage devices

Coordinator disks and coordination point servers

VCS resolves network partitions by using *coordinator disks* or *coordination point servers* as extra communication channels that allow partitioned cluster nodes to detect each other. For resiliency against disk failure, VCS fencing requires three (or any larger odd number of) dedicated coordinators. Coordinator disks must have the following properties:

- **Persistent Group Reservation support.** Coordinator disks must support SCSI-3 Persistent Group Reservations (PGR)
- **Cluster-wide accessibility.** Coordinator disks must be accessible by all cluster nodes, ideally each one via a separate I/O path

Coordinator disks do not store data, so small-(10 megabyte or greater) LUNs are the most suitable candidates. The VCS fencing algorithm requires that a cluster have three or some larger odd number of coordinators, and that coordinators always be accessed in the same order.

When a cluster node determines (by the absence of heartbeat messages) that it cannot communicate with one or more other nodes, it first blocks I/O to shared file systems to protect against data corruption. It then requests exclusive access to the first coordinator (using the PGR protocol if the coordinator is a disk). All nodes in a partitioned cluster do this at approximately the same time.

The SCSI-3 Persistent Group Reservation protocol and the VCS protocol for coordination point servers only permit one node to successfully reserve the first coordinator; all other nodes' requests fail because the coordinator is already reserved. A node's subsequent behavior depends upon its success in reserving the first coordinator:

- **Successful node.** The successful node immediately requests reservations for the other two coordinators

- **Other nodes.** Nodes that failed to reserve the first coordinator voluntarily wait for a short period before requesting reservations on the other coordinators

Reconfiguring a partitioned cluster

Once a node succeeds in reserving more than half of the coordinators, it and the nodes with which it can communicate on the private network perform a cluster reconfiguration resulting in a cluster that does not include nodes with which they cannot communicate. The nodes of the reconfigured cluster complete failover by performing the following actions:

- **Revoke partitioned nodes' access to data disks.** The new cluster's CVM master commands all PGR-capable data disks in CVM shared disk groups to revoke the registrations of partitioned nodes so that commands from them are no longer honored. This is called *data disk fencing*. Data disk fencing greatly diminishes the potential for a cluster partition to result in data corruption, and is therefore highly recommended
- **Fail over virtual IP addresses.** Any IP addresses configured as high availability service groups fail over to nodes in the reconfigured cluster
- **Fail over services from partitioned nodes.** VCS restarts high-availability services from nodes that were partitioned out of the cluster on their designated failover nodes
- **Resume service to clients.** Restarted applications on the reconfigured cluster resume perform crash recovery and resume service to clients

VCS forces partitioned nodes that fail to reserve a majority of the coordinators to shut down abruptly ("panic"). They remain inoperative until the private network has been repaired and an administrator has added them back into the cluster.

While cluster fencing is internally complex, from an administrative standpoint it is a simple "set-and-forget" facility. A VCS administrator designates three or more coordinator disks or coordination point servers and activates fencing. Thereafter, the only administrative requirements are non-routine maintenance such as disabling fencing, checking coordinator status, and replacing failed coordinator disks.

Data disk fencing

For PGR-capable shared data disks, the CVM Master sets their PGR keys to reserve them for exclusive use by cluster members ("fences" them). When a cluster reconfigures, CVM removes the keys for the departed members to avoid the potential for data corruption in case removed nodes behave improperly (e.g., by attempting to flush data). Without the PGR capability, there is a remote but real possibility that badly-behaved applications in nodes removed from the

cluster could corrupt data by continuing to access disks after reconfiguration.

Feature 2: Database management system I/O accelerators

Database administrators (DBAs) often use files as "storage containers" for database metadata and data because they simplify common administrative tasks, such as moving, copying, and backing up selected subsets of database records, indexes, and logs. Moreover, when database storage requirements change, files are significantly easier to expand or shrink than disks or virtual volumes.

But the data caching and I/O serialization that file systems use to isolate users from each other can hamper database I/O performance. Since database management systems are the sole users of their files, and since they coordinate their own file accesses to avoid conflicting updates, these file system protection mechanisms are by and large unnecessary. CFS includes three mechanisms that enable database management systems to bypass most of the unneeded file system protections against concurrent access and the overheads they incur:

- **Oracle Disk Manager (ODM).** Oracle Corporation publishes a specification for an *Oracle Disk Manager* (ODM) API that its database management system products use to optimize I/O operations. Perhaps the most important function of ODM is asynchronous file I/O between Oracle's own buffers and the disks on which the files resides. CFS includes an ODM library that uses CFS and CVM capabilities to implement the functionality expressed in the ODM APIs

  Administrative hint 1

  Systems that use CFS Quick I/O files for Oracle database storage should be upgraded to the Oracle Disk Manager library, to improve integration with Oracle current and future releases.

- **Quick I/O.** The Quick I/O for databases feature is the functionally similar precursor of CFS' ODM library implementation. Still supported by CFS on enterprise UNIX platforms, Quick I/O provides advantages similar to ODM's for any database management system (or other application) that synchronizes its own I/O requests and coordinates buffer usage internally

- **Concurrent I/O.** For database management systems and other applications that do not include their own APIs for storage access, CFS includes the *Concurrent I/O* (CIO) feature that makes it possible for any application to perform asynchronous file I/O directly between its own buffers and the disks on which the files reside. Administrators can specify concurrent I/O as a mount option, applying it to all files in a file system. Alternatively, application developers can declare **cio** as a cache advisory for specific files

Using these CFS mechanisms, database administrators can enjoy the convenience of file-based storage administration without paying a penalty in diminished database I/O performance compared to that of "raw" block storage

devices. The CFS database I/O acceleration mechanisms enhance database management system I/O performance in three ways:

- **Asynchronous I/O.** CFS database I/O accelerators make it possible for database management system execution threads to issue I/O requests and continue executing without waiting for I/O to complete. When a thread requires the result of its I/O request, or when it has exhausted its work queue, it waits for the I/O to complete
- **Direct I/O.** CFS database I/O accelerators schedule data transfers directly between database manager buffers and disk storage. They do not copy data between database manager buffers and operating system page cache as CFS would for normal application I/O requests
- **Write lock avoidance.** CFS database I/O accelerators bypass the operating system's normal file write locking mechanisms. This increases parallel execution by allowing concurrent requests to be issued to CVM and thence to the hardware I/O driver level

Chapter 11 discusses the ODM, Quick I/O, and CIO database I/O acceleration mechanisms in detail.

Feature 3: The File Change Log (FCL)

An administrator can configure CFS to maintain a *File Change Log* (FCL) for each mounted file system in which it records information about all changes made to the file system's files. FCL records identify:

- **Files.** Files or directories affected by the change
- **Operations.** Creation, expansion, truncation, deletion, renaming, and so forth
- **Actors.** Process IDs and user and group IDs under which changes were requested
- **Times.** Times at which operations were performed

In addition, FCLs can periodically record:

- **Filestats.** I/O activity (called *filestats*) against individual files
- **All accesses.** All file opens, including those for read-only access. These can be useful for auditing purposes.

An FCL records file change events, but does not record changed data.

FCLs are *circular*. If an FCL fills to its maximum allowable size without being cleared or saved by an administrator, CFS overwrites the oldest information in it. When this occurs, CFS writes a record into the FCL indicating that records have been deleted. Administrators can enable and disable FCL recording at any time, and can adjust an FCL's size to increase or decrease the number of records that can be retained. Additionally, administrators can set the minimum interval between successive open and write records to limit the space that the FCL consumes during periods when files are being updated frequently, while still capturing the fact that files were updated. Administrators can also copy FCL contents to regular files to preserve a permanent record of file system changes, and can clear FCL contents at any time (for example, after copying) to "start fresh" with file system change recording.

Administrative hint 2

An administrator uses the **fcladm** console command to manage the FCL, including starting and stopping recording, saving FCL contents, and adjusting parameters that control FCL operation.

Each CFS instance maintains an FCL for each mounted file system in which it records changes that it makes. Periodically, the primary CFS instance merges all instances into a master FCL in which all records, even those from different nodes that refer to the same object are recorded in proper sequence.

CFS includes APIs that any application can use to retrieve FCL records. Applications that use these APIs retrieve a single cluster-wide stream of file change history records in which all records are in proper sequence, even if they were written by different instances.

Data management applications can use FCL information in several ways, for example:

- **Backup.** Backup programs can read FCL records to determine which files in a file system changed during a given period, thus eliminating the need to examine every file to determine which to include in an incremental backup
- **Replication.** *Episodic* (periodic) replicators can read FCL records to quickly identify files that have changed since the last replication episode, and must therefore be communicated to the replication target during the next episode
- **Search.** Search engines that build persistent indexes of file contents can use the FCL to identify files that have been created, modified, or deleted since their last index update, and thus perform incremental index updates rather than full file system scans
- **Audit.** If a file system's FCL is configured to record every access to every file in a file system, auditors can use it to determine the history of accesses and modifications to a file and the users and applications that made them

- **Workflow.** Workflow applications can read FCL records to identify documents that have been created, modified, or deleted, and use the information to schedule tasks accordingly

CFS itself uses the information in the FCL. The Dynamic Storage Tiering feature (Chapter 10) uses FCL filestats to compute files' *I/O temperatures*, in order to determine whether they should be relocated to alternate storage tiers based on the I/O activity against them relative to overall file system activity.

Feature 4: The file system history log

CFS maintains a log of all maintenance performed on a file system, including file system resizings, volume additions and removals, volume and file system data layout changes, and so forth. Entries in a CFS file system's history log may be made by the file system kernel or by file system utilities. Kernel-initiated history log entries include:

- **Resizing.** While initiated by administrative action, resizing is actually performed by the file system kernel. CFS writes separate history log records whenever a file system itself or its intent log is resized
- **Disk layout upgrade.** When the CFS kernel upgrades a file system's *disk layout* (the pattern used to organize disk blocks into files and metadata) while it is mounted, CFS writes a record in the file system's history log
- **Volume set changes.** Each time a storage volume is allocated to or removed from a file system, CFS writes a history log record. The history log contains a complete record of changes to a file system's storage complement, no matter how many times volumes have been added to or removed from it
- **Allocation policy change.** A multi-volume file system may have a DST (Chapter 10) policy assigned. When an administrator alters the policy in a way that affects initial file allocations (not relocations), CFS writes a history log record
- **Metadata I/O error.** Any unrecoverable I/O error when reading or writing file system metadata has the potential for corrupting a file system's data or structural information. CFS writes a history whenever file system metadata cannot be retrieved

History log entries written by file system utilities include:

- **Cross-platform Data Sharing (CDS) conversion.** The **fscdsconv** utility writes a history log record when it converts a file system for mounting on a different supported platform
- **Offline upgrade.** When a file system's disk layout is upgraded offline (while the file system is not mounted), CFS writes a history log record capturing the upgrade

- **Creation.** When the **mkfs** and **mkfs_vxfs** utilities create a new CFS file system, they create a history log and record the creation date and time and other parameters in it
- **Full file system checking.** In the rare instances when a full file system check (**fsck**) is required (for example, when the file system's superblock becomes unreadable), CFS records the check event in the file system's history log

The CFS file system history log is intended for the use of Symantec and partner support engineers, so no user-accessible utilities for it are included in CFS products. Having a file system history log leads to faster problem diagnosis and resolution by giving support engineers instant access to reliable, up-to-date information about the state of a file system and how it got that way.

Feature 5: Flexible snapshots and clones

Snapshots, images of data sets as they appeared at an instant in time, are one of an administrator's most useful data management tools. A snapshot is a "stable" (unchanging) image of a data set that can processed by auxiliary applications while production applications continue to process the data set itself. Snapshots can be used for:

- **Sourcing backup.** A snapshot can be the source for making a backup copy of data set contents that is consistent as of a single point in time
- **Data recovery.** With periodic snapshots of a data set, files that are inadvertently deleted or become corrupted can be recovered by copying their images from a snapshot taken prior to the corrupting event
- **Analysis.** Snapshots can be used to analyze, or "mine," stable images of a data set while production applications continue to process the data set itself
- **Test and development.** Snapshots of production data sets are realistic data against which to test new software developments and operating procedures

Snapshots can be classified as:

- **Full-size.** Complete copies of their parent data sets. Full backups of static data sets and mirrors separated from mirrored volumes are both full-size snapshots
- **Space-optimized.** Copies of parent data set data that is modified after snapshot creation. For unmodified data, space-optimized snapshots point to the parent data sets' images

Full-size snapshots can be separated from their parent data sets, moved to other systems, and processed completely independently. Space-optimized snapshots can only be processed by systems that have access to their parent data sets, because they rely on their parent data sets for unmodified data. Nevertheless, users and administrators find space-optimized snapshots attractive for most applications for two primary reasons:

- **Creation speed.** Space-optimized snapshots can be created nearly instantly, whereas full-size snapshots cannot be created any faster than their parent data sets can be copied
- **Space occupancy.** Space-optimized snapshots occupy physical storage in proportion to modifications made to their parent data sets; not in proportion to their size. In most cases, space-optimized snapshots occupy a tiny fraction of the space occupied by their parent data sets. For example, a space-optimized snapshot of a 100 gigabyte data set, 1% of whose contents have changed since the snapshot was taken, occupies only about one gigabyte of storage. Because they are so compact, it is usually feasible to maintain dozens of space-optimized snapshots, even of very large data sets

Some snapshot technologies produce writable images called "clones" that can be modified by applications without altering the parent file systems from which they were taken. Full-size snapshots inherently have this property, but some space-optimized snapshots, including those available with CFS, are writable as well. Clones are useful for training, destructive software testing, and performing "what if" analyses.

Administrators can choose between full-size and space-optimized snapshots of sets of CVM volumes, as well as CFS space-optimized file system snapshots called *Storage Checkpoints*.

Administrative hint 3

An administrator uses the fsckptadm console command to create, delete, and otherwise manage a file system's storage checkpoints. The VxVM vxsnap command is used to manage volume-level snapshots.

The CFS Storage Checkpoint facility produces space-optimized read-only file system snapshots as well as writable file system clones. CFS Storage Checkpoints can be mounted and used as though they were file systems, either cluster-wide or by a single node, and used for any of the typical snapshot purposes—backup to tape, data mining, individual file recovery, and (for writable Storage Checkpoints) training and other destructive testing. A special form of Storage Checkpoint, the *Nodata Storage Checkpoint*, can be used to keep track of the numbers of changed file system blocks without preserving their prior contents. Nodata Storage Checkpoints are useful for making block-level incremental backups or replicas of a file system image.

CFS Storage Checkpoints occupy storage capacity in the parent file system's own volumes. Thus, in most cases, creating Storage Checkpoints does not require the allocation of additional volumes or LUNs, and so can be accomplished by the administrator responsible for the file system, without recourse to system or storage hardware administrators. As with file systems, an administrator can assign quotas to Storage Checkpoints to limit the amount of space they are permitted to consume.

Feature 6: Named data streams

Applications often have a need to associate auxiliary data or application-specific metadata with files. Keywords to facilitate searching, information about a file's provenance, projects with which the file is associated, references to related documents, and so forth can all be useful or necessary adjuncts to the data in a file itself.

One way to manage application-specific metadata is to create a database or spreadsheet table with a row for each file and a cell for each type of metadata. This approach can be made to work, but has several drawbacks:

- **Management overhead.** A database or spreadsheet is an additional object to manage. Moreover, there is no automatic, permanent association between a file and its application-specific metadata. For example, when the file is copied, renamed, or deleted, changes to the application-specific metadata must be managed separately
- **Metadata inefficiency.** A rectangular table implicitly associates every type of metadata with every file represented in it, even if most types of metadata are not relevant for most files
- **Restricted metadata types.** Paradoxically, even though it is structurally wasteful, a rectangular table restricts the types of metadata that can be associated with files to what can practically be stored in a cell

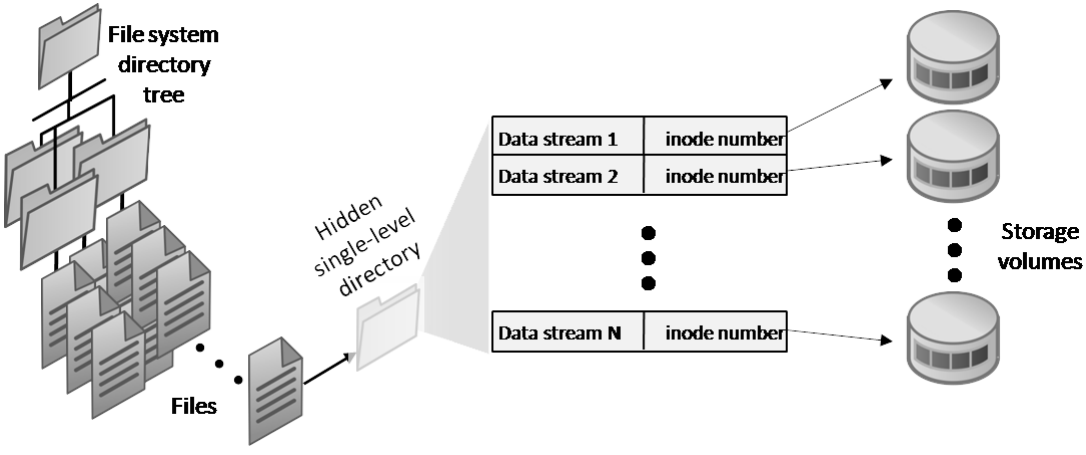
CFS *named data streams* offer a simpler and more robust approach to associating auxiliary data and application-specific metadata with files. A named data stream may be thought of as a file that is permanently associated with a data file. Named data stream-aware applications create, write, read, and delete named data streams using CFS-specific APIs similar to the corresponding POSIX APIs for manipulating data files. Thus, the management of application-specific metadata can be embedded within applications; no external operating or administrative procedures are needed. Named data streams are invisible to applications that do not use the specialized APIs.

From a CFS structural standpoint, named data streams are files that are visible only within the context of the data file to which they are attached. Data stream names are entirely at application discretion. There is no artificial limit to the number of named data streams that can be attached to a file. Thus, for example, dozens of supporting documents can be attached to a single data file. Because CFS treats named data streams as files, they can range from very small to very large. Thus, for example, an application can attach objects such as audio or video clips to a data file.

Administrative hint 4

Administrators can refer utility programmers to the *Veritas™ File System Programmer's Reference Guide*, which describes the APIs used to create and manipulate named data streams.

Figure 1-3 Named data streams



As Figure 1-3 suggests, CFS organizes a data file's named data streams as a single-level directory that is linked to the data file. Each named data stream is a file whose data location is described by an inode, but the stream is visible only through the named data stream APIs. A data file's named data stream directory file contains the list of stream names and their inode locations.

Applications can use the named data stream APIs to associate a data stream with multiple files. CFS stores the stream once, and maintains a link count in its inode, which it increments each time an application associates the stream with an additional file, and decrements when the stream is unlinked.

In most cases, operating system commands cannot manipulate named data streams separately from the files with which they are associated. Sun Microsystems' Solaris versions 9 and later, however, implement special command syntax that enables certain operations on named data streams.

Feature 7: Portable Data Containers (PDC)

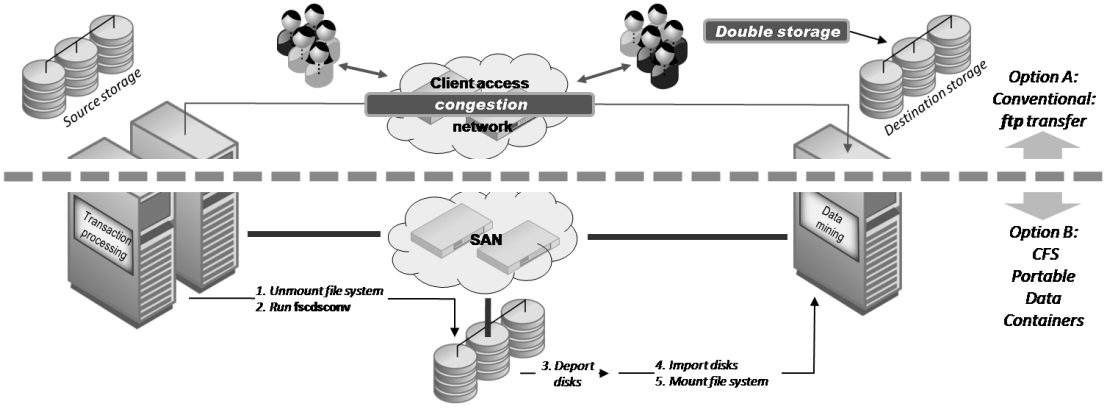
For a variety of reasons, many data centers routinely move large amounts of data between unlike UNIX or Linux platforms. For example, a data center may process its business transactions on a Solaris platform, and periodically mine snapshots of transactional databases on a Linux platform. In situations like this, data centers usually copy data sets from one platform to the other over network links using **ftp** or other file transfer tools. But transferring multiple terabytes of files can be time consuming, and can saturate a network, interfering with, or in extreme cases, even denying service to production applications and users. In addition, they increase storage requirements, since storage for both source and destination data sets must be available simultaneously, at least while the copy is being made and used.

Storage networks that form physical links between storage devices and multiple

systems suggest a potential alternative. In principle, it should be possible to:

- **Disconnect.** Logically disconnect disks from the cluster that produced the data
- **Reconnect.** Logically connect the disks to a *destination* system that will process it
- **Use.** Mount the file system on the disks on the destination system and process the data without having used any incremental storage or consumed network bandwidth, and without having taken time to transfer data

Figure 1-4 Portable Data Containers vs network file copying



This is sometimes called "serial sharing" of data. It works very effectively between application platforms of the same type; indeed, it is one of the principal value propositions of storage networks. Unfortunately, different UNIX and Linux platforms do not format their disk storage devices and file systems in exactly the same way. Moreover, applications that support multiple platforms sometimes use different file data formats for each platform they support.

For many common data types, however, including text, html, pdf files, audio-visual media streams, and others, applications on one platform *would* be able to use data created on a different platform if the underlying disk and file formats were compatible. CFS and CVM *portable data container* (PDC) technology enables cross-platform serial data sharing between unlike platforms. PDCs make it possible to logically move CVM volumes that contain a CFS file system between two unlike platforms, even in cases where the source and destination platforms' "endianness"[7] differ from each other. Figure 1-4 illustrates the use of

7. A computer's *endianness* is the order in which bytes of data in its memory are aggregated into larger structures such as words and longwords. In a big-endian computer, the most significant byte of a word or longword occupies the lowest memory address. In a little-endian computer, the least significant byte of a word or longword occupies the lowest memory address.

PDCs as an alternative to bulk network data transfer between unlike computing platforms and contrasts it with the conventional network data copying technique.

Administrative hint 5

The *Veritas™ Storage Foundation Cross-Platform Data Sharing Administrator's Guide* contains instructions for upgrading CVM disk formats so that volumes can serve as portable data containers for cross-platform data sharing.

The first requirement for moving data between unlike platforms is disk format compatibility. The CVM disk format has evolved over time to accommodate larger and more complex storage configurations. A system administrator can designate SCSI and Fibre Channel disks formatted with CVM format version 110 or any newer version as *CDS disks*. Administrators can logically move CDS disks between unlike computing platforms that are connected to a common storage network. CVM implementations on all supported UNIX and Linux platforms recognize and properly handle CDS disks, regardless of platform type. If all the disks in a CVM disk group are CDS disks, volumes constructed from them can act as Portable Data Containers, and the file systems on them can be serially shared between any combination of Linux, Solaris, AIX, and HP-UX platforms supported by Storage Foundation.

Administrative hint 6

For cross-platform data migrations that are performed on a regular basis, an administrator can avoid the need to enter migration parameters every time by using the **fscdsadm** console command to record them permanently.

To share data serially between unlike platforms, not only must source and destination platforms recognize the disk formats, they must also recognize and accommodate the file system format. While CFS file system metadata is structurally identical across all platforms, numerical values in it are endian-specific (unlike CVM volume metadata), primarily because the frequency with which numeric items in file system data structures are manipulated makes real-time conversion between big and little endian representations impractical. Therefore in order for a file system to be moved between a big-endian platform and a little-endian, CFS must convert the file system's metadata between big and little endian representations.

Moreover, not all operating systems supported by CFS observe the same limits on file size and group and user IDs. For example, a CFS file created on a Solaris platform might be too large to be opened by a Linux application, or its owning userID may be too large to be represented on a Linux system.

CFS's **fscdsconv** utility program both verifies that the CFS implementation on a specified destination platform can accommodate all files in a source file system, and converts the source file system's metadata structures to the form required

by the destination platform. To move a CFS file system between two unlike platforms, an administrator executes the following steps:

- **Unmount.** Unmount the file system to prevent applications from accessing it during conversion
- **Convert.** Run the **fscdsconv** utility against the device that contains file system to convert its metadata and to discover any minor incompatibilities between the source and destination platforms
- **Resolve.** Make adjustments to resolve any minor incompatibilities such as userID range and path name lengths
- **Deport.** Split the disks that make up the file system's volumes into a separate disk group, and deport the group from the source cluster
- **Import.** Import the disks to the destination system and create a VxVM disk group and volumes
- **Mount.** Mount the converted file system contained on the imported volumes for use by applications on the destination system

Portable Data Container conversion time is related to a file system's size, the number of files it contains, and the complexity of their layout (number of extents), and whether the source and destination platforms actually are of different endianness. It is typically several orders of magnitude less than network copy time. Moreover, using CDS does not consume any "extra" storage for a second copy of data, or enterprise network bandwidth. Even the storage network bandwidth it consumes reading and writing metadata is a small fraction of what would be required to read the entire file system contents for network transfer.

Cross-platform conversion of CFS file systems is crash-recoverable—if a system crash occurs during conversion, the administrator can invoke the **fscdsconv** utility again after crash recovery to complete or reverse the conversion.

Administrators can use Portable Data Containers to simplify and streamline the transfer of data between unlike platforms either on a one-time basis, as for example, when systems are being refreshed by platforms of a different type, or periodically, as for example, when different steps in a workflow are performed by different types of platforms.

Portable Data Containers make it possible to transfer entire file systems between unlike platforms without copying large amounts of data. CDS does not manipulate the data *within* files however. For PDC-based serial data sharing between unlike platforms to be usable, the format of data within files must be understood by applications on both source and destination platforms. For example, the Oracle database management system supports a feature called Transportable Table Spaces that enables database data to be moved between platforms. With Portable Data Containers, CFS file systems that contain Transportable Table Spaces can be moved between unlike computing platforms without bulk copying.

Feature 8: User and group quotas

CFS supports both user and group quotas. CFS quotas limit both the number of files and the amount of space that individual users and groups of users are permitted to consume. CFS file and space consumption quotas can be:

- **Hard.** CFS fails operations that would cause a hard quota to be exceeded
- **Soft.** CFS permits soft quotas to be exceeded for a limited time that the administrator can specify. After the time limit expires, no further space allocations or file creations are possible. A user or group soft quota must be lower than the corresponding hard quota if one exists

CFS stores externally-visible user and group quota limit and current consumption information in separate files in the root directory of a file system's primary fileset. In addition, per-node structural files maintain information about current space and inode (file) consumption that CFS updates as changes occur. CFS periodically reconciles all nodes' consumption information into a master quota structural file, and reconciles internal and external resource consumption data whenever quota control is enabled or disabled.

Administrative hint 7

CFS quota management commands have unique names to avoid conflicts with UNIX commands used to manipulate quotas for other types of file systems. The primary management commands are **vxquotaon**, **vxquotaoff**, and **vxedquota** (for editing quota files). The **vxrepquota**, **vxquot**, and **vxquota** commands can be used to report information about quota usage.

An administrator can enable CFS user and group quota control together when mounting a file system by using the **–o quota** mount option. Either type of quota control can be enabled or disabled independently by issuing the **vxquotaon** and **vxquotaoff** administrative commands with the corresponding parameters.

When quota control is enabled, CFS automatically checks usage against quota limits as applications attempt to create files or append data to existing ones. No explicit quota checking is required. If file creation or appending is subject to both user and group quota control, CFS applies the more restrictive of the two.

Although external quota files are editable with a text editor, administrators should use the **vxedquota** command to edit them in order to avoid inadvertent formatting errors. CFS quota management commands are available to CNFS cluster administrators, but not to NFS client computers.

Administrative hint 8

An administrator can use the **setext** console command to pre-allocate space for an ordinary file. If Quick I/O (page 36) is enabled, the **qiomkfile** command can also be used to create Quick I/O files with pre-allocated space.

Feature 9: Sparse files

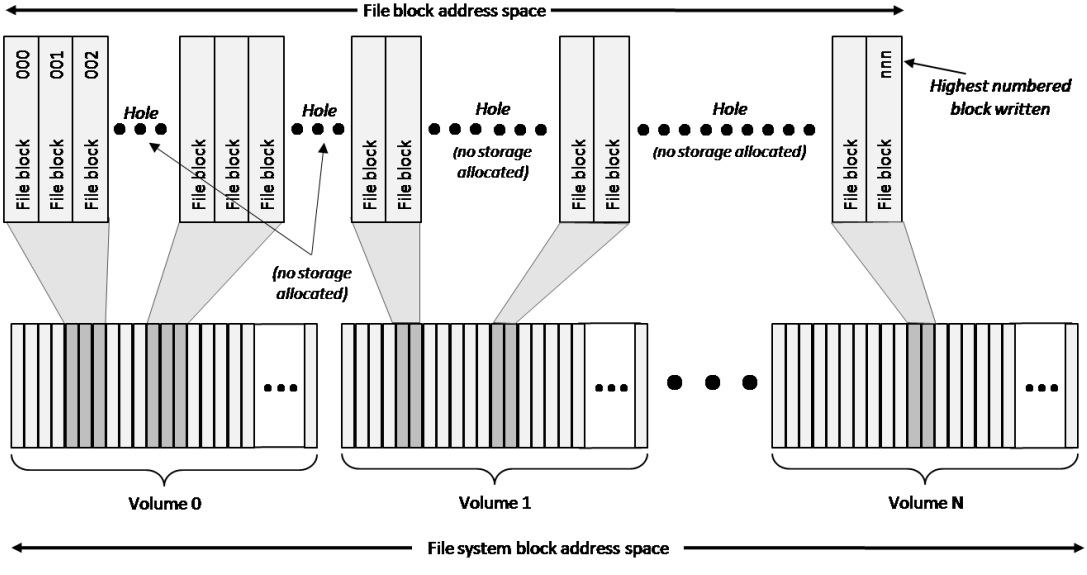
Unless space for them is pre-allocated, CFS files are inherently "sparse"—CFS does not allocate storage space for file blocks until an application writes data to the blocks. In this respect, CFS files may be thought of as being thinly provisioned.

Administrative hint 9

An administrator can use the fsmap console command to locate the "holes" in the file block address space of a sparse file.

Any CFS file in which file blocks are first written non-sequentially is automatically sparse, The file's metadata reflects a notional size that encompasses the largest file block address written, but storage capacity is only allocated for file blocks that an application has actually written. Sparse files consist of extents (ranges of file system block addresses) in which applications have written data and *holes*—parts of the file block address space for which no file system blocks have been allocated. Figure 1-5 illustrates storage allocation for a sparse file.

Figure 1-5 A sparse file



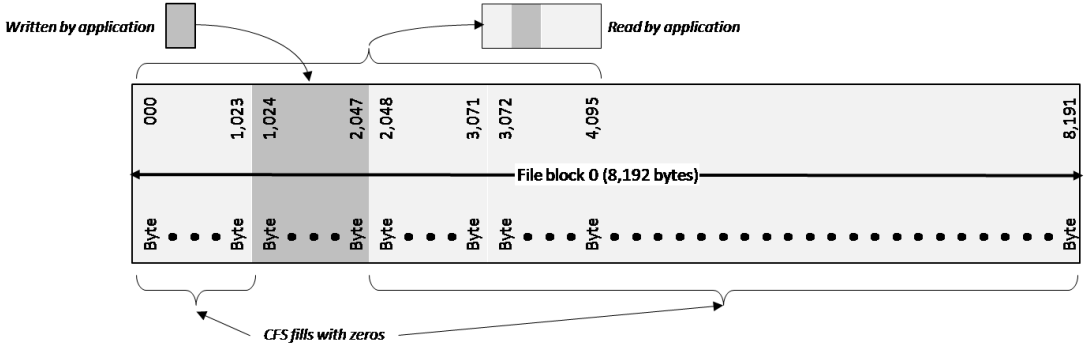
CFS does not expose files' sparseness when applications read and write data. When an application writes data into a file byte range for the first time, CFS allocates the necessary file system blocks, and fills any areas not written by the application with zeros. For example, if a file system's block size is 8,192 bytes, and an application writes data to file byte addresses 1,024-2,047 for the first time, CFS allocates a file system block, zero-fills bytes 0-1,023 and bytes 2,048-

8191, and writes the application's data into bytes 1024-2047 of the block.

By zero-filling newly allocated partially written file system blocks, CFS guarantees that it never returns "uninitialized" data to an application's read request. If an application reads from file block addresses that have never been written, CFS simply fills its read buffer with zeros. If it reads data from an as-yet unwritten area of a file block to which it has written data, CFS returns the zeros with which it filled the block when it was allocated. Continuing the foregoing example, if the application were to follow the write immediately by reading file bytes 0-4,095, CFS would return a buffer containing:

- **Leading zero fill.** The 1,024 bytes of zeros written when the file system block was allocated to the file
- **Application data.** The 1,024 bytes of data written by the application
- **Trailing zero fill.** 2,048 of the 6,144 bytes of zeros written when the file system block was allocated to the file

Figure 1-6 An example of sparse I/O



Sparse files are particularly useful for applications that need simple indexing schemes to manage large data structures. For example:

- **Indexed records.** Some applications manage indexed records with sparsely populated index spaces. For example, a 7-digit index (for example, a telephone number) is an index space of ten million potential entries. Locating records by multiplying their index values by the record size avoids the necessity for the application to manage complicated index trees. Because CFS only allocates space when data is first written to a file block range, the file occupies storage space in proportion to the number of actual records, not in proportion to its potential size
- **Sparse matrices.** Many scientific and engineering applications involve matrix calculations that are too large to be contained in memory. They perform their calculations on sub-matrices and write intermediate results to persistent storage. A common characteristic of these applications is that many of the sub-matrices are known a priori to contain zeros. Sparse files

simplify these applications by making it possible for them to treat a file as though it were large enough to contain the entire matrix, and simply not write to any of the file block addresses that represent zero sub-matrices. CFS allocates storage only to parts of the matrix written by the application, so actual storage consumed is related to size of the non-zero sub-matrices, not to the size of the entire matrix

In both of these examples, CFS effectively provisions file storage for the applications as they require it, which they signal by their initial data writes to file block addresses. The applications are spared the complexity of having to manage index trees or other storage space management structures, and at the same time need not grossly over-provision storage.

CFS itself makes use of the sparse file concept to simplify the indexing of user and group quota files in file system structural file sets. For example, there are four billion possible unique user IDs and group IDs in a UNIX system. Obviously, no file systems even approach that number of actual users or groups. CFS computes a record number for each user or group for which a quota is assigned by multiplying the user or group ID by the size of a quota file record to give an offset into the respective quota file block address space. When it writes a user or group quota file record for the first time, CFS allocates storage space and creates an extent at whatever file block offset is indicated by the write request. Users and groups to which no quotas have been assigned remain as holes in the file block address space.

Feature 10: Storage tiers

For applications that must keep tens or hundreds of terabytes of data online, the cost of storage matters. Many enterprises control cost by adopting the concept of two or more storage *tiers*—sets of storage devices that differ significantly in cost, and as a consequence, typically have different I/O performance and availability characteristics. Enterprises store especially critical data, or data that is frequently accessed, on the "top" (highest performing, most robust, and hence, most expensive) tier, and less critical data on "lower," more economical tiers. As files progress through different phases of their life cycles, administrators relocate them from tier to tier according to their importance to the enterprise at the moment, perceived performance requirements, or other factors.

The viability of storage tiering depends entirely on an ability to place files on a storage tier commensurate with their value and I/O requirements, and to relocate them as their importance or requirements change (for example, as they age, as the I/O activity against them increases or diminishes, and so forth). For data centers that manage millions of files, this can be an expensive task, requiring significant administrative effort and skill. Moreover, it is difficult to "get it right"—to precisely match millions of constantly changing files with the right type of storage, and to adjust operating procedures and applications so that they can continue to find the data they need to process as it moves

throughout its life cycle. The combination of administrative cost and susceptibility to error inhibit some data centers from taking advantage of the cost reductions inherent in the storage tiering concept.

Two unique features of CFS completely automate file placement and relocation, and thereby make it possible to fully exploit storage tiers of different performance and resiliency with minimal service disruption and administrative effort:

- **Multi-volume file systems (MVFS).** Conventional UNIX file systems are inherently homogeneous (single-tier) because they occupy a single disk or virtual volume. A CFS file system, on the other hand, can occupy as many as 8,192 CVM volumes. The volumes occupied by a CFS file system are called its volume set, or *VSET*. An administrator organizes each file system's VSET into *storage tiers* by assigning *tags* to them. Identically tagged volumes in a file system's VSET form a storage tier.

  For example, volumes that mirror LUNs presented by two disk arrays might be labeled **tier1**, and volumes constructed from high-capacity, low-RPM SATA drives **tier2**.

  Because a CFS file system's name space encompasses all of its volumes, CFS can allocate storage for a file on any volume in its VSET, based, for example on its file type or owner

- **Dynamic Storage Tiering (DST).** An administrator of a CFS file system can define a policy that causes files to be automatically allocated on specific storage tiers based on their names, types, sizes, and other attributes. If a file system's volumes are tagged so that each tier consists of similar volumes, and so that different tiers have distinctly different cost, performance, and availability properties, the effect of the policy is to place files appropriately as they are created.
  In addition, DST periodically scans a file system's directory tree and automatically relocates files from one tier to another based on policy rules that specify frequency of access, position in the name space hierarchy, size, and other criteria that can change during a file's lifetime. DST relocates files' data between volumes, but does not change their logical positions in the directory hierarchy that applications use to access them

Thus, not only does CFS automatically place files on the proper storage tiers initially; it automates the process of relocating them to appropriate storage tiers at different phases of their life cycles. Throughout the process, files' logical positions in the file system directory hierarchy remain constant, so from the user and application point of view, relocation is completely transparent. From the administrative point of view, however, storage utilization is optimized and service level agreements are met, down to the level of individual files.

Chapter 10 on page 171 has a detailed survey of CFS multi-volume file systems and dynamic storage tiering.

Using CFS

The fundamental properties of CFS—concurrent access to highly available shared file systems by applications running on as many as 32 nodes in a cluster—make it a particularly suitable solution in several scenarios that occur frequently with mission critical enterprise applications. These include:

- Fast failover, particularly for database management systems in three-tier database application architectures
- Highly available concurrent data access for multiple applications, including workflow applications in which processing steps operate on data sequentially
- Storage consolidation for efficiency and economies of scale
- Highly scalable NFS file serving

The chapters that follow describe these and explain why CFS is the ideal data management solution in each case.

Chapter 2

Using CFS: application scenarios

This chapter includes the following topics:

The fundamental advantages of CFS are high availability and scaling. Applications that use CFS to access their data can restart on alternative servers in a VCS cluster and continue to operate if the servers on which they are running fail. Alternatively, as many as 32 instances of an application can serve clients from the same CFS data files, scaling both compute power and client connectivity to very high levels. These two properties can be combined in client-server applications structured as VCS parallel service groups, that balance client load across instances and use virtual IP address (VIP) service groups that redirect clients to alternate application instances in the event of a server failure. (The CNFS file server described in Chapter 3 uses this technique to balance load and keep NFS services highly available.)

These properties of CFS clusters make them particularly suitable for certain types of applications and data center scenarios:

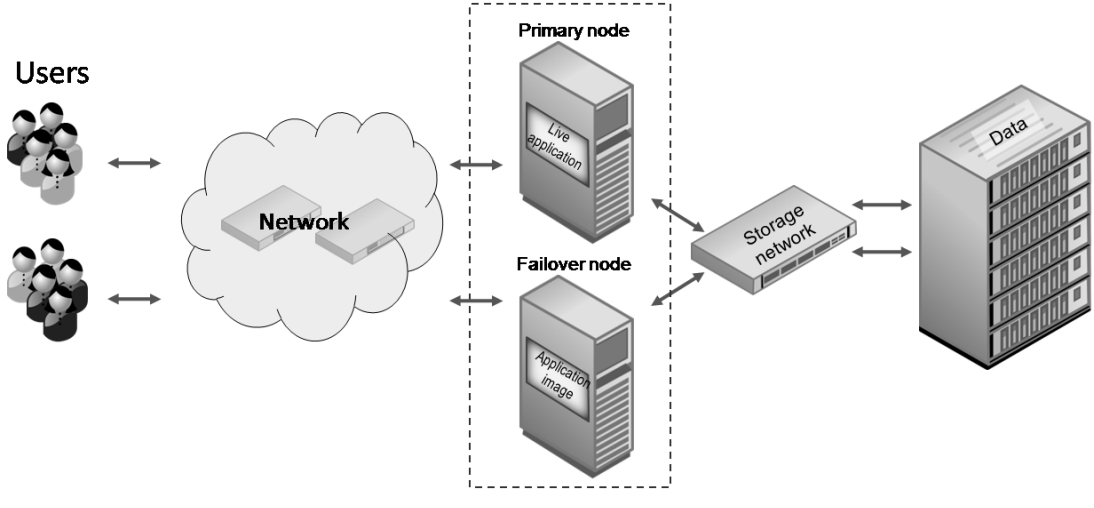
- **Fast failover.** Compared to other highly available system architectures, CFS clusters typically fail over more quickly because volumes and file systems are already imported and mounted respectively on failover nodes. This eliminates a major source of failover time

- **Workflow applications.** Application suites in which modules execute different stages of a work flow and pass data among themselves are particularly well-suited to CFS cluster implementations. Passing a file in a CFS file system from one workflow application stage to another is as simple as sending a message to the receiving stage that the file is ready to be operated upon
- **"Scale out" applications.** An increasing number of applications are achieving scale by running in parallel instances on multiple servers. Some applications of this type operate on data partitions, but more frequently, all instances must operate on a common data set. Because CFS file systems can be mounted on up to 32 cluster nodes simultaneously, it is an ideal solution for application "scale out"
- **Storage consolidation.** The storage connected to a CFS cluster is completely interchangeable among the cluster's nodes and file systems. If applications are consolidated into a CFS cluster, redeploying storage from one to another to meet changing needs becomes a simple administrative operation. The need for physical reconfiguration and the attendant inflexibility are eliminated

Basic application clustering

Figure 2-1 represents the simplest form of high-availability cluster—the active/passive failover cluster. In this configuration a *primary node* runs a live application which processes data stored on disks connected to it by a storage network. A second, *failover node*, also connected to the storage network, has the application installed but not running.

Figure 2-1 Active/passive failover cluster



The goal of an active/passive cluster is to provide continued application availability to clients if the primary node, the links connecting the primary node to clients or storage, or the application executable image itself should fail. Cluster managers such as VCS monitor the nodes, the links, and the live application. If they detect a failure critical to application execution, they initiate failover by transferring control of the network resources to the failover node, performing any necessary data recovery, and starting the application on the failover node. One of the network resources transferred to the failover node is the IP addresses that clients use to communicate with the application. Except for a time delay while failover is accomplished, and in some cases, the need to reconnect and re-authenticate themselves, clients experience no effects from the failover—they still communicate with the same application using the same IP addresses.

The basics of highly available applications

While clusters exist in several forms, their common architectural principle is to integrate redundant computing, connectivity, and storage resources into the environment. Under the control of the cluster management software, these redundant resources assume the function of the primary resources in the event of a failure. Thus, in addition to a redundant failover node, a cluster would typically also be equipped with redundant storage and client network paths.

In an active/passive cluster, cluster software automates:

- **Failure detection.** The primary node, a network link, or the application itself may fail. Cluster management software detects and responds to all of these failures
- **Cluster reconfiguration.** If the failure is a node failure, the failed node must be ejected from the cluster, and the remaining nodes must agree on cluster membership
- **Transfer of resource control.** The cluster manager withdraws control of data storage devices and client connections from the failed node and enables them on the failover node
- **Data recovery.** In general, failures occur while I/O operations or application transactions are in progress. Some form of “cleanup,” such as file system checking or log replay is required
- **Application restart.** The cluster manager starts the application instance on the failover node
- **Client reconnection.** Clients must reestablish communication with the restarted application

The sum of these processes is called *failover*. The new application instance is often called the *failover instance*, and the node on which it runs, the *failover*

node. In some situations, resource utilization can be improved by running non-critical applications such as data mining on failover nodes during periods of normal operation. In others, failover time is critical, so the failover application is pre-started and idling, ready for instant takeover if the primary instance fails.

Failure detection

Cluster managers like VCS typically detect node failures through regular heartbeat messages from each node to others in the cluster. If several heartbeat intervals pass with no message being received from a node, other nodes conclude that it has failed, initiate cluster reconfiguration.

Failure of other critical resources, such as network links, the application itself, or auxiliary applications like print services, is typically detected by a similar heartbeating mechanism within the node. VCS, for example, monitors each of an application's resources by periodically executing a script or program that is specific to the resource type. If a critical resource fails to respond properly, VCS initiates failover of the application that depends on it to another node.

Cluster reconfiguration

If a cluster node fails, as opposed to an application or other resource, the remaining nodes must eject the failed one and converge on a common view of cluster membership. (A similar reconfiguration occurs when a node is added to a live cluster.). In the case of VCS, a specialized Group Atomic Broadcast protocol includes message types and conventions that enable the surviving nodes of a cluster to reach consensus on membership within seconds.

Any cluster technology must be able to distinguish between a failed node and partitioning, or partial failure of the network links that the nodes use to intercommunicate. A primary node that has crashed will not interfere with a restarted application running on a failover node, but a node that cannot communicate with other nodes must have a foolproof means of determining whether it is part of the surviving cluster or has been ejected. VCS uses three *coordinator disks* (see "Coordinator disks and coordination point servers" on page 34.) to make this distinction.

Transfer of resource control

A failover node must obtain control of the storage devices that contain the failed application's data. If an application or database management system fails, its storage devices must be deported (removed from the primary node's control). Whether the primary node itself, or some critical application resource was the

failure, the failover node must import (take control of) the application's storage devices. In systems with hundreds of file systems and virtual volumes, deporting and reimporting volumes and remounting file systems can take hours. Because CVM shared disk groups are imported on all cluster nodes, this potentially time-consuming step is eliminated for clusters implemented with the VCS-CVM-CFS stack.

Recovering data

Before data can be accessed by restarted applications on failover nodes, the structural integrity of file systems and databases on the imported volumes must be verified, and repaired if necessary. Full file system checking of very large file systems can take days. In almost all cases, however, a CFS file system's structural integrity is restored after a node failure when the primary node replays the failed node's log of outstanding transactions. This is an important timing consideration for relational databases that store data in CFS container files, because database recovery cannot begin until the integrity of the underlying file system has been restored.

Application restart

Once file system structural integrity has been assured or restored, applications can restart. Some applications are *stateless*—each interaction with a client is independent of all prior actions. For these, restart is the same as initial startup. The NFSv3 server in CNFS (Chapter 3 on page 71) is of this type (as long as the Network Log Manager is not in use). More often, applications conduct multi-step transactions with their clients; if these are interrupted by a node failure, they must be recovered in a manner similar to file system recovery. Relational database management systems fall into this category. Before they can resume service to clients after a failure, they must verify database integrity by replaying their own work in progress logs.

Client reconnection

The final piece of recovery is reconnection of clients to the restarted application on the failover node. Cluster managers typically transfer control of virtual IP addresses used by clients to the failover node as part of the failover process. When clients attempt reconnection, they are connected with the restarted application on the failover node. From that point, interaction is client-specific.

Some client applications, such as NFS, retry failed operations until they succeed. NFSv3 in particular is *stateless*—no client request depends on any previous request, so no further reconnection protocol is required. For applications like

this, reconnection is transparent, although users may notice a time delay. Other applications require that a user or administrator re-establish the connection with the application, including authentication and other reestablishment of context.

CFS and active-passive failover

Table 2-1 lists typical timings for the main functional components of application failover in a CFS cluster environment.

Table 2-1 Why CNFS for NFS file sharing

Contributor to failover time	Conventional architecture	CFS cluster	Comments
Fault detection	~20 sec	~20 sec	Detection is actually much faster. Typically, cluster managers require multiple evidence of failure to avoid "false failover"
Cluster reconfiguration	5-10 sec	5-10 sec	Time for nodes to negotiate new membership
Storage device transfer	Varies	0	Depends on number of devices and volumes Can be minutes or longer
File system verification	Varies	<1 sec	Depends on file system verification technology.
Infrastructure total		25-30sec	CFS eliminates the two largest and most variable components of failover time
Client reconnection	N × 30 sec	N × 30 sec	N = number of TCP connect request timeouts. May be partly concurrent with other failover steps

Depending upon the number of volumes and file systems used by an application, transfer of storage device control and file system verification can take minutes or longer. These are also the least controllable steps in terms of the time they take, because they depend both on the number of volumes and file systems, and on the number of file system transactions in progress at the time of the failure. Using CFS to store application data or for database container files essentially eliminates both of these contributions to failover time.

During normal operation, all cluster nodes import (make available to CFS) all shared CVM disk groups, so all nodes have simultaneous access to shared volumes. When a cluster node or application fails, the volumes that hold its data and the disks that make them up are already known to and accessible by the

failover node; deportation and re-importation of shared disks is completely eliminated. At worst, if a failed node is CVM Master, I/O to shared volumes pauses momentarily while the remaining cluster nodes select a new master.

Similarly, CFS shared file systems are mounted on all cluster nodes during normal operation, so full verification and re-mounting are not required. CFS recovery consists of electing a new primary instance if necessary, recovering locks that were mastered by the failed node and replaying the failed node's intent log to complete any transactions in progress at the time of the failure.

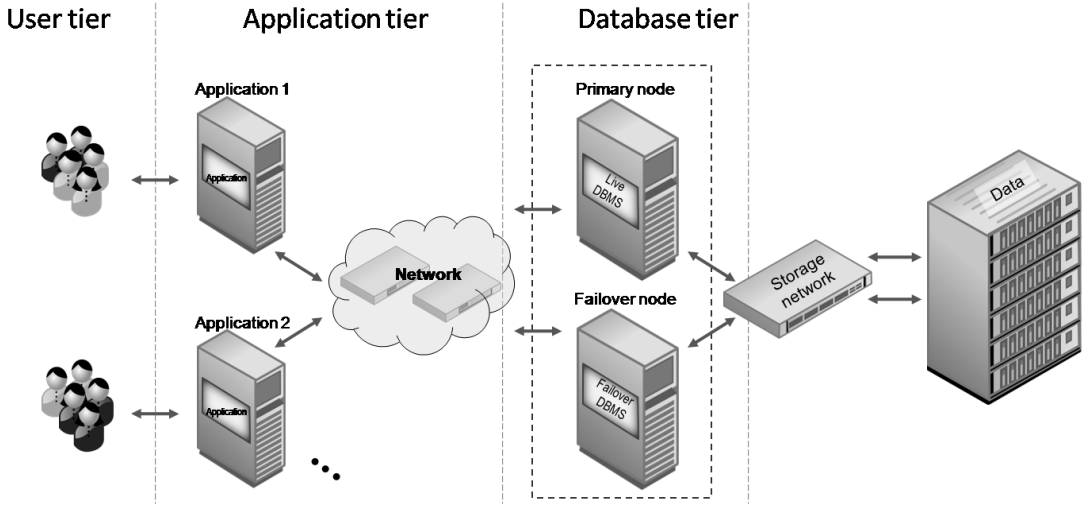
Because CFS virtually eliminates the two most time-consuming steps, typical application failover times are under a minute, compared with failover times in minutes or tens of minutes that are characteristic of approaches requiring storage device and file system failover.

CFS and highly-available database applications

One type of "application" that can benefit from active-passive clustering is relational database management systems such as Oracle. These are often deployed to manage the data in a *three-tier* architecture such as the one illustrated in Figure 2-2. In this architecture, presentation and user interface, business logic, and data management run in separate tiers of computers:

- **User.** The user tier employs "thin clients"—minimally equipped desktop computers equipped with browsers of forms management software
- **Application.** Servers in the application tier implement business functions. Different platforms may be employed, as may different levels of availability and performance. Many unrelated applications may access a single database or set of databases
- **Data management.** This tier runs the database management system (DBMS) and provides database access to clients in the application tier

Figure 2-2 Three-tier database application architecture



The basic principle of this three-tier architecture is separation of function. A separate data management reinforces a canonical representation of enterprise data that is independent of individual applications. Similarly, separating business logic from data insulates applications from each other, but at the same time, integrates them through the common data they process. It becomes easier to deploy new applications, particularly with virtual servers, without being concerned about application cross-talk. A secondary benefit of the architecture is independent scaling at each layer. Either data management or application capacity can be added as required.

With one database serving multiple applications, availability becomes especially critical. For this reason, database management systems are often deployed as the “application” in active-passive clusters. The database management system and its remote client access module are structured as service groups that run in a primary cluster node, with a failover node at the ready.

In this scenario, failover time is critical, because when the database management system is down, none of the applications that depend on it can function. Using CFS files as containers for database data is particularly beneficial in this scenario, because it eliminates the entire transfer of storage device control component of failover listed in Table 2-1 on page 58. Within half a minute after a failure, a failover database management system can be replaying the failed instance’s activity logs to restore database to complete or reverse transactions in progress at the time of failure so that service can be restored to clients.

CFS and clustered database management systems

Most database management software vendors offer versions of their products that are themselves clustered. Oracle Corporation, for example, offers the Real Application Cluster (RAC) edition of Oracle, IBM Corporation the Enterprise Extended Edition of DB2, and Sybase the Sybase Cluster Edition. Different products differ in detail, but in general, clustered database managers consist of multiple instances that run concurrently in different cluster nodes and coordinate with each other to access a common database. The physical architecture is similar to that represented in Figure 2-2, but clusters of more than two nodes are common, and all are active concurrently, running cooperating database management instances.

Clustered database managers make fault recovery functionally transparent to clients. Moreover, because mutually aware database manager instances are already running on all cluster nodes at the time of failure, recovery is virtually instantaneous.

The VCS-CVM-CFS stack supports clustered database managers as well, providing seamless sharing in their underlying file and storage layers. In the case of Oracle RAC, for example, the Storage Foundation RAC Edition (SFRAC) includes VCS, CVM, and CFS, along with additional Oracle-specific utilities to deliver an easy-to-configure highly available storage solution for parallel Oracle databases.

Thus, the designer of a mission-critical database management system has two Storage Foundation options for creating a highly available database tier:

- **Active-passive.** An active-passive cluster in which the database management system fails over to the passive node within a few seconds of a failure (with the option to use the failover node to run non-critical applications under normal circumstances)
- **Parallel.** A parallel database cluster in which multiple database management system instances cooperate in real time to provide clients with a common view of the database as well as near-instantaneous service resumption if a node in the database tier fails

Both options use the VCS-CVM-CFS stack for rapid failover of the storage and file layers.

Given these two options, the choice would seem obvious—near instantaneous failover should always be preferable to a service interruption, even of only a few tens of seconds. But this analysis ignores two factors that are becoming increasingly important in information processing:

- **Cost.** Database management system (and other) software vendors typically place price premiums of as much as 300% on the capabilities embodied in clustered versions of their products, Moreover, additional hardware and

network facilities may be required to run them, and they are generally more complex to administer

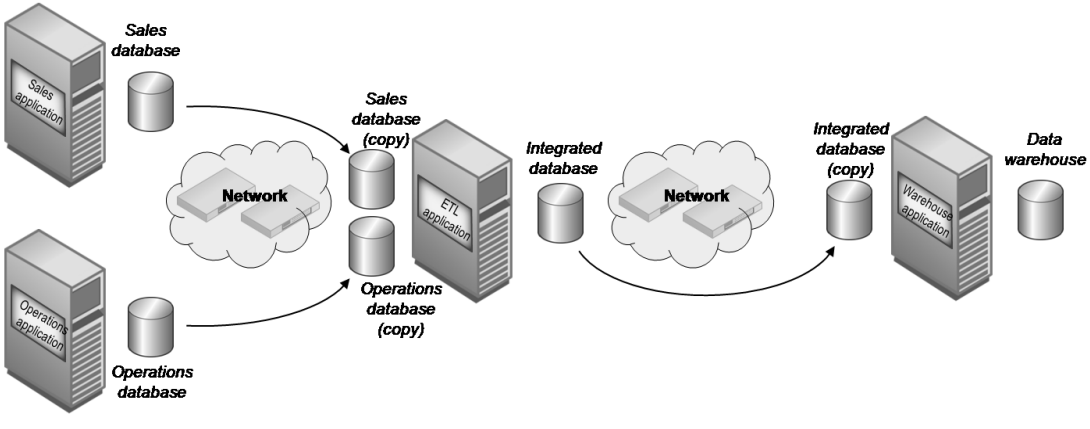
- **Performance.** Clustered database management system instances (and other clustered applications) communicate extensively with each other. Thus, while failover is nearly instantaneous with a clustered database management system, performance under normal operating conditions may actually be lower than that of an equivalent non-clustered version of the same database manager

Thus, the application designer must evaluate the value of instantaneous failover, and decide whether the incremental costs, both direct and in the form of reduced performance, of clustered database management and other application software is justified by that value. For databases and applications that "absolutely, positively must be available" all the time, the cost of a clustered database management system may be justified. For many applications, however, an "outage" of a few seconds while a database management system instance starts up on a failover node is a cost-effective compromise.

CFS and workflow applications

Another important class of application for which the VCS-CVM-CFS stack is ideal is the *workflow* application suite. Many core business processes are essentially sequences of steps in which the completion of one step is a trigger for the next step. For example, closing the books on sales for a month might trigger quarterly roll-ups, commission compensation runs, sales analyses, and so forth. Figure 2-3 illustrates a common type of workflow application that is part of many enterprises' core information technology processes.

Figure 2-3 Typical workflow business application



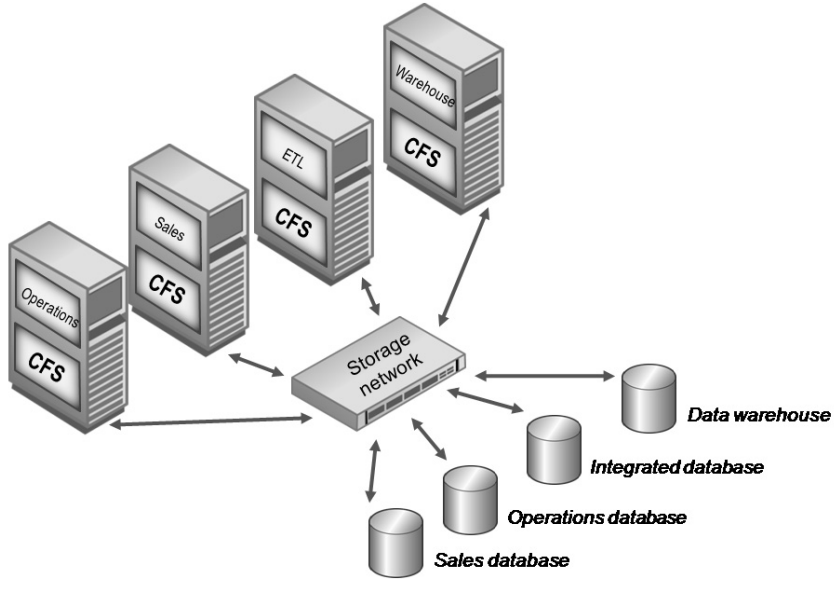
In Figure 2-3, sales and operations (shipping, inventory control, general ledger accounting and similar) applications constantly update their respective

databases. Periodically, an ETL (extraction, transformation, and loading) application extracts information from the online databases (or more likely, snapshots of them), integrates their contents and stores them in a form suitable for later business intelligence analysis, or mining, and ultimately, long-term storage in a data warehouse. Ideally, the ETL and data warehousing applications can read directly across the network from their respective source databases; in the worst case, bulk data copying, and the additional storage it implies would be required.

Reduced to their essence, workflow applications consist of processes that ingest data from prior steps, analyze or transform it, and create output data for analysis by subsequent steps. The data produced at each step must be made available to the step that consumes it. It is not uncommon for data files output by one process to be copied over an IP network to storage devices that are accessible by the next step in the chain.

Copying data in bulk consumes time as well as storage and network resources, and is therefore undesirable. A preferable solution is to use one of the forms of shared file storage described in the introduction to this paper. Figure 2-4 represents the same workflow application deployed in a CFS cluster.

Figure 2-4 Using data sharing to streamline workflow applications



As Figure 2-4 suggests, with a CFS cluster, all stages of the workflow have direct physical access to all data. (Security considerations may result in some barriers to logical access, for example, to operational data by sales personnel and applications.)

The obvious advantage of data sharing for workflow applications is the elimination of data copying and extra storage to hold copies. Less obvious, but equally important, is flexibility. If business processes change, resulting in additional workflow steps or a need for some step to access additional data sets, all that is necessary is access permission; no additional storage, bandwidth, or administrative setup time is required.

CFS clusters make particularly attractive platforms for workflow applications for several reasons:

- **Performance.** Data is accessible to all application steps at storage network speeds and latencies, with no remote file access protocol overhead
- **Flexibility.** Storage capacity can be easily re-provisioned among application steps. During high-volume selling seasons, file systems for the sales databases can be expanded. When the load shifts toward accounting and data warehousing, the sales databases can be shrunk, and the storage they relinquish allocated to other applications' file systems
- **Security.** Shared data access security is enforced uniformly across the cluster
- **Versatility.** Data that is only required by a single application step can be stored in privately-mounted file systems to minimize lock traffic. Private file systems can quickly be re-mounted in shared mode if multi-node access is needed
- **Application availability.** Critical application steps can be structured as VCS failover services to make them highly available, while non-critical ones can execute on single nodes

CFS and scale-out applications

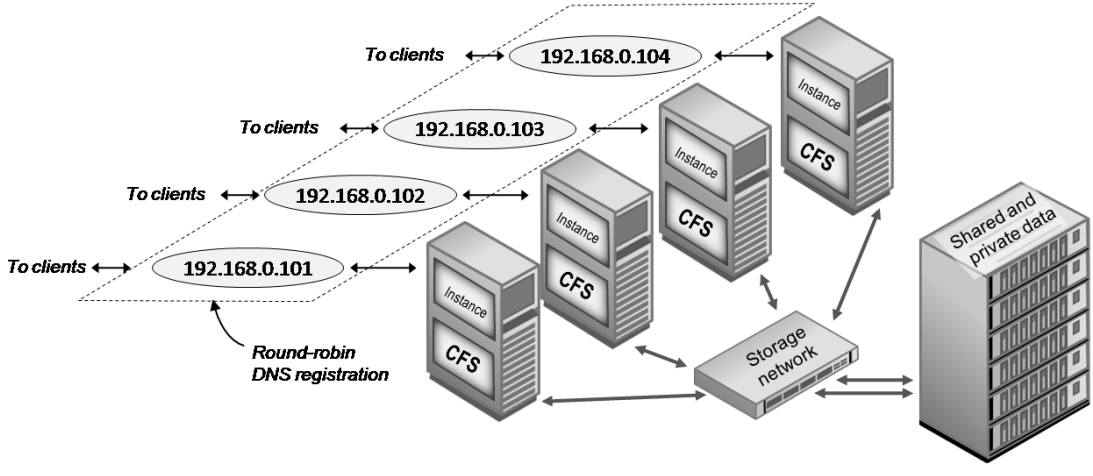
With supported configurations of up to 32 clustered servers, CFS is a natural platform for "scale-out" parallel applications in which capacity is increased by adding nodes running instances of the application to the cluster. In a typical scale-out architecture, all application instances require access to core data files, while some data, such as activity logs, is likely to be generated and managed on a per-instance basis. To this end, CVM and CFS support both:

- **Shared data.** Cluster-wide volumes and file systems for data that is shared among application instances
- **Per-node data.** Private volumes and file systems for data that is maintained on a per-node basis

Scale-out applications may consist of multiple instances of the same executable image, replicated in different cluster nodes to increase the aggregate service capacity. This design, illustrated in Figure 2-5, is a common one for high-volume business transaction processing applications such as point-of-sale data capture.

The scenario illustrated in Figure 2-5 works especially well for highly partitionable applications that serve large numbers of clients because load balancing is easy to accomplish. Network administrators register the name of the cluster or application along with a "round-robin" list of the IP addresses that are assigned to each node. As successive clients perform DNS lookups on the cluster name, they receive a rotated list of IP addresses, from which they normally connect to the first that responds. This tends to balance client connections across nodes, and therefore across application instances.

Figure 2-5 Homogeneous scale-out application model



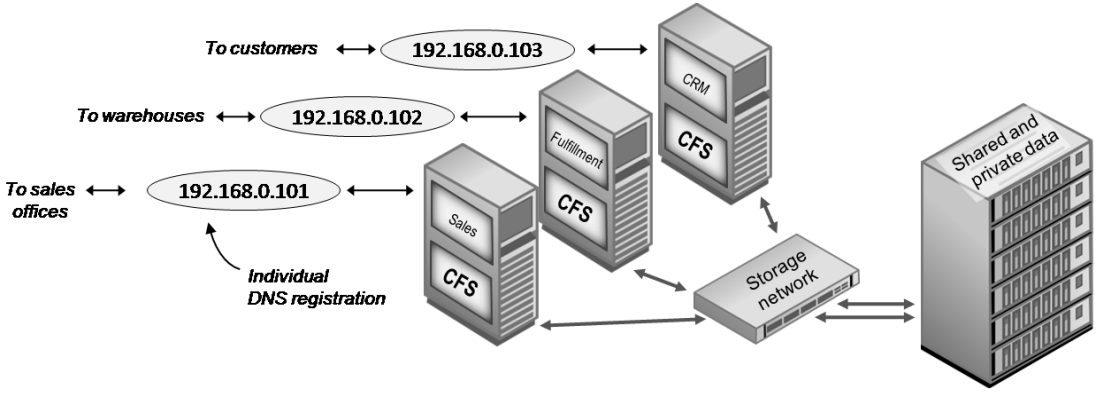
In this model, each node's IP addresses are typically structured as VCS failover service groups with designated alternate nodes. These are called *virtual IP addresses*, or VIPs, because they can move from node to node during failover. Each VIP has a dependency on the parallel service group that represents the actual application instance. If the application instance fails, VCS fails the VIPs that depend on it over to other nodes.

In this design, when a node fails, its VIPs fail over to other nodes, which start the service groups that represent them. Like the primary node's VIP service groups, these groups have dependencies on the local application instance's service group, which is running. Once started, the VIPs on the failover nodes accept messages from clients and pass them to their local application instances. Application instances must be prepared to handle incoming messages from clients. They may be stateful, and require that the client reestablish its credentials by logging in, or like CNFS services (Chapter 3), they may be stateless, and simply accept and process incoming messages.

Another type of scale-out application bears a resemblance to workflow. These are asymmetric scale-out applications in which different modules on separate servers perform different operations on a common or overlapping set of data objects. Figure 2-6 is an example of this type of scale-out application in which a

sales module generates orders which it passes to a fulfillment module. The fulfillment module generate packing and shipping orders and communicates them to warehouses. As the warehouses fulfill orders, they communicate to the fulfillment module, which forwards shipping and tracking information to a customer relationship management module that permits customers to track their shipments directly.

Figure 2-6 Workflow-style scale-out application model



As Figure 2-6 indicates, in this example, each major function of the integrated application executes on a different server. All operate on common data, however. The primary distinction between asymmetric scale-out and workflow applications is the granularity of shared data. Whereas workflow applications tend to operate on large batches of data, scale-out applications share at the transaction level. As a result, the modules of an asymmetric scale-out application tend to communicate by placing messages on queues. CFS is an ideal vehicle for making message queues persistent, so that they are readily shareable by producers and consumers, and also so that failure of a cluster node or an application module does not result in lost messages.

CFS and storage consolidation

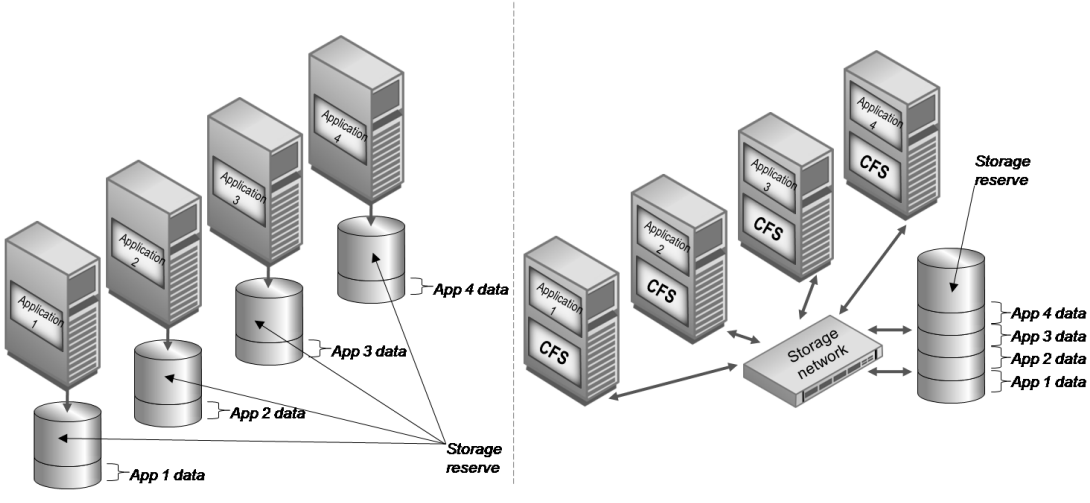
Groups of unrelated applications that run on the same UNIX platform can benefit in two ways if their servers are consolidated into a VCS cluster and their file systems are managed by CFS:

- **High availability.** Applications that run on nodes of a VCS cluster can be configured as high-availability service groups so that if a node fails, the application it is running can restart on a failover node with full access to its data, sharing the failover node's computing and network resources with the application that normally runs on it

- **Storage utilization.** Consolidating applications' file systems onto CVM shared volumes creates opportunities for improving storage utilization, because it makes the underlying storage shareable. If individual applications' file systems reside on shared volumes, imbalances between applications provisioned with excess storage and those provisioned with too little can be redressed by administrative operations while the applications are running

Figure 2-7 illustrates how a CFS cluster can minimize storage consumption across the data center by consolidating the storage resources for completely unrelated applications.

Figure 2-7 Storage consolidation with CFS



Application designers and managers often provision for "worst case" data storage requirements because once an application is deployed, provisioning additional storage for it can be a time-consuming operation involving the operations and network management organizations as well as the owner of the application. This leads to the situation illustrated in the left panel of Figure 2-7, in which storage, whether directly attached or on a network, is dedicated to applications that use only a relatively small fraction of it.

This contrasts with the CFS cluster approach illustrated in the right panel of Figure 2-7. In this scenario, the applications run on nodes of a CFS cluster, and all storage is managed as shared CVM volumes. Each node mounts its own application's file systems; no application has access to any other application's data.

But because CFS file systems and CVM volumes are resizable, storage capacity can be taken from applications that are over-provisioned and given to those that require additional capacity using Storage Foundation administrative operations performed by the cluster administrator, with no need to involve the storage or network administration functions.

Online re-provisioning of storage

An administrator can increase or reduce the storage complement of a CFS file system in one of two ways:

- **Volume resizing.** The administrator can increase the size of one or more of the CVM volumes occupied by a file system, and the file system expanded to utilize the additional capacity. The steps can be reversed to decrease the space available to a file system. In most cases, both volume and file system resizing can be accomplished in a single step
- **Addition of volumes.** The administrator can add volumes to a file system's volume set. Capacity added in this way automatically becomes part of one of the file system's storage tiers. To reduce file system size by removing volumes, the administrator first evacuates the volumes, and then removes them

Thus, if one application controls more storage than it requires, and another is running short, the administrator can shrink the first application's file system removing or resizing volumes and provisioning the released capacity to the second application.

Multi-tenant CFS file systems

If application performance considerations, data set sizes and data center security policies permit, provisioning storage for multiple applications can be made even more flexible by consolidating their file systems into a single CFS file system (for example, as top-level subdirectories of the file system root). With a single file system, all underlying storage is available to whichever application requires it. As applications create, extend, and delete files, storage CFS allocates or frees space dynamically. Thus, all available storage is effectively part of a pool whose granularity is the file system block size. No unused space is dedicated to any one application, and applications can consume the space they require, up to the limit of the file system size.

CFS provides full POSIX isolation of applications data, using either user and group access restrictions or inheritable access control lists that limit file and directory access to specific users and groups. Thus, by running each application under specific user and group identifiers, and applying the appropriate protections to files and directories, administrators can isolate applications from each other, even if their data resides in the same file system.

In this scenario, administrators can use CFS hard and soft quotas to regulate the amount of storage available to individual applications. From a storage flexibility standpoint, quotas have the advantage over separate file systems occupying separate volumes, because while they regulate the *amount* of storage an application can consume, they do not restrict specific applications to specific

blocks of storage.

Administrators can configure Dynamic Storage Tiering ("CFS Dynamic Storage Tiering (DST)" on page 172) to bias the allocation of specific applications' files to specific storage tiers based on the user and group IDs of the applications. Thus, administrators can give preference to certain applications without enforcing hard restrictions on their storage allocation.

Consolidating the storage for multiple *tenants* (applications) into a single file system has two advantages:

- **Granularity.** Each data center has some minimal unit in which it manages storage capacity. For some it is a disk or RAID group; for others a standardized virtual volume. When a data center consolidates applications into a cluster, this unit, which may be terabytes, becomes the granularity with which storage can be moved between applications. When multiple applications' data is consolidated into a single CFS file system, however, individual applications allocate and free storage space in units as small as a single file system block
- **Administrative simplicity.** While CFS file systems and the volumes on which they reside can be expanded and contracted, expansion and contraction are administrative operations. When multiple applications' data resides in a single file system residing on a single volume set, each application has instant access to the file system's entire pool of storage, subject only to restrictions imposed by hard and soft quotas

But there are limitations associated with sharing a file system among unrelated applications as well. Chief among them is the 256 terabyte size limitation on CFS file systems. A second limitation is the performance demands made by multiple applications running on different servers. While CFS inter-node resource locking is minimal for files and directories that are not actually shared, file system metadata is shared among all instances whose nodes mount it, and some inter-node lock traffic is inevitable.

Chapter 3

Using CFS: scalable NFS file serving

This chapter includes the following topics:

Figure Intro-3 on page 13 illustrates the network-attached storage (NAS) model for shared file system access, and the accompanying text describes the two primary factors that limit its ability to scale:

- **Latency.** Because they represent a higher-level abstraction, file-level data access protocols necessarily entail more translation between requests forwarded by clients and I/O operations on disks than their block-level counterparts. Moreover, they are less amenable to zero-copy I/O, so data read and written by clients must often be copied in memory. Primarily for these reasons, NAS head processors tend to saturate, especially when client I/O workloads are I/O request- or metadata operation-intensive
- **Bottlenecking.** All I/O to a NAS system passes through and is processed by the NAS head—essentially a server with certain processing, memory, and I/O bandwidth resources. Depending on the nature of the I/O workload, at least one of these resources tends to become the limiting factor in a NAS system's throughput

A third limitation of the NAS model has more to do with implementation than with architecture—inflexibility. Typically, NAS systems offer limited configuration options. The number and types of disks, processors, cache

memories, and network connections that can be configured with a given system tends to be fairly narrow. It can be difficult to configure a NAS system with very powerful processing capacity to handle metadata-intensive workloads, or with very large (but cost-effective) storage capacity to handle lightly loaded archival applications.

But even with these scaling and flexibility limitations, the administrative simplicity and universal applicability of the NAS model make it attractive to enterprise storage users. NAS technology is readily available in the form of purpose-built systems, but since all major UNIX operating systems include NFS server software components, many users choose to deploy conventional servers with back-end disk arrays as dedicated NAS systems.

CFS as a basis for scalable NAS file storage

The properties of the CFS architecture make it an ideal foundation for relieving the limitations that characterize NAS storage systems:

- **Performance scaling.** CFS can be configured in clusters of up to 32 nodes, each with multi-core processors, large memories, multiple high-performance gigabit Ethernet interfaces for client access, and multiple storage network interfaces for access to back-end storage
- **Storage flexibility.** CFS supports almost any combination of Fibre Channel, SAS, and iSCSI-connected storage devices. These can be configured as storage tiers with different cost and performance characteristics
- **File system scaling.** CFS supports individual file systems of up to 256 terabytes capacity and up to a billion files per file system, with no practical limit on the number of file systems hosted by a cluster
- **Advanced features.** CFS supports Storage Checkpoints (space-optimized snapshots and writable clones of a file system) and automatic policy-based relocation of files between storage tiers, advanced features that are increasingly expected to be part of enterprise-class file storage systems
- **Availability.** CFS-based clusters are inherently highly-available, able to sustain both network, processor node, and storage failures without interrupting service to clients
- **Universality.** CFS runs on all major UNIX and Linux platforms, from the very economical to enterprise-class servers with up to 64 multi-core processors. Enterprises can select the most appropriate platform for file storage, based on requirements for capacity, resiliency, and performance

CNFS: integrated scalable NFS file serving

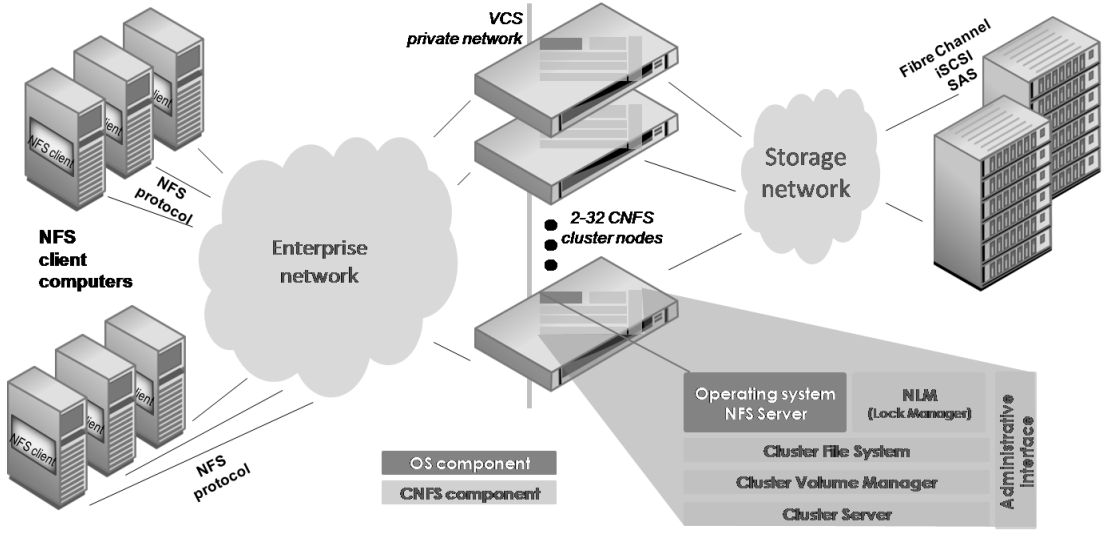
UNIX and Linux operating systems include both client and server-side Network File System (NFS) software. Any UNIX system can act as a NAS server, or alternatively, as a client, using NFS to access file systems hosted by another server. File systems hosted by NFS servers integrate into clients' directory hierarchies; for most purposes, they are identical to local file systems.

To deliver the capabilities enumerated in the preceding section over NFS, CFS integrates its host operating systems' Network File System (NFS) server components into the VCS-CVM-CFS framework to create scalable *clustered NFS* (CNFS) file services. With CNFS, administrators can configure highly available, high-performing, high-capacity NAS servers for NFS clients.

CNFS architectural overview

Figure 3-1 illustrates NAS file sharing based on CNFS file services. As Figure 3-1 suggests, each cluster node runs the VCS-CVM-CFS stack, and in addition, the host operating system's NFS server, which is encapsulated in a VCS parallel service group. Each client connects to an NFS server instance in one of the cluster nodes. The server instances execute NFS requests directed to them by making file system requests to their local CFS instances. The CFS instances cooperate to coordinate the NFS servers' concurrent access to file systems.

Figure 3-1 Architecture for CNFS-based file services

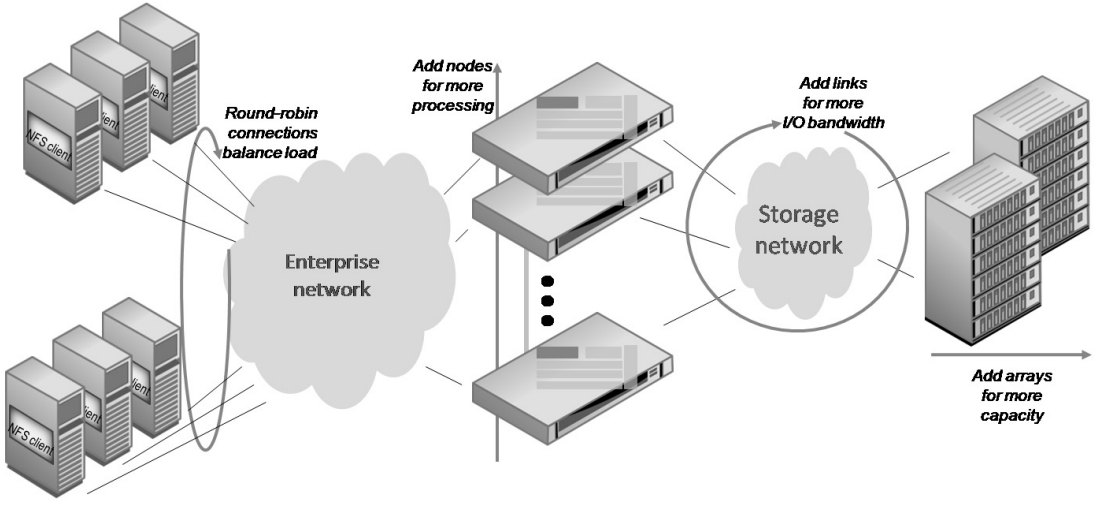


CNFS scalability

All nodes in a CNFS cluster actively serve clients at all times. Clients that use a DNS service to determine a CNFS cluster's IP address receive all nodes' NFS server IP addresses in round-robin order, so that client connections and the load they generate tend to balance evenly among cluster nodes. For situations in which I/O loads and client rosters are static, client administrators can direct traffic to specific CNFS cluster nodes by specifying IP address-based connections.

A single CNFS cluster can scale to as many as 32 nodes, each with multiple gigabit Ethernet interfaces for connecting to clients through its data center network. Nodes can be added to a CNFS cluster while it is actively serving files to NFS clients. The network administrator adds their IP addresses to the cluster's DNS registration, so they immediately begin to participate in load balancing as new clients connect to the cluster.

Figure 3-2 CNFS scaling and load balancing



The server nodes in a CNFS cluster can range from the very economical to the very powerful, in terms of processing, memory, and storage network ports to connect to the back-end storage network.

CNFS clusters also scale in terms of storage capacity, limited only by the maximum capacities of their back-end disk arrays and the storage network connectivity of their component servers. Moreover, CNFS storage configurations are completely flexible—any Fibre Channel, iSCSI, or SAS (for smaller clusters) storage systems can be configured, using any required combination of solid state, high-performance, and high-capacity disk drives. CFS Dynamic Storage Tiering (see Chapter 10 on page 171) automates the

relocation of files between different types of storage as their states and the requirements on them change over time.

CNFS availability

Based on a combination of Storage Foundation availability technology and hardware configuration flexibility, CNFS clusters can be configured for end-to-end high availability of data and file services:

- **Disk arrays.** Most CNFS clusters use enterprise-class disk arrays for back-end storage. To protect against data loss due to disk failure, disk arrays are typically configured to present LUNs based on mirrored or RAID disk sets. Some even offer protection against double disk failures. Typically, arrays can present LUNs on two or more storage network interfaces, to protect against loss of access to storage if a storage network link fails
- **CVM aggregation.** To enhance data availability even further, CVM can mirror two or more LUNs, thus protecting against total failure of a disk array. CVM dynamic multipathing complements disk arrays that support multi-path access, providing simultaneous LUN access on multiple storage network paths for arrays that support that feature, and “active-passive” failover access for less capable arrays
- **Automatic recovery.** All CNFS components, including the NFS cluster service, the Network Lock Manager (NLM), CFS, and CVM, are designed to recover from cluster node failures, and provide continuous service to clients by restarting the necessary services on alternate nodes.

  CNFS high availability is essentially transparent to applications. If a cluster node fails, pending requests time out. NFS client software retries timed out requests continuously until IP address failover is complete. If NLM is in use, client and server jointly recover outstanding locks.

  Because CNFS mounts all NFS-shared file systems on all cluster nodes, any NFS server instance can present any file system to clients. CNFS typically recovers from node failures faster than alternative architectures, because neither volume importing, full file system checking, or file system remounting is typically required, if the Network Lock Manager is in use, the client lock state must be recovered
- **Aggregated network links.** CNFS clusters use their host operating system network stacks to communicate with clients. The number of links, and the ability to *aggregate* two or more into a single high-bandwidth data path are therefore limited only by the ability of cluster nodes’ operating systems and the network infrastructure to support network link aggregation

CNFS load balancing

All nodes in a CNFS cluster actively serve clients at all times. Typically, a data center network administrator registers the CNFS cluster's name with DNS, along with all IP addresses that are bound to NFS server instances in *round-robin* mode.

Clients query DNS to obtain an IP address for connecting to a CNFS cluster. DNS responds to each request with the complete list of CNFS server IP addresses in rotating order. Because clients typically connect to the first IP address in a DNS list, connections, and therefore I/O load, tend to distribute uniformly among cluster nodes. If administrator-controlled load balancing is desirable, client administrators can specify IP addresses in NFS mount commands.

Configuring CNFS for file system sharing

CNFS automates most of NFS file service configuration. A single console command configures CNFS by creating the VCS cluster service group frame work, and specifying a shared file system for use by the Network Lock Manager (NLM) and its CVM volume. Thereafter, administrators can either share existing CFS file systems, or can create, mount, and share a new file system with a single command.

Sharing CNFS file systems with clients

Administrative hint 10

Administrators use the **cfsshare** command with different sub-commands for configuring, sharing, and unsharing CNFS file systems.

A CNFS cluster administrator uses the **cfsshare share** console command to make an already-configured file system accessible to NFS clients. The command causes VCS to issue commands the NFS server to initiate sharing of the indicated file system. NFS servers have several administrator-specifiable options that govern the behavior of shared file system. Because the NFS servers in a CNFS cluster are operating system components, the syntax for specifying these options is platform dependent. The **cfsshare share** command accepts a character string in which NFS mount options are specified in the form required by the platform. It passes the string to the NFS server without verification or modification.

While the syntax and semantics of supported platforms' NFS mount commands vary slightly, the commands and options generally represent common properties of NFS-mounted file systems. Administrators should ensure that the

following options are specified appropriately for the platform's NFS server:

- **Access mode.** File systems can be NFS-mounted for either read-only or read-write access by clients. Sharing a file system in read-only mode overrides individual client computer and user write permissions
- **Synchronicity.** NFS servers can be configured to report write completion to clients while data is still in page cache (*asynchronous*) or to withhold notification until data has been written persistently (*synchronous*)
- **Security restrictions.** Clients can be permitted to connect to an NFS-shared file system via any TCP port (by UNIX convention, TCP ports below 1024 are assumed to be restricted to root users, and therefore secure, whereas connections on higher-numbered ports can requested by any user
- **"Root squashing".** NFS-shared file systems can be configured to grant or deny ("squash") root users of authorized client computers root access to file systems. If root squashing is specified, CNFS replaces root users' file system access permissions with the permissions of an account on the server (usually called **nobody** or **nfsnobody** by default, but alterable to meet data center standards)
- **Write delay.** NFS-shared file systems can be configured to hold data written by clients in cache for a time in the expectation that it will be able to coalesce it with data from subsequent write requests. Delayed writing improves disk I/O efficiency, particularly for file systems whose I/O loads are predominantly sequential
- **Authorizations.** NFS-shared file systems can be made accessible by any client computer, or restricted to specific client computers or client *netgroups* maintained by external NIS or LDAP directory servers. Typically, NFS servers must be configured to use NIS or LDAP, and must be supplied with the IP address of the NIS or LDAP server, or with a server name that it can use to perform a DNS IP address lookup
- **Network transfer size.** NFS servers can be configured to support maximum network data transfer sizes (usually called **rsize** and **wsize**). Clients negotiate network data transfer sizes when they NFS mount file systems. The largest values supported by both client and server become the maximum size for all network transfers

In addition to server mount options, client administrators may either *soft mount* or *hard mount* NFS file systems. Typically, hard-mounting is the default. Timed-out requests to soft mounted file systems cause NFS clients to report errors to applications. With hard-mounted file systems, clients continually retry timed-out requests until they succeed. Applications and users cannot interrupt retries, and so may hang indefinitely, for example if the server has failed. Specifying the **intr** NFS client mount option makes it possible for a user to interrupt retries (for example, by typing CTRL-C), and thus release an application hang

CNFS protocols

Administrative hint 11

For most UNIX and Linux platforms, UDP is the default mount option for NFS-served file systems. Therefore, client administrators must explicitly specify TCP in NFS mount commands.

CNFS supports Version 3 of the NFS protocol[8] (commonly called NFSv3). NFSv3 incorporates the eXternal Data Representation (XDR) standard[9] for platform-independent data representation, so NFS client and server implementations are interoperable regardless of platform type. NFSv3 messages are contained in standard Remote Procedure Call (RPC) protocol messages and transported by either UDP or TCP. In general, TCP is preferable, particularly in WANs and chronically congested networks, because of its flow control and dropped packet handling.

By itself, NFSv3 is *stateless*; servers do not retain information about clients between successive NFS operations. All NFSv3 operations are independent of any previous ones. Statelessness simplifies recovery from server failures compared to other ("stateful") protocols. An NFS server recovering from a fault need not "remember" previous interactions with clients. Clients repeatedly reissue requests that time out because of server failures.

CNFS includes two additional protocols used by many NFS-based applications:

- **Mount.** Clients use the mount protocol to determine which file systems are being shared by an NFS server, and to obtain served file systems' file IDs for use in subsequent NFS calls
- **Lock.** The lock protocol, implemented in CNFS by a cluster-wide *Network Lock Manager* (NLM), enables clients to place advisory locks on entire files and ranges of bytes within files

The CNFS implementation of NLM is a cluster-wide highly available distributed lock manager. Each NLM instance manages the advisory locks for NFS clients connected to its node. If a node fails, the NLM instance on the node that assumes control of its virtual IP addresses interacts with clients to recover their NLM locks. Both failover and NLM lock recovery are functionally transparent to client applications. Typically, when a cluster node fails, its clients retry timed-out NFS requests until IP address failover and NLM lock recovery are complete, at which time they succeed. Client applications that use NLM advisory locking run unaltered when their file systems are served by a CNFS cluster.

8. Defined in RFC 1813. An NFSv4.1 has been published, but is not yet widely implemented or deployed.
9. Defined in RFC 1014.

Network Lock Manager (NLM) advisory file locking

Some UNIX and Linux applications use NLM to synchronize shared access to NFS-served files. NLM locking is *advisory*; that is, applications adhere to it voluntarily. If all applications that access a given file system follow the NLM protocol, simultaneous accesses by multiple clients do not return out-of-date data or cause data corruption. NLM does not protect against applications that access data for which they have been denied locks, or against applications that simply ignore the protocol.

Client computers running applications that use NLM become *stateful*—each NLM server instance maintains a directory in a shared file system. The directory contains a file for each client with outstanding locks. The server holds the actual locks in memory, however.

When a client requests an NLM lock on a file, the NLM instance that handles the request uses CFS's Global Lock Manager (GLM, discussed in Chapter 8 on page 147) to gain control over NLM lock grants to the file. It then queries other instances to determine whether there are conflicting NLM locks, and if there are none, grants the client's request. This guarantees that different NLM instances do not grant conflicting locks to different clients.

CNFS NLM instances are structured as parallel VCS service groups upon which the NFS server virtual IP address (VIP) service groups depend. If a cluster node fails, its VIP service groups fail over to pre-designated failover nodes. Prior to coming online, the failed over VIP service group instructs all NLM instances to enter *grace mode*, in which they only accept lock reclamation requests from clients. While in grace mode, the failed over NLM instance *re-masters* (recreates) the failed node's in-memory lock database by dropping its own locks, merging the failed node's client list with its own, and instructing all clients in the merged list to reclaim their locks. When re-mastering is complete, all NLM instances revert to normal operation.

Since the failed node's virtual IP address is now served by the failover node, which has full access to CFS shared file systems, failover is functionally transparent to clients. Clients access the same file systems via the same IP addresses both before and after failover.

CNFS in a nutshell

The CNFS value proposition of scalability, availability, performance, and flexibility make it an ideal NFS file serving solution in virtually any

environment. Table 3-1 summarizes the what and why of the “CNFS advantage.”

Table 3-1 Why CNFS for NFS file sharing

CFS cluster feature	Comments
High availability	CNFS builds on the time-proven VCS-CVM-CFS stack to deliver highly available NFS file access that scales to thousands of clients
Load balancing	With CNFS round-robin connection management client connections and the I/O load they generate are distributed throughout the cluster. Clients can also connect to specific nodes if that is desirable
Multi-dimensional scaling	Users can grow CNFS clusters independently in the storage capacity (add disk arrays), I/O performance (add storage network or Ethernet interfaces), or processing power (add cluster nodes) dimensions
Stretch clusters	With extended Fibre Channel or iSCSI, cluster nodes can be separated by distances of up to 100 kilometers for disaster protection
Advanced CFS features	Advanced CFS features, notable Dynamic Storage Tiering, Dynamic Multi-Pathing for storage devices, and Reclamation of thinly-provisioned storage are all available in CNFS clusters
Platform flexibility	CNFS clusters can be configured using Solaris (SPARC and x86), AIX, and Linux platforms. They fit nicely into data center vendor management strategies, and are ideal for repurposing replaced equipment
Price/performance	Because users can shop for the most cost-effective computing, storage, and network components, CNFS clusters provide the best user control over the cost of meeting their NFS file serving needs

Part II

Inside CFS: framework and architecture

- The VCS cluster framework
- CVM and CFS in the VCS framework
- Inside CFS: disk layout and space allocation
- Inside CFS: transactions
- Inside CFS: the Global Lock Manager (GLM)
- Inside CFS: I/O request flow
- CFS Differentiator: multi-volume file systems and dynamic storage tiering
- CFS Differentiator: database management system accelerators

Chapter 4

The VCS cluster framework

This chapter includes the following topics:

CFS and CVM are both closely integrated with the *Veritas Cluster Server* (VCS) cluster framework. In conjunction with CVM and VCS, CFS unifies as many as 32 interconnected nodes and their data storage resources into a single system that is:

- **Robust.** VCS automatically detects both application and cluster node failures and restarts ("fails over") applications to pre-designated alternate nodes. Applications configured to fail over are called *failover* or *high availability* applications
- **Scalable.** With VCS, it is possible to run multiple instances of applications concurrently on different cluster nodes. Applications configured for concurrent execution of multiple instances are called *parallel* applications

In the VCS context, CFS instances are a parallel application. As a layer in the application I/O stack (Figure Intro-1 on page 8), CFS instances cooperate to make it possible for business logic and database management system applications to access data in shared file systems, no matter which cluster nodes they are running on. This chapter describes the VCS framework and how CFS fits into it as background for the more detailed descriptions of CFS architecture in subsequent chapters.

VCS components

The main components of the VCS framework are:

- **had.** A *high availability daemon* that runs on each cluster node. **had** monitors cluster node state, and implements pre-defined cluster policies when events require it
- **Communication protocols.** VCS includes two specialized protocols, called LLT and GAB that CFS instances use to intercommunicate on a cluster's private network
- **Agents.** Scripts or executable modules that control the operation and monitor cluster resources (including applications)
- **Configuration files.** Files that define the computer systems, resources, applications, and policies that make up a cluster

The high availability daemon

The VCS high availability daemon (**had**) is the cluster "engine." An instance of **had** runs on each cluster node and dynamically maintains a replicated state machine that provides all nodes with the same view of cluster state at all times.

In addition, each **had** instance monitors the resources connected to its node, and takes appropriate action (for example, initiates application failover) if it detects a critical resource failure.

All **had** instances obtain cluster configuration and policy information from a single *cluster configuration file* normally called **main.cf**. The configuration file specifies cluster resources and their organization into service groups. Optionally, it may also define interdependencies among service groups.

VCS private network protocols

A VCS cluster requires a non-routed Ethernet private network on which its nodes can intercommunicate using a two-layer protocol stack:

- **Low-Level Transport (LLT).** LLT is a high-performing, low-latency replacement for the standard IP stack, used by VCS for all cluster communications. LLT distributes traffic across the cluster nodes performs heartbeating to ensure that all nodes are functioning properly and responsive
- **Group Atomic Broadcast (GAB).** GAB uses LLT as its underlying protocol by which it manages cluster membership and provides reliable communication among nodes. CFS instances use GAB to communicate with each other

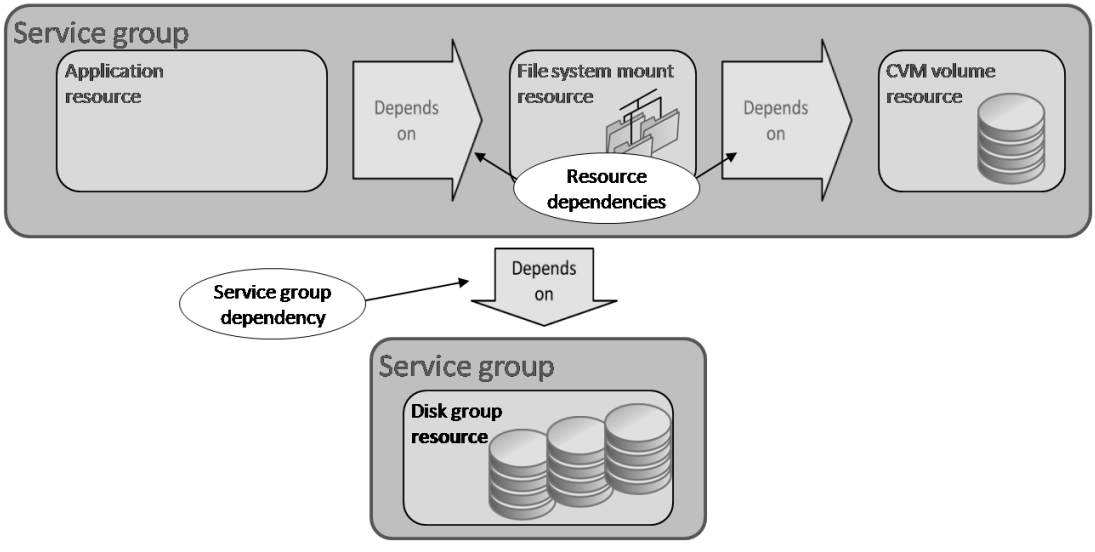
Using GAB and LLT, **had** instances maintain a cluster-wide up-to-date view of the state of all cluster nodes and the applications running on them. CFS instances also use GAB (which in turn uses LLT) to exchange messages, for example, to request delegation of allocation units (Chapter 6). Finally, the Global Lock Manager through which CFS coordinates access to shared file system resources uses GAB as its mechanism for communicating among instances.

The LLT module on each cluster node uses the private network to transmit *heartbeat* messages that help to detect node failures. LLT makes heartbeat information available to GAB, from which nodes' responsiveness or non-responsiveness becomes visible to **had**. **had** uses this information to make various policy decisions, including whether it should *reconfigure* the cluster by adding or removing nodes. During reconfiguration, VCS momentarily freezes any application I/O activity to shared CFS file systems to maintain data consistency.

The VCS service group structure for applications

VCS encapsulates applications and the resources they require to run (disk groups, CVM volumes, CFS file system mounts, network interfaces and IP addresses, and so forth) within logical entities called *service groups*. The VCS framework manages service groups by monitoring their resources while they are operating, and by starting and stopping them in response to changes in cluster state as well as to administrative commands. Figure 4-1 illustrates VCS resource and service groups and dependencies among them.

Figure 4-1 VCS service groups and dependencies



VCS resources

VCS treats everything required for an application to function—disk groups, CVM volumes, CFS file system mounts, IP addresses, network interfaces, databases, and application executable images themselves—as *resources*. Each resource is of a *type* known to VCS through three components:

- **A name.** Designers use resource type names to declare the types of the resource instances they create
- **Attributes.** Some resource attributes are used by VCS; others are supplied as parameters to the resources' own agents
- **An agent.** Agents are executable modules that do what is necessary to monitor and control the operation of resource instances. Each resource type's agent is unique

The agent for a resource type contains four *methods*, either as entry points in its executable image or as scripts:

- **Online.** Executes the actions required to make the resource operational. For example, the online method for the CFS file system mount resource type invokes the operating system **mount** command using as parameters a CVM volume name and a mount option string specified in the resource instance definition

- **Offline.** Executes the actions required to bring the resource to an orderly shutdown. For example, the offline method for CFS file system mounts invokes the operating system **umount** command
- **Monitor.** Executes actions that determine whether the resource is functioning properly. For example, the monitor method for CFS file system mounts calls an API that checks file system status
- **Clean.** Executes the actions required to bring a resource to a known state prior to restarting it after a failure. The clean method for CFS file system mounts performs a forced unmount operation if one is required

Storage Foundation products that contain CFS include VCS agents for file system mounts and other types of resources required for CFS to function.

Resource type definitions

VCS resource types are defined in *type definition files* that contain templates for the resources in text form. By default, a VCS cluster includes a general type definition file called **types.cf** in the directory **/etc/VRTSvcs/conf/config**. The **types.cf** file contains templates for the standard VCS resource types. When a product that contains CVM and CFS is installed, the Common Product Installer adds **CFSTypes.cf** and **CVMTypes.cf** to the **/etc/VRTSvcs/conf/config** directory. If SFRAC is installed, the installer adds **OracleTypes.cf**, and so forth. Resource type definition files are included by reference in a cluster's **main.cf** configuration file, much in the way that C programs often include header files that contain data structure type definitions.

A resource type definition names the resource type and specifies its parameters, including any argument list passed to its agent entry points. For example, Fragment 4-1 illustrates the type definition for the **CFSMount** resource type.

Fragment 4-1 Type definition for the CFSMount resource type

```
type CFSMount (                                                    [01]
  static int RestartLimit = 2                                      [02]
  static str LogLevel                                              [03]
  static str ArgList[] = {MountPoint,BlockDevice,MountOpt}         [04]
  NameRule = resource.MountPoint                                   [05]
  str MountPoint                                                   [06]
  str BlockDevice                                                  [07]
  str MountOpt                                                     [08]
  )                                                                [09]
```

Resource type definitions include specifications for the parameter values supplied when resource instances are created. In the case of the **CFSMount** resource type in Fragment 4-1, these include parameters related to clustering (**RestartLimit**, **LogLevel**), parameters used by the mount agent, and a template for the argument list that VCS passes to the resource type's online method each

time it starts an instance of the type (in this case, mounts a file system).

Resource instances

To configure an application as a VCS service group, a designer specifies its resources in the main.cf file by referring to their type definitions. For example, Fragment 4-2 illustrates the specification of a **CFSMount** resource named **filesystem01**.

Administrative hint 12

Administrators do not usually code **CFSMount** resource definitions directly. They use the **cfsmntadm** console command to add and remove CFS file systems in a VCS cluster configuration.

Fragment 4-2 Specification of a CFSMount resource

```
CFSMount filesystem01 (                                          [01]
   Critical = 0                                                  [02]
   MountPoint = "/mnt01"                                         [03]
   BlockDevice = "/dev/vx/dsk/dg01/disk01"                       [04]
   MountOpt = "blkclear,mincache=closesync"                      [05]
   )                                                             [06]
```

The resource specification attributes identify:

- **Volume.** the CVM volume on which the file system resides (**/dev/vx/dsk/dg01/disk01**)
- **Mount point.** The file system mount point in the hosting node's name space (**/mnt01**)
- **Mount options.** The mount options (**blkclear**, **mincache=closesync**) used by the **CFSMount** online method, which is an operating system mount command

The attributes are specified using names in the **CFSMount** resource type definition.

Organizing and managing resources

VCS manages resources by organizing them into *service groups*. A VCS service group can contain both *critical* and *non-critical* resources. Critical resources must be operational in order for the group to function; non-critical resources need not. For example, CFS can manage many cluster file systems simultaneously. No particular file system (**CFSMount** resource) such as the one specified in Fragment 4-2 is required for the CFS *service* to operate. **CFSMount** resources are therefore non-critical from the point of view of the CFS service group.

While a service group is online, VCS invokes the **monitor** methods of each of its resources periodically to verify that they are functioning properly. If a resource designated as critical to a failover service group fails, VCS stops the service group and restarts it on an alternate node in the cluster. Similarly, if a node running one or more failover service groups fails to issue heartbeat messages on schedule, VCS ejects it from the cluster, causing a reconfiguration. After the reconfiguration, VCS restarts the failover service groups on alternate nodes.

Some of the resources in a service group may depend on other resources. For example, a **CFSMount** resource depends on the CVM disk group in which the volume that holds the file system is contained. Others are independent. For example, the IP address that an NFS server uses to export a file system has no dependency relationship with the file system. (An NFS share resource, however, depends on both file system and IP address.)

VCS starts and stops a service group by invoking the **online** and **offline** methods of its resources in a sequence that is determined by their dependency relationships. Dependent resources are started after the resources on which they depend.

VCS service groups

An application designer can structure a VCS application service group either as:

- **High availability.** (also called failover service groups). VCS automatically restarts high availability service groups on pre-designated alternate nodes if they or the nodes on which they are running fail
- **Parallel.** VCS starts and monitors multiple concurrent instances of parallel service groups on different cluster nodes. VCS does not perform failover of parallel service groups

Both types of service group consist of:

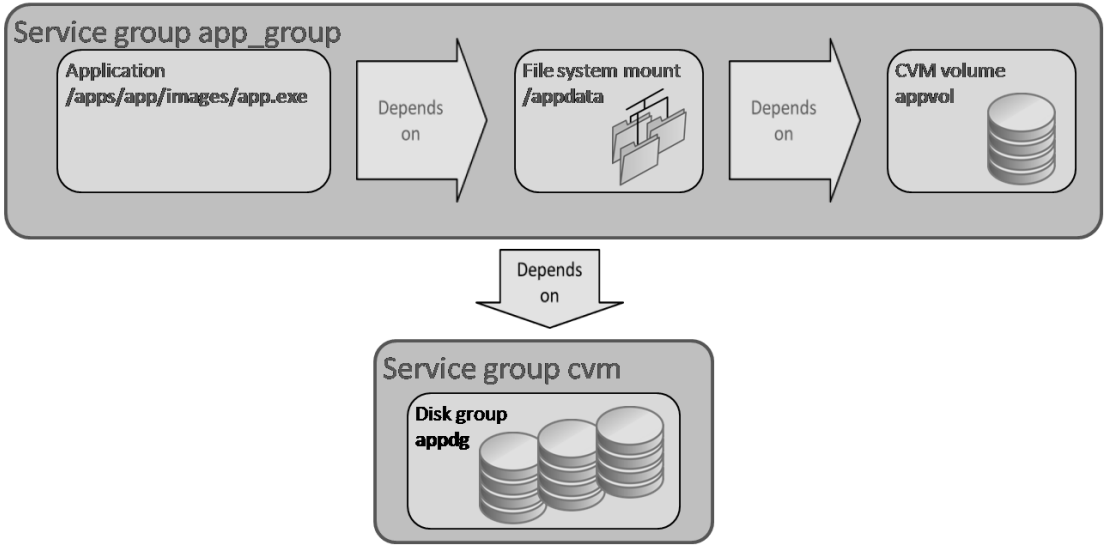
- **Resources.** A dependency tree of the resources that make up the service group
- **Node list.** An ordered list of cluster nodes on which the service is eligible to run
- **Service group interdependencies.** A dependency tree of other service groups on which the service group depends

When starting a failover service, the **had** instance on whichever active cluster node appears first in its node list starts it on that node. **had** starts an instance of a parallel service group simultaneously on every node in the group's eligibility list.

Service group resources

Figure 4-2 represents a simple service group. In this example, an application (**app.exe**) requires a file system mount (**/appdata**), which in turn requires the CVM volume (**appvol**) that holds the file system's data and metadata.

Figure 4-2 Sample VCS service group



Fragment 4-3 defines: is a definition for this service group as it might appear in a cluster's **main.cf** file.

Fragment 4-3 A simple VCS service group definition

```
group app_group (                                                [01]
   SystemList = { node0 = 0, node1 = 1 }                         [02]
   AutoFailOver = 0                                              [03]
   Parallel = 1                                                  [04]
   AutoStartList = { node0, node1 }                              [05]
   )                                                             [06]
   AppType appresource (                                         [07]
       Critical = 0                                              [08]
       Sid @node0 = vrts1                                        [09]
       Sid @node1 = vrts2                                        [10]
       Owner = appowner                                          [11]
       Home = "/apps/app"                                        [12]
       Pfile @node0 = "/apps/app/images/app.exe"                 [13]
       Pfile @node1 = "/apps/app/images/app.exe"                 [14]
       )                                                         [15]
   CFSMount app_mntresource (                                    [16]
       Critical = 0                                              [17]
       MountPoint = "/appdata"                                   [18]
       BlockDevice = "/dev/vx/dsk/appdg/appvol"                  [19]
       )                                                         [20]
   CVMVolDg app_voldgresource (                                  [21]
       CVMDiskGroup = appdg                                      [22]
       CVMVolume = { appvol }                                    [23]
       CVMActivation = sw                                        [24]
       )                                                         [25]
   requires group cvm online local firm                          [26]
   appresource requires app_mntresource                          [27]
   app_mntresource requires app_voldgresource                    [28]
```

Fragment 4-3 defines:

- **Name and type.** The service group name (**app_group**) and type (**parallel**)
- **Run location.** The nodes on which the service is eligible to run (**node0** and **node1**), and on which VCS is to start it automatically (in the specified order)
- **Resources.** The resources that make up the service group (all of which must be accessible to both nodes), including:
  - The application executable image (**appresource**, which specifies the image **app.exe**)
  - The CFS file system mount (**app_mntresource**, which specifies the **/app-data** mount point and the **/dev/vx/dsk/appdg/appvol** CVM volume that contains the file system)
  - The CVM disk group containing the file system's volume (**app_voldgresource**, which specifies the volume **appvol**)
- **Resource dependencies.** The dependencies among the service group's resources:

  - **appresource** (the executable image) requires **app_mntresource** (the file system mount)
  - **app_mntresource** requires **app_voldgresource** (the CVM disk group containing the file system's volume)
- **Service group dependency.** The other service group (**cvm**, the group containing the CVM and CFS underlying support framework) on which this group depends

To start this parallel service group, either automatically when the cluster starts up, or upon administrator command, VCS **had** instances on **node0** and **node1** call the **online** entry points of the **CVMVolDg**, **CFSMount**, and **AppType** agents. Because resource dependencies are specified, the agents are called in sequence.

To stop the group, VCS executes the corresponding **offline** methods in reverse order.

Service groups in operation

While a service group is online, VCS continually monitors each of its resources. As long as all resources' **monitor** methods report operational status, VCS takes no action. If a critical resource fails, VCS makes a designated number of attempts to restart it, and, failing to do so, stops the service group as described in the preceding paragraph.

Thus, VCS monitors state on two levels:

- **Cluster.** The LLT module on each node monitors heartbeat messages from other nodes. If a node's heartbeats from a node cease to arrive, the remaining nodes use GAB protocol services to eject it from the cluster and fail over its high availability service groups according to the policy specified in **main.cf**
- **Service group.** Each node monitors the resources of the service groups it manages. If a critical resource in a service group fails and cannot be restarted, **had** stops the group. **had** restarts high availability service groups on alternate nodes as indicated in the cluster's **main.cf** file. It does not restart parallel service groups, since they are presumed to be running on all eligible nodes

Under most circumstances, service group starting, stopping, and failover are completely automatic. Administrative intervention is only necessary in exceptional cases, such as removal of a node for maintenance. In local and "metropolitan" clusters, VCS completely automates the restoration of service after application, critical resource, or node failure.[10]

10. Through its Global Cluster Option (GCO), VCS supports pairs of widely separated clusters. Global clusters can be configured to require administrative involvement in failover, but for local and metropolitan clusters, automation is the rule.

Service group dependencies

Service groups may depend on each other. For example, the service group specified in Fragment 4-3 depends upon the **cvm** service group (line 26 in the fragment). The service group definitions and interdependencies specified in a cluster's **main.cf** file are effectively the cluster's policy. **had** instances respond to events such as node or application failure by enforcing the resource criticality policy provisions specified in **main.cf**.

Virtual IP addresses for parallel service groups

VCS starts a parallel service group on all cluster nodes that are eligible to run it. For *scale-out* applications in which each client connects to one of several application instances on different nodes, designers typically balance the load among nodes by designating a separate IP address for each application instance, and registering all of them with DNS in "round-robin" mode, so that DNS rotates the order of the list when responding to successive lookup requests

Designers frequently make parallel applications highly available as well by encapsulating the IP addresses that clients use to connect to them in VCS high-availability service groups for which they specify the application instance's service group as a critical dependency. This is called virtualizing IP addresses, and the IP addresses are called *virtual IP addresses* (VIPs).

If an application instance or its server fails, VCS restarts the service group containing the VIP on its failover node. When clients reconnect or retry requests using the VIP, they communicate with the failover server, which is running an instance of the parallel application. The client's messages are directed to that instance, which re-establishes connection and services requests according to its own application-level protocols.

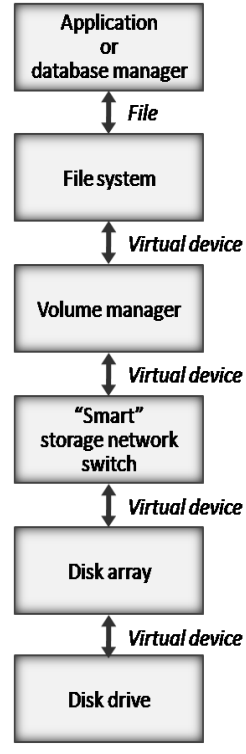
Chapter 5

CVM and CFS in the VCS framework

This chapter includes the following topics:

- Virtual volumes and volume managers
- CVM-CFS synergies
- Putting it all together: CVM and CFS in the VCS environment
- The value of CVM as a CFS underpinning

Today, almost all data center storage is virtualized. Disk arrays, intelligent storage network switches, and *host-based volume managers* that run in application and database servers all coordinate access to groups of rotating and solid-state disks[11] and present clients with disk-like virtual devices with some combination of capacity, resiliency, I/O performance, and flexibility that is superior to that of the component devices. Virtual devices may be partitions, concatenations, mirrors, or stripe or RAID groups of underlying devices. They may be "thinly provisioned," in that physical storage capacity is allocated to them only when clients access the corresponding virtual storage capacity.



Virtual devices may themselves be virtualized. For example, a host-based volume manager can mirror data on two virtual devices that are actually RAID groups presented by different disk arrays; the resulting mirrored virtual device can survive even the failure of an entire disk array. CFS file systems store data in virtual devices called *volumes* that are instantiated by the host-based *Cluster Volume Manager* (CVM). CVM volumes enable all nodes in a cluster to access their underlying storage devices concurrently.

Virtual volumes and volume managers

Whether it is disk array, network, or host-based, a storage virtualizer has two primary functions:

- **Mapping.** Volume managers maintain persistent *maps* that relate volumes' block addresses to block addresses on underlying storage devices. For example, volume managers keep track of sets of mirrored disks, and the stripe order and parity locations of disks in a RAID group. Some volume manager maps are algorithms that relate physical and virtual device block locations. Others, such as maps for thinly provisioned volumes, are

11. In practice, most enterprise storage devices are actually logical units (LUNs) presented by disk arrays, which virtualize the storage capacity of the disk drives they contain. Storage Foundation documentation and this book both refer to disk array LUNs and directly connected disk drives interchangeably as *disks*, except where the context requires more specific reference.

correspondence tables that relate device and volume block numbers to each other

- **I/O request translation.** Volume managers use their disk block maps to translate client I/O requests that refer to volume block addresses into requests to underlying storage devices. For example, for each client write request to a mirrored volume, a volume manager issues write commands addressed to corresponding blocks on each of the volume's mirrored devices

Virtualizers that run in application servers, usually called host-based volume managers, have two properties that make them especially useful in mission-critical cluster environments:

- **Extreme resiliency.** Volume managers can create volumes from physical or virtual devices presented by different disk arrays. This enables them to keep data available to applications even if, for example, a disk array suffers a total meltdown
- **Multi-path access.** Volume managers can improve I/O performance and data availability by transparently managing application access to storage devices connected by two or more network paths

The conventional shared storage device paradigm does not guarantee the execution order of concurrent I/O commands to overlapping disk blocks. For example, if two clients write data to the same disk block at approximately the same time, either request may be executed first. File systems and database management systems enforce ordering requirements among disk reads and writes by regulating access to their own critical data structures.

CVM transforms the read and write requests that CFS addresses to volume blocks into I/O commands that it issues to the underlying disks. CFS preserves file data and metadata integrity by using the Global Lock Manager (GLM, Chapter 8 on page 147) to serialize accesses that might conflict with each other.

CVM volume sharing

CVM presents consistent volume state across a VCS cluster as nodes import and access volumes concurrently. CVM volumes may be *private* (accessible by one node) or *cluster-wide* (accessible by all nodes, also called *shared*). CFS file systems must occupy CVM shared volumes; they cannot use CVM private volumes or raw disks for storage. A CFS file system that occupies shared CVM volumes may be mounted on a single node, a subset of the cluster's nodes, or on all nodes. Table 5-1 summarizes the CFS file system mounting rules.

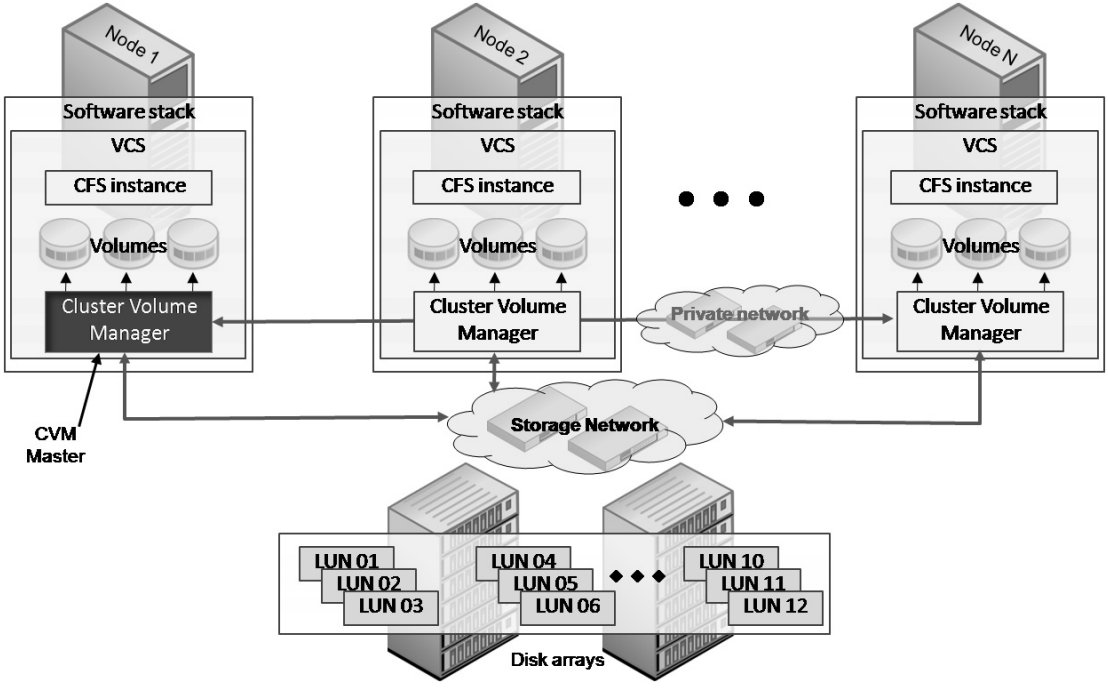
Table 5-1 CFS-CVM mounting rules

CVM volume type ↓	Single-host (Vxfs) mount	Single-node CFS mount	Multi-node CFS mount	Cluster-wide CFS mount
Private	Yes	No	No	No
Cluster-wide	Yes	Yes	Yes	Yes

CVM in the VCS environment

Figure 5-1 illustrates how CVM fits into the VCS environment. As the figure suggests, CVM instances are structured as VCS parallel service groups, with an instance running on each cluster node. All instances access storage devices directly through a storage network using Fibre Channel, iSCSI, or SAS technology. Each CVM instance transforms CFS read and write requests to volumes into I/O operations on the underlying devices, and executes the corresponding commands.

Figure 5-1 Cluster Volume Manager software topology



One CVM instance in a cluster serves as the *Master instance*; others instances are its *slaves*. The CVM Master instance manages disk group and volume configuration, which includes device membership, access paths, and node accessibility. The Master coordinates volume configuration changes, both those directed by administrative command and those resulting from disk or node failure, so that all CVM instances' views of volume state are identical at all times.

Cluster-wide consistent view of virtual volumes

CVM organizes disks into *disk groups* whose membership administrators specify. The disk group is the atomic unit in which CVM instances *import* (gain access to), *deport* (relinquish access to), *activate* (present to CFS) and *deactivate* (withdraw accessibility to) disks. CVM maintains a redundant, persistent record of each disk group's membership, volumes and other underlying structures in dedicated *private regions* of storage on the disks in a disk group.

All CVM instances in a cluster must present the same view of disk group and volume configuration at all times, even in the event of:

- **Storage device failure.** For example, if a disk is added to or removed from a mirrored volume, all CVM instances must effect the change and adjust their I/O algorithms at the same logical instant
- **Cluster node failure.** If a cluster node fails while it is updating one or more mirrored volumes, CVM instances on the surviving nodes must become aware of the failure promptly, so that they can cooperate to restore volume integrity

CVM guarantees that all of its instances in a cluster have the same view of shared volumes at all times, including their names, capacities, access paths and "geometries," and most importantly, *states* (for example, whether the volume is online, the number of operational mirrors, whether mirror resynchronization is in progress, and so forth). A volume's state may change for one of three reasons:

- **Device failure.** A disk that is part of a volume may fail or become unreachable on the network. When this occurs, simple, striped and concatenated volumes fail, and mirrored volumes are at risk
- **Cluster node failure.** If a cluster node fails, the remaining nodes cannot readily determine the state of shared volumes to which the failed node may have been doing I/O
- **Administrative command.** Administrators may disable volumes, add or remove mirrors, increase or decrease capacity, add disks to or remove them from a disk group, and so forth

Whatever the reason for a volume state change, all nodes in the cluster must perceive the change at the same logical instant. When a CVM Master detects or is informed by a slave that a volume's state has changed, it initiates a cluster-wide transaction to process the change. It stores the new volume state persistently in the private regions of the disks that contain the disk group's CVM metadata, marked as a pending change. It then communicates the pending change to slave instances, causing them to initiate a coordinated volume state change transaction. All instances block further I/O to the affected volumes and allow outstanding I/O operations to complete. When all I/O is complete, the Master completes the transaction, making the pending state change the current volume state. Once the transaction is complete, all instances resume I/O to the disk group, adjusting their I/O algorithms as required.

For example, during a cluster reconfiguration that follows a node failure, CVM puts mirrored volumes into a *read-writeback* mode in which every client read is satisfied by reading data from one mirror and writing it to corresponding blocks of all other mirrors. This ensures that the same data is returned, no matter which mirror is used to satisfy a client read request. CVM volumes can be configured with *dirty region logs* (DRLs) that keep track of outstanding writes so that during recovery, only block regions flagged as potentially at risk need to be copied in read-writeback mode. For volumes configured without DRLs, a CVM background thread traverses the entire block spaces in read-writeback mode. CVM distributes responsibility for recovering mirrored volumes after a node

failure among the remaining cluster nodes on a volume-by-volume basis.

If the cluster node on which a CVM Master instance is running fails, the cluster reconfigures. As part of the reconfiguration, a new CVM Master instance is selected and volume states are adjusted as described above. Any IO that requires Master involvement is delayed until the new master has been selected.

CVM-CFS synergies

Several of CVM's advanced features interact synergistically with CFS. The sections that follow list the most important of these features alphabetically and describe how CFS exploits them to enhance file system performance and robustness.

Automatic sparing, disk synchronization, and intelligent copying

Administrative hint 13

Administrators use the **vxedit** command with the **set spare=on** option to designate a disk as a spare for its disk group.

When a disk in a mirrored volume fails, the risk of data loss from subsequent disk failures increases. Consequently, it is desirable to replace the failed disk and restore its content as quickly as possible. CVM's automatic sparing feature allows administrators to pre-designate disks as *spares* to be used as replacements for failed disks in the same disk group. When CVM detects a disk failure, it automatically removes the failed disk from its volume and replaces it with a designated spare from its disk group if a suitable one is available, thus eliminating the human intervention time element from the repair process.

When new or replaced disks are added to a mirrored volume, the new disk must be *synchronized with* (made identical to) the volume by copying volume contents to corresponding blocks on the new device. In most cases, CVM synchronizes volumes while they are "live"—being used by applications. Until a new disk is fully synchronized, CVM cannot use it to satisfy read requests. Write requests must update all disks, however, including the one being synchronized. CVM Master instances guarantee that slave instances perceive the addition of disks to volumes at the same instant, and adjust their read and write access control accordingly. Similarly, when synchronization is complete, a CVM Master coordinates a cluster-wide resumption of normal access control.

Finally, CFS-CVM SmartMove technology minimizes the time and resources required to migrate data between disks. When using SmartMove to copy the

contents of one volume to another, CVM queries CFS to determine which volume blocks are in use, and copies only those blocks. SmartMove obviously saves time whenever data is copied between volumes, but is particularly advantageous when the “disks” underlying the target volume are LUNs in a disk array that supports thin provisioning. Because CVM only writes actual data, the disk array only allocates space for actual data; no physical storage is allocated for unusef file system blocks.

Coordinated volume and file system resizing

Most UNIX file systems are expandable—if a file system requires more space, an administrator can increase the size of the underlying volume, and “grow” the file system to utilize the enlarged capacity. CFS file systems are both expandable *and* shrinkable, for example, to reclaim storage that is no longer required so it can be redeployed. CFS’s ability to reduce the size of a file system, is relatively uncommon, as is its two-dimensional expansion capability:

- **Volume expansion.** An administrator can increase the size of one or more of the CVM volumes a file system occupies, and then grow the file system to utilize the increased space
- **Volume addition.** Alternatively, an administrator can add volumes to a file system’s volume set, tagging the added volumes so that they become part of a storage tier (Chapter 10)

When reducing the size of a file system, CFS first relocates files from volume block locations that lie beyond the reduced size, freeing the storage space to be removed, and then adjusts file system metadata to reflect the reduced capacity. Once a file system’s size has been reduced, the administrator can reduce the size of the underlying volume, and reuse the space freed thereby.

Database management system I/O accelerations

Compared to so-called “raw” block storage devices, CFS files used as storage containers for relational databases offer several advantages, particularly in cluster environments:

- **Administrative simplicity.** Database administrators can easily resize database container files dynamically, while the database manager is using them to satisfy client requests. In contrast to virtual volume resizing, file resizing typically does not require coordination with a storage or system administrator
- **Error-proofing.** When a database uses CFS files as storage containers, the underlying disks are not visible to database administrators and can therefore not be compromised by accident

- **Storage cost-effectiveness.** CFS's Dynamic Storage Tiering feature (Chapter 10 on page 171) can automatically place database container files on the most cost, performance and resiliency-appropriate storage tiers (groups of CVM volumes) for each type of data. For example, active tables can be placed on high-performance storage, while archive logs can be relocated to lower-performing, but reliable storage as they age
- **Data protection flexibility.** Using CFS files as database storage containers allows administrators to employ techniques like space-optimized snapshots of CVM volumes, and file-oriented backup software like Symantec's NetBackup to protect database data

Historically, the main disadvantage to using files as database storage containers has been the impact on performance. In order to isolate clients from each other, most UNIX file systems move data between operating system page cache and application buffers and lock access to files while they are being written. These functions impede database management system performance, and moreover are not necessary, because database management systems carefully synchronize their own I/O requests to avoid potential conflicts.

CFS includes database I/O acceleration mechanisms that overcome these obstacles, so that database management systems can use CFS files as storage containers without incurring performance penalties relative to raw storage devices.

For Oracle, CFS includes a library that implements the Oracle Disk Manager (ODM) API specification. With ODM, the Oracle database management system can perform asynchronous I/O to CFS files and transfer data directly to and from its own buffers. To Oracle database administrators, the advantage of ODM is consistent, predictable, portable database behavior that makes optimal use of the underlying storage infrastructure's capabilities.

For database management systems that do not offer similar APIs, CFS offers a concurrent I/O (CIO) facility. CIO can be activated as a file system mount option. It behaves similarly to ODM in that it enables asynchronous I/O directly to and from a database manager's own buffers. Any application that synchronizes its own I/O requests and manages its own buffers to avoid premature reuse can make use of CIO without modification. Alternatively, applications can activate CIO for specific files by specifying advisories.

Using the CFS database acceleration mechanisms, relational database management systems and other applications that coordinate their own I/O internally can simultaneously achieve both raw device I/O performance and file system ease of administration. CFS database accelerators, enhance database management system I/O performance in three ways:

- **Asynchronous I/O.** Database manager execution threads are able to issue I/O requests and continue executing without waiting for them to complete
- **Direct I/O.** Database manager I/O requests cause data to be transferred directly to and from its own buffers. When a database manager accelerator is

active, CFS does not copy data to or from operating system page cache on its way between database manager and disk storage

- **Write lock avoidance.** Database management system write requests bypass operating systems' file write locking mechanisms, allowing the operating system to pass multiple write requests to a single file through to the I/O stack in parallel

The CFS data caching and file I/O serialization protections are unnecessary with database managers, because they themselves guarantee that they do not issue potentially conflicting I/O commands concurrently, or reuse buffers before I/O is complete.

CFS database accelerators are cluster-aware. Their instances communicate with each other to maintain the structural integrity of database container files and to keep administration simple.

Because CFS is frequently the basis for Oracle database infrastructures, the CFS ODM library includes features that implement other ODM APIs. Three such features are:

- **File descriptor virtualization.** The CFS ODM library saves memory by mapping Oracle's file descriptors to file handles so that each database requires one handle per file shared among all Oracle processes, rather than one per file per Oracle process
- **I/O request consolidation.** ODM "bundles" Oracle's I/O requests and delivers them to the operating system kernel in groups. This minimizes context switches between the Oracle database manager and the operating system that hosts it
- **File management.** The ODM library supports the *Oracle Managed File* capability, which among other features, automatically generates names for the files that Oracle creates, ensuring that they are unique across a cluster

One final feature of the CFS ODM library that is especially significant is that it enables Oracle to *resilver*[12] a mirrored volume after system crash.

When a system with CVM-mirrored volumes fails, it is possible that writes to a volume may have been in progress at the time of the failure. The contents of the disks that make up mirrored volumes may be inconsistent for either of two reasons:

- **Incomplete writes.** A multi-sector write may have been interrupted while in progress. Disks (and disk array LUNs) generally finish writing the last sector sent to them, but not all sectors of a multi-sector write may have been sent. After the failure, a multi-sector Oracle database block may be "torn"—containing partly old and partly new content

12. The *resilvering* metaphor is apt. After a failure that may leave a mirror tarnished, resilvering restores its perfectly reflective quality.

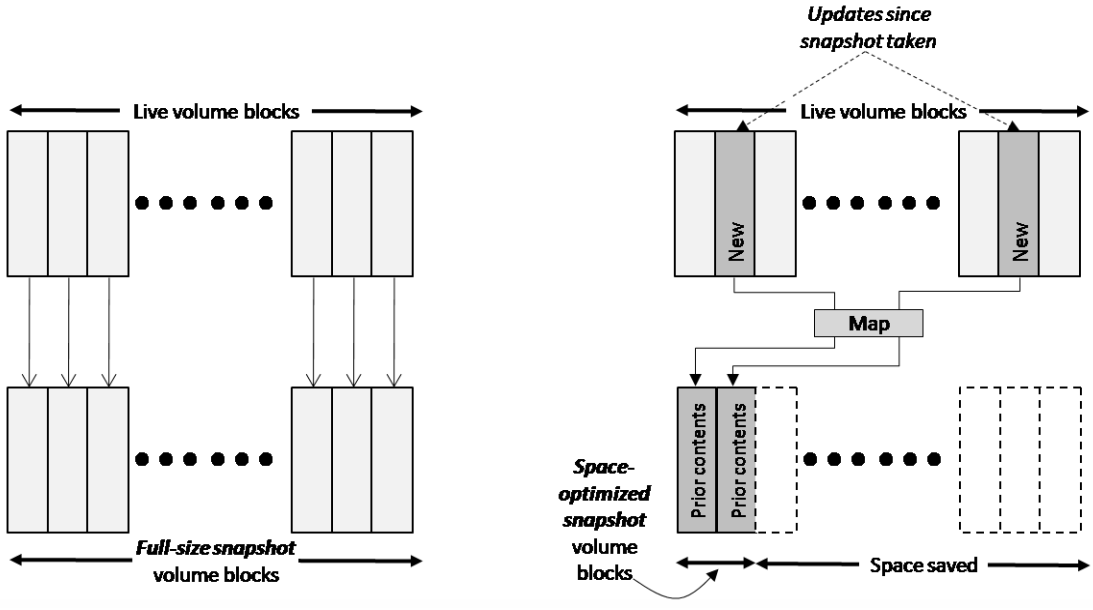
- **Unprocessed writes.** Writes to some of the disks of a mirrored volume may not have been executed at all at the instant of failure. After the failure, all mirrors will contain syntactically valid Oracle blocks, but some mirrors' block contents may be out of date

Oracle maintains leading and trailing checksums on its data blocks to enable it to detect incomplete writes after recovery from a failure. For unprocessed writes, it uses an I/O sequence number called the system control number (SCN) that is stored in multiple locations to detect out-of-date blocks. When Oracle detects either of these conditions in a database block, it uses the ODM APIs to request a re-read of the block from a different mirror of the volume. If the re-read content is verifiable, Oracle uses the ODM API to overwrite the incomplete or out-of-date content in the original mirror, making the block consistent across the volume.

CVM volume snapshots

A CVM *snapshot* of a live volume captures a virtual image of the volume's contents at an instant of time. The image may be a physical block-for-block copy ("full-size"), or it may contain only the pre-snapshot contents of blocks updated since snapshot initiation and a table that relates their locations to their volume block addresses. CVM documentation refers to the latter as "space-optimized" snapshots. Figure 5-2 contrasts full-size and space-optimized snapshots.

Figure 5-2 Full-size and space-optimized snapshots of CVM volumes



Full-size snapshots have certain advantages:

- **Flexibility.** Because they are full images of volumes and the file systems on them, full-size snapshots can be split from their disk groups and taken off-host for processing
- **Performance.** Because a full-size snapshot occupies completely separate storage devices from its parent data set, processing the snapshot has little impact on I/O to the parent data set
- **Failure tolerance.** A full-size snapshot made for data analysis, for example, does double duty in that it can be a recovery mechanism if the parent data set fails

The advantages of full-size snapshots notwithstanding, space-optimized snapshots are attractive because they minimize storage consumption, and because snapshot initiation is nearly instantaneous. The storage space occupied by a space-optimized snapshot is related to the amount of live volume data that changes during the snapshot's lifetime rather than to the size of the live volume.

The first change to any given volume block after a space-optimized snapshot is initiated results in the block's original contents being preserved ("snapped"). Thus, the first write to a block after snapshot initiation consumes more time and I/O resources than subsequent writes.

Full-size snapshots consume more storage space, and take longer to initiate

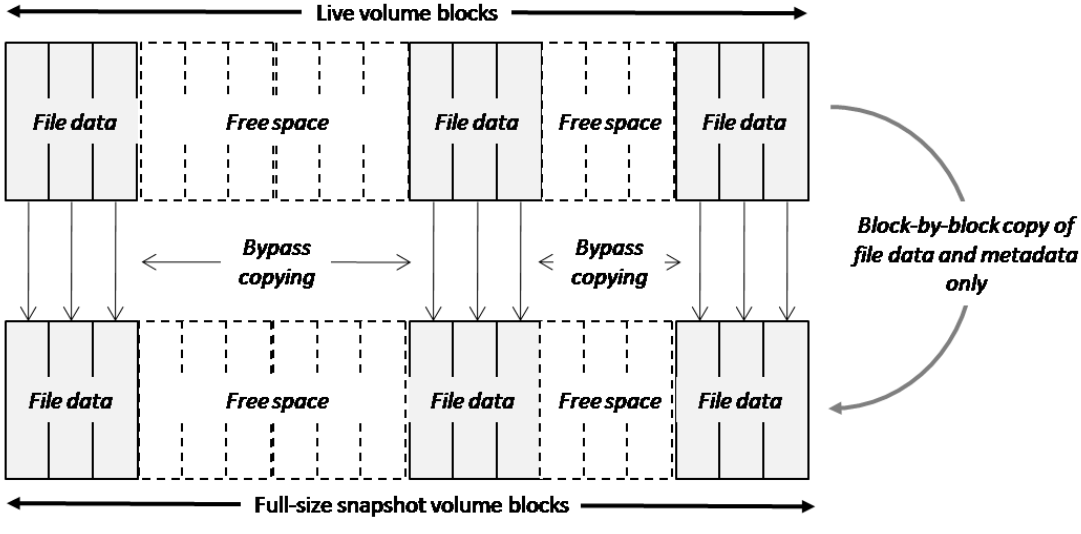
(because the complete contents of the snapped volume must be copied to the snapshot), but they have the advantage of being complete volume images that can be deported for use by other systems, either in or outside of a cluster.

In a conventional full-size snapshot implementation, a volume manager cannot determine which volume blocks are used by the file system and which are free, so it copies all volume blocks when creating a full-size volume snapshot. (This is also true of adding or restoring a disk to a mirrored volume.) Thus, creating a full-size snapshot of a volume that contains a lightly populated file system results in a good deal of useless copying of the contents of unallocated volume blocks.

SmartSync for full-size snapshots of CVM volumes

The Storage Foundation *SmartSync* feature eliminates the copying of meaningless blocks during full-size volume snapshot creation and mirrored volume disk resynchronization. When creating a snapshot or adding a disk to a mirrored volume, CVM makes a SmartSync query to CFS requesting a list of volume block ranges that contain file system data and metadata, and copies only those block ranges, bypassing blocks that CFS regards as unused space. Thus, with SmartSync, the time to create a full-size snapshot or add a disk to a mirrored volume is related to the amount of data that the volume contains, and not to the volume's size.

Figure 5-3 Full-size snapshot creation and mirrored volume synchronization with SmartSync



SmartSync is also useful when migrating from conventional (“thick”) disk array LUNs to thinly provisioned ones. Because SmartSync only copies blocks that are actually in use, only those blocks are written to a target LUN, and therefore only those blocks are allocated physical storage capacity (“provisioned”) by the disk array.

Volume geometry-based I/O optimization

CVM reports the *geometry* of volumes used by CFS. The most relevant volume geometry parameters are:

- **Mirrored volumes.** The number of disks across which data is mirrored
- **Striped volumes.** The number of columns (disks) and stripe unit size

For volumes that are both striped and mirrored, both parameters are relevant.

CFS uses CVM geometry information to optimize space allocation and I/O algorithms. Two important examples of how CFS uses CVM geometry information are:

- **Allocation for small file performance optimization.** If the volume blocks allocated to a small file are split between two columns of a striped volume, an I/O request to the file may result in two I/O commands to two disks. This uses more system resources and takes longer to execute than a single command. To minimize this possibility, CFS uses the volume stripe unit size that CVM reports as one of the inputs to space allocation for small files. If possible, it allocates space for small files at volume block locations that fall into a single column of a striped volume.
- **Sequential read-ahead.** When CFS detects that a file is being read sequentially, it automatically enters *read-ahead* mode in which it pre-reads a certain amount of data in anticipation of upcoming application read requests. When determining how much data to read ahead, CFS takes volume geometry into account. In particular, CFS uses the number of disks in a volume (and for striped volumes, the stripe unit size) to determine the number of concurrent anticipatory reads to schedule. Since each read request results in read commands directed to different disks, the commands can transfer data concurrently, effectively increasing aggregate read-ahead bandwidth.

Administrative hint 14

CFS is aware of CVM volume geometry, but not that of disk arrays. If a disk array LUN is already striped, the incremental benefit of extensive CVM-level read-ahead is likely to be minimal. Administrators can use the **read_pref_io** and **read_nstream** file system tunables to conserve buffers by minimizing read-ahead for such

Putting it all together: CVM and CFS in the VCS environment

As parallel VCS service groups, CFS and CVM are configured similarly, but not identically in the VCS environment. Both are based on “engines,” called **vxfsckd** and **vxconfigd** respectively, that provide their core functionality. Instances of the engines run in all cluster nodes structured as VCS resources.

Each CFS file system is represented as a VCS **CFSMount** resource on each node that mounts the file system. The disk groups that contain the CVM volume(s) on which a file system resides are represented as **CVMVolDg** resources.

Each file system resource has a VCS dependency on the disk group resource that contains its volumes. At least one disk group must depend on the **CVMCluster** executable resource to manage CVM membership and cluster-wide volume state reconfigurations. Figure 5-4 illustrates a typical CVM-CFS resource configuration—an application that serves clients over a network by accessing data in a database. Both application and database failover service groups use CFS file systems based on CVM volumes.

Administrative hint 15

CFS and CVM resource type definitions are contained in the **CFSTypes.cf** and **CVMTypes.cf** files respectively. The Storage Foundation common product installer automatically installs them in directories that allow them to be included by reference in the main.cf file.

The Common Product Installer also creates the **CVMcluster** resource representation in **main.cf** automatically.

Figure 5-4 Typical service group configuration (database application)

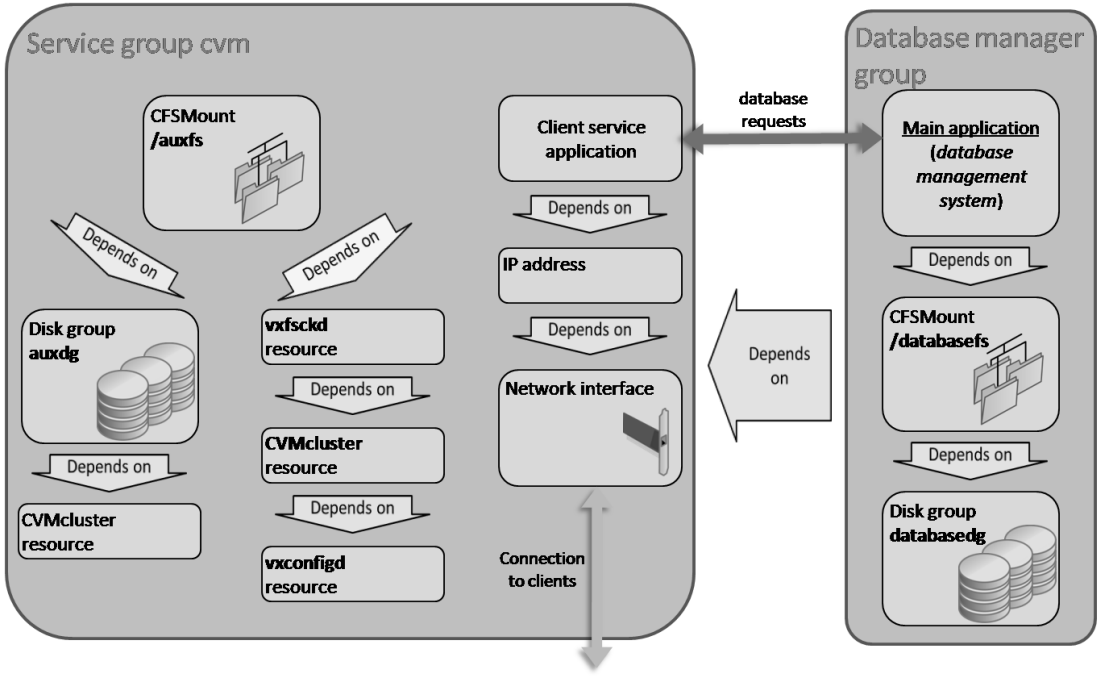


Figure 5-4 represents two service groups: one consisting of the resources required to run the database management system, and the other, the resources that the application requires to interact with clients and make database requests. The database management system service group has a group dependency on the application group, because it requires the **CVMCluster** resource in that group in order to make the volumes that contain the database accessible throughout the cluster.

The database management system service group is structurally similar to the group illustrated in Fragment 4-3 on page 91. The group's application (the database management system executable image) resource depends on the **CFSMount** resource that represents the file system in which the database is stored. The file system resource in turn depends on the **CVMVolDg** resource the represents the disk group containing the volumes used by the file system for storage.

The value of CVM as a CFS underpinning

CFS requires CVM volumes for its storage, even when physical storage is provided by LUNs presented by disk arrays that themselves stripe and mirror data across disks. CVM guarantees CFS a consistent view of its underlying disks and LUNs at all times, including during volume state changes and cluster reconfigurations. Moreover, as the preceding sections illustrate, CVM interacts with CFS to enhance performance, robustness, and administrative simplicity. Finally, with the ODM library included in all CFS product versions, CVM and CFS integrate closely with Oracle database management systems to provide the I/O performance of raw disks along with the robustness and administrative convenience of files.

Chapter 6

Inside CFS: disk layout and space allocation

This chapter includes the following topics:

- UNIX file system disk layout
- The basic CFS disk layout
- Filesets
- CFS space allocation
- Cluster-specific aspects of the CFS data layout

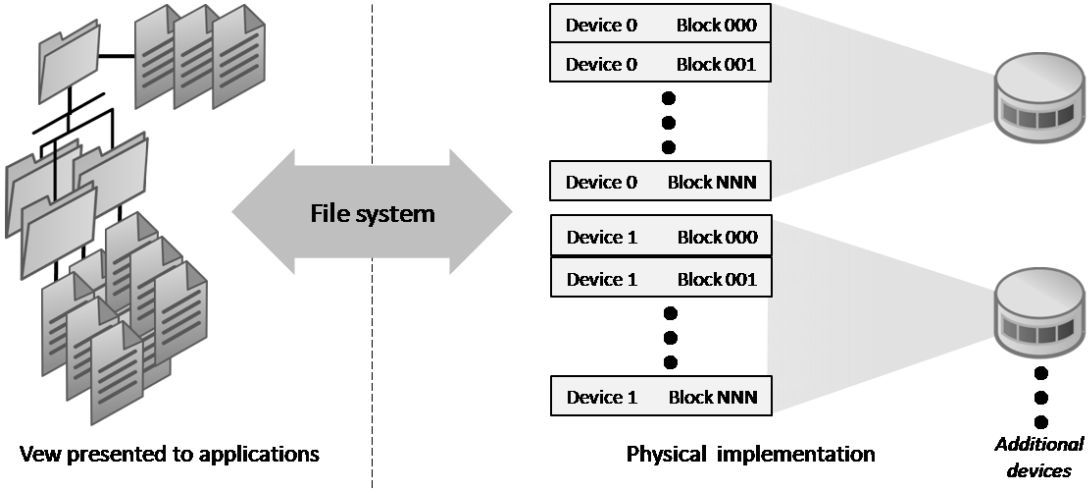
Fundamentally, any UNIX file system[13] manages:

- **A name space.** The names of all data files in the file system and the data and attributes associated with them
- **A pool of storage space.** A set of blocks of storage capacity located on one or more disks. Blocks of storage are constantly being freed or allocated to hold file data or metadata

Figure 6-1 represents the file system abstraction from a UNIX application and user perspective. As the figure suggests, a file system provides applications with the appearance of a dynamic hierarchical tree structure in which files and directories are constantly being created, extended, truncated, and deleted. To do this, the file system uses the much simpler "flat" block address spaces[14] of disk-like logical units (LUNs) as persistent storage.

13. The term *file system* is commonly used to refer to both (a) the body of software that manages one of more block storage spaces and presents the file abstraction to clients, and (b) a block storage space managed by such software and the files it contains. The intended meaning is usually clear from the context.

Figure 6-1 The UNIX file system abstraction



At the enterprise data center level, a file system must be able to execute requests from multiple client concurrently in order to provide acceptable performance. Moreover, it must do this in such a way that each file appears to be:

- **Isolated.** Completely separate from other files so that clients perceive a file independently of any other files in the file system
- **Correct.** Even when multiple clients are accessing, and even updating a file, it appears to be a single ordered stream of bytes

Maintaining a correct mapping between a complex hierarchy of directories and files and one or more flat disk block address spaces becomes even more challenging in a cluster, where the clients manipulating files and directories may be running on different cluster nodes.

A file system manages the pool of disk blocks assigned to it. At any time, some blocks are *allocated,* holding file data or metadata, and the remainder are *free*, available for allocation as needed. The file system must manage its storage correctly and efficiently, even as competing applications request free space for new and extended files and return space to the free pool by deleting files. Again, in a cluster, delivering correct operation and adequate performance are even more challenging, because cooperating file system instances running on multiple nodes must be capable of executing concurrent application requests against the file system. The reliability and high performance of CFS in complex cluster environments stem primarily from two sources:

14. The address space of a logical unit is “flat” in the sense that each block in it is addressed by a single unique number. The block numbers are sequential, starting with 0.

- **Data layout.** CFS has an extremely flexible and robust on-disk layout for data and metadata
- **Transactions.** CFS uses a *distributed transaction* architecture that allows all cluster nodes on which a file system is mounted to perform transactional metadata operations concurrently

Chapter 7 describes the CFS distributed transaction architecture. This chapter provides the background for that material by describing how CFS lays out file system metadata and data on the virtual disks it manages.

UNIX file system disk layout

All UNIX file system on-disk data layouts have a few common components:

- **Starting point**. A well-known address containing the *superblock* that serves as a starting point for locating metadata structures
- **Storage space descriptors**. Persistent metadata that describes the state of the storage space managed by the file system
- **File and file system metadata**. Persistent metadata, including both structures that describe the file system itself, and structures that contain file attributes and locations of the disk blocks that contain file data. In UNIX file systems, the latter structures are usually referred to as *inodes*
- **Directories**. Correspondence tables that relate user-friendly file names to the locations of the inodes that contain the files' metadata and data locations

In addition, server-class file systems require persistent data structures, usually in the form of *logs* or *journals*, that track the status of file system metadata operations. If a server fails while file system metadata operations are in progress, the on-disk representation of metadata may be inconsistent. These structures make it possible to restore a file system's structural integrity.

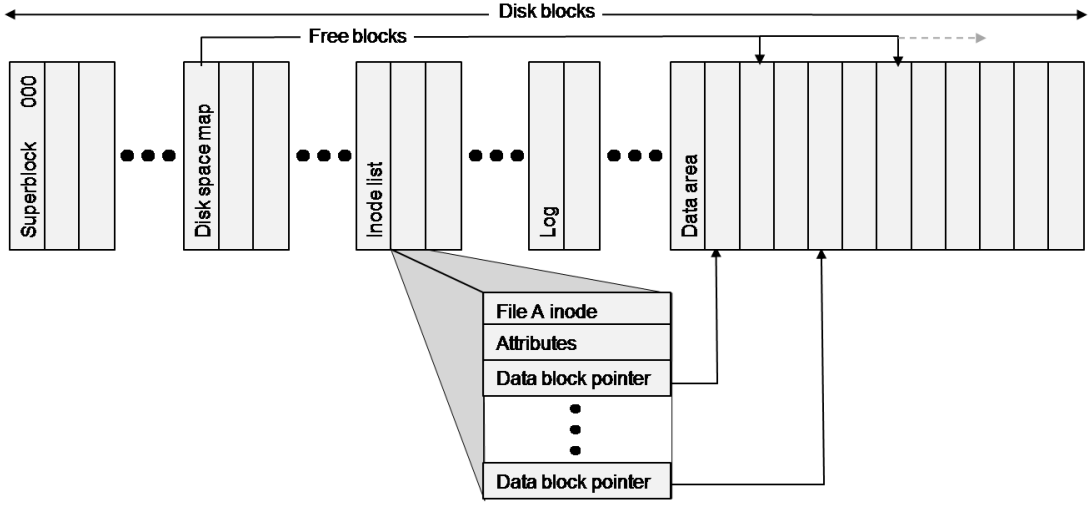
Each type of file system has rigidly defined rules for laying out data and metadata on disk storage. Strict adherence to data layout rules is important because it makes it possible to disconnect a disk containing a file system (data) from one computer and connect it to another that is running the same type of file system (code) supporting the same data layout, and read and write file data correctly. Perhaps more importantly, if one upgrades or even changes an operating system, as long as the new operating system runs a file system (code) that supports the same on-disk data layout as the old one, data in old file systems (data) can be read and written correctly.

Common features of UNIX file system on-disk data layout

Figure 6-2 is a simple representation of on-disk data layout that is typical of UNIX file systems. A file system's *superblock*, the starting point for locating

metadata structures, is found at a fixed location in the file system's disk block address space (for example, **block 000** in Figure 6-2). In most file systems, the superblock is replicated at one or more additional well-known locations so that if the disk block containing the original superblock becomes unreadable, critical file system data structures can still be located. A file system's superblock usually contains the disk block addresses of other key data structures, plus some metadata items that pertain to the entire file system, such as mount status and on-disk data layout version.

Figure 6-2 Typical UNIX file system disk layout[15]



File system disk space management

One of a file system's most important functions is to manage the constantly changing states of the *disk block address space* of storage assigned to it. As clients create, extend, truncate, and delete files, a file system is constantly allocating blocks to files and freeing them for other use. A file system's superblock typically contains the location of a ***disk space map***, in which the file system records the current status of the disk blocks assigned to it.

Different file systems manage disk space in different ways. One common mechanism is a bit map, in which the N^{th} bit represents state of the N^{th} disk block of the file system's overall storage pool. For example, a bit value of '1' might indicate that the corresponding block is available for allocation, while value of '0' would indicate that the block is allocated to a file. With this scheme,

15. Figure 6-2 is a simplified example to illustrate general principles. It is not intended to represent any actual UNIX file system disk layout accurately.

whenever the file system allocated disk blocks to a file, it would clear the corresponding bits in the disk space map. Similarly, whenever it deallocated blocks (for example, executing a client request to delete a file), it would set the corresponding bits in the map.

In practice, file systems use more elaborate techniques to manage the disk space assigned to them. However it is managed, a file system's disk space map must be *persistent*—its state must be preserved across file system shutdowns, power failures, and system crashes, even if they occur while the map is being updated. Moreover, access to the disk space map must be strictly serialized, so that, for example, two concurrent file system execution threads do not allocate the same block to different files.

Identifying and locating files

The superblock is also the starting point for locating the data structure that identifies and locates files and directories in the file system. In UNIX file systems, this structure is called the index node list, or *ilist*, and individual files are described by the index nodes, or *inodes*,) that make up the list. In most file systems, all inodes are of a single fixed size, and are located via information in the superblock.

At any point in time, each inode in the ilist is either *allocated* (in use describing a file) or *free* (available to be used to describe a new file). As Figure 6-2 suggests, each active inode contains:

- **Metadata**. Common information about the file, such as owner, creation time, time of last update, access permissions for other users, and so forth
- **File data location(s)**. One or more descriptors that identify the disk blocks that hold the file's data

The format of disk block descriptors differs from file system to file system. In some, each descriptor represents a fixed number of contiguous disk blocks by pointing to the first of them. Others, including CFS, use richer structures that describe *data extents,* ranges of consecutively numbered file system blocks of varying length. Variable-length extents make it possible to describe large files very concisely, leading to more efficient space allocation and better application I/O performance.

Write caching and file system logs

UNIX file systems improve their performance by holding data and metadata updated by clients in a *volatile* (non-persistent) main memory cache for some time after signaling to clients that their updates have been completed. They write updates to persistent disk storage either on application command (for example, the POSIX **fsync()** API), or "lazily," either as I/O resources become available, at regular intervals, or a combination of the two. Write caching

improves application performance, because applications do not wait for disk writes to complete before progressing. But there is an undesirable side effect: if a system crashes before the file system persists a metadata update that the application perceives as complete, and may therefore have acted upon, the update is "lost"—not reflected anywhere in the file system when the system restarts.

Some file systems, including CFS, adopt a compromise strategy to guard against lost updates by persistently logging their *intent* to update data and metadata before actually doing so. File system logs are typically small (proportional to the number of operations likely to be in progress at a time rather than to the size of the file system); and if stored on high-performance devices such as high-RPM or solid-state disks, can be updated much faster than file data and metadata. Typically, file system logs are stored in the file system's own disk block address space; their locations are found either directly or indirectly by referring to the superblock.

File data storage

The majority of the storage capacity managed by a file system is used to hold file data. UNIX file systems manage this storage in fixed-size blocks that are an integer multiple of disk sector sizes. Blocks of storage in this part of the space are constantly changing status from free space to being assigned to files and the reverse. A file system's single most important function is maintaining the integrity and consistency of the data structures that describe the files it contains.

The basic CFS disk layout

The CFS disk layout[16] includes all of the major elements of file system metadata described in the preceding section. It is unique, however, in one key respect:

All metadata structures in a CFS file system are stored in files.

The concept of organizing file system metadata as files is an extremely powerful one, leading directly to much of CFS's flexibility and extensibility. For example, one way for an administrator to increase or decrease a CFS file system's storage space is to add CVM volumes to or remove them from its assigned storage complement. This is possible because the volumes that a CFS file system manages are described in a file, which is extended when a volume is added, and

16. The disk layout of CFS, and of the single-host VxFS file system on which it is based, has undergone significant evolution since the file system was first shipped in 1992. This section describes the current disk layout, some of whose features are not present in earlier versions, particularly those that preceded disk layout version 4.

contracted when one is removed.

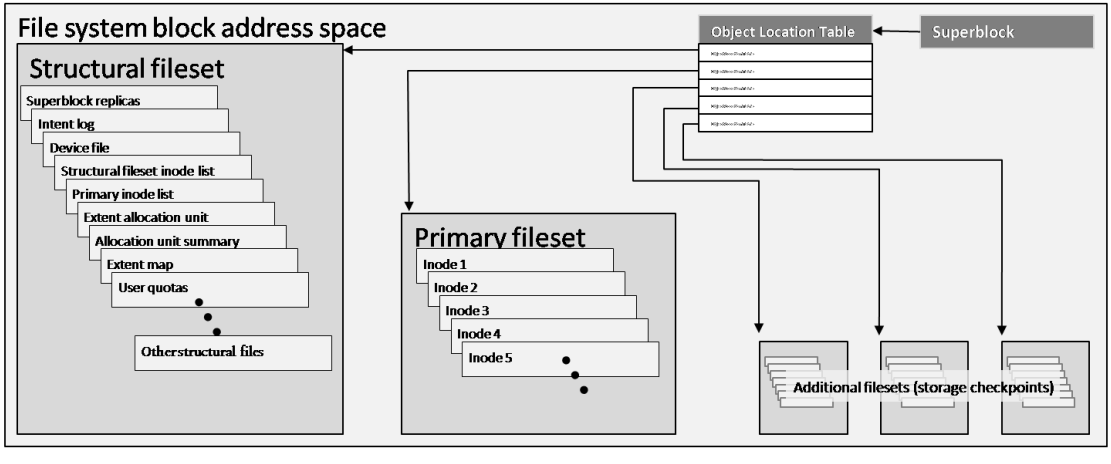
Filesets

The metadata and data in a CFS file system are organized as *filesets*, that can be regarded as "file systems within a file system." The fileset concept allows disjoint groups of files used by different entities to share the storage capacity of a volume or VSET. At a minimum, each CFS file system contains two filesets:

- **The structural fileset**. Files that contain file system metadata. CFS does not expose the structural fileset to administrators or users, although administrators can examine and manipulate some of its contents indirectly
- **The primary fileset**. Files that contain user data and the metadata that describes them. The primary fileset is the user's view of a CFS file system

A CFS file system may contain additional filesets. Each *Storage Checkpoint* (snapshot or clone) of a file system is represented by a fileset. Figure 6-3 illustrates the CFS fileset concept.

Figure 6-3 CFS filesets



The CFS structural fileset

The starting point for navigation of a CFS file system is its *superblock*. The superblock of a CFS file system consists of a single disk sector divided into read-only and read-write areas. The read-only portion holds invariant information defined when the file system is created. Because the superblock is so fundamental to a file system's structure, CFS replicates it multiple times during the life of a file system so that if a main superblock becomes corrupted or

unreadable, the file system checking utility (**fsck**) can locate a replica and use it during the course of restoring file system structural integrity.

One important item in the read-only area of superblock is a pointer to a replicated structure called the Object Location Table (OLT). The OLT is the master list of locations of structural files that contain CFS metadata. Some structural files are *instance-specific*—each CFS instance has a private version of the file. A per-node object location table (PNOLT) structural file has a record for each node in the cluster that contains the locations of the node's instance-specific structural files. Per-node structural files and their advantages are discussed on page 135.

CFS replicates the inodes of several especially critical structural file types. For example:

- **Inode list**[17]. The primary fileset's inode list
- **Extent bitmaps**. The storage space bit map files (one per device managed by the file system)
- **Intent log**. The file system instance's intent log.

CFS stores replicated inodes in different disk sectors so that an unreadable disk sector does not result in loss of critical file system structural data. During updates, it keeps these files' replicated inodes in synchronization with each other.

Using files to hold metadata makes CFS flexible and space-efficient, and at the same time enhances the performance of certain operations. For example, when a conventional UNIX file system is created, it typically reserves an inode list consisting of a sequential array of disk blocks whose size is proportional to the file system's size. Once reserved, the inode list's size cannot be changed. This is undesirable for two reasons:

- **Inflexibility**. It places a fixed limit on the number of files a file system may contain
- **Waste**. If a file system contains only a few large files, most of the space reserved for inodes is wasted

In contrast, the inode lists for both structural and primary filesets in a CFS file system are themselves files. When an administrator creates a file system, CFS initially allocates inode lists with default sizes. CFS automatically increases the size of inode list files as necessary when adding files and extents to the file system. Thus, the limit of one billion files in a CFS file system is based on the maximum practical time for full file system checking (**fsck**), and not on the amount of space assigned to it.

A CFS structural fileset contains about 20 types of files that hold various types

17. CFS structural file types are identified by acronymic names beginning with the letters "IF."

of metadata. Table 6-1 lists the subset of structural file types that relate to the most user-visible aspects of a CFS file system, and the advantages of using structural files for metadata as compared to more conventional file system designs.

Table 6-1 CFS structural files (representative sample)

Structural file type	Contents	Advantages over conventional file system structures
Label file	Locations of OLT and superblock replicas	OLT allows for flexible metadata expansion Replicated superblocks are resilient to disk failure
Intent log (replicated inodes)	Circular log of file system transactions in progress	Enables administrator to control intent log size as file system size or transaction intensity increases
Device file (replicated inodes)	Identities and storage tiers of file system volumes	Makes it possible to add and remove storage volumes Enables Dynamic Storage Tiering (Chapter 10 on page 171)
inode list (replicated inodes)	List of inodes that contain metadata and on-disk locations for user files	Decouples the maximum number of files in a file system from file system storage capacity
Attribute inode list (replicated inodes)	List of inodes hold hold extended file attributes	Matches space occupied by extended attribute inodes to actual number of extended attributes in a file system Conserves space occupied by extended attributes
User quota	List of limits on users' storage consumption	Minimizes storage space consumed by quota structures Enables cluster-wide quota enforcement

Structural files for space management

In addition to the structural files listed in Table 6-1, CFS uses three structural files to manage *allocation units*, the structures it uses to manage the storage space assigned to a file system. Table 6-2 lists the three structural files, all of which have replicated metadata. Collectively, the three describe the state of a

file system's allocation units and the file system blocks they contain.

Table 6-2 CFS structural files for managing free space

Structural file type	Contents	Advantages over conventional file system structures
Allocation unit state (IFEAU)	Overall allocation unit state	Instantly determine whether an allocation unit is completely free, completely allocated, or partially allocated
Allocation unit summary (IFAUS)	Number of extents of various sizes available in each allocation unit	Quickly determine whether an extent of a given size can be allocated from a given allocation unit
Extent map (IFEMP)	Detailed map of available storage in each allocation unit	Fast allocation of optimal size extents (Usually referred to as "EMAP")

Using structural files to hold space management metadata structures has two main advantages:

- **Compactness**. CFS can describe very large contiguous block ranges allocated to files very concisely (in principle, up to 2^{56} file system blocks with a single extent descriptor)
- **Locality**. It localizes information about free space, thereby minimizing disk seeking when CFS allocates space for new or extended files

CFS space allocation

Ideally, file system space allocation should be efficient in three dimensions:

- **Computation**. Allocating and freeing storage space should require the least possible amount of computation and I/O
- **Data structures**. The data structures used to track allocated and free space should be robust and rapidly searchable
- **Utilization**. Available space should be allocated optimally, with minimal fragmentation

CFS space allocation incorporates two concepts that make it particularly efficient, both for file systems containing large numbers of files and file systems that host a few very large files:

- **Allocation units**. The space occupied by a CFS file system is divided into a number of *allocation units*, each containing 32,768 of file system blocks. The Extent Map structural file represents the state of the file system blocks in each allocation unit using a multi-level bitmap that makes searching fast and

efficient when CFS is allocating space for files. To further speed searching, each allocation unit's record in the Allocation Unit Summary structural file lists the number of free extents of various sizes it contains. Finally, the Extent Allocation Unit Summary file expresses the overall state of each allocation unit (completely free, completely allocated, or partly allocated).

- **Variable-size extents**. The addresses of file system blocks allocated to files are contained in *extent descriptors* stored in the files' inodes. In principle, a single extent descriptor can describe a range of as many as 2^{56} consecutively located file system blocks. Thus, as long as contiguous free space is available to a file system, even multi-gigabyte files can be represented very compactly

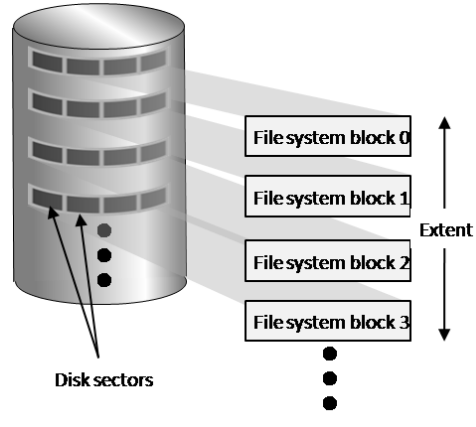
Administrative hint 16

Smaller file system block sizes are generally preferable for file systems that will contain smaller files. Larger file system block sizes reduce the number of extents required to map large files.

File system blocks and extents

CFS treats the disk storage space on each volume assigned to it as a consecutively numbered set of *file system blocks*. Each file system block consists of a fixed number of consecutively numbered disk sectors of 512 bytes. When creating a file system, an administrator specifies its *file system block size*, which remains fixed throughout the file system's life. CFS supports file system block sizes of 1,024, 2,048, 4,096, of 8,192 bytes (2, 4, 8, and 16 512 byte disk sectors respectively).

Figure 6-4 CFS file system blocks and extents stack



The file system block is the smallest unit in which CFS allocates disk space to files. Thus, a one-byte file occupies one file system block of disk space. Smaller file system block sizes are therefore generally more suitable for file systems that are expected to contain smaller files because they "waste" less space (space allocated to a file, but not containing file data). Conversely, larger file system block sizes are more appropriate for file systems that are expected to contain larger files because they describe large files more concisely.

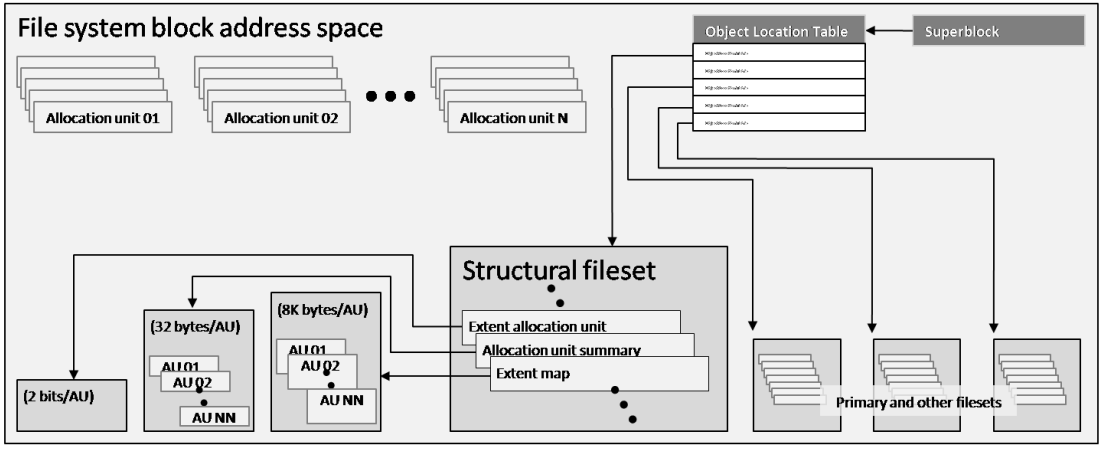
CFS refers to a range of consecutively numbered file system blocks described by

a single descriptor as an *extent*. Space and file management data structures describe the locations and sizes of extents of free and allocated storage space.

Allocation units

To manage free storage space efficiently, CFS organizes the space on each volume assigned to a file system into *allocation units.* Each allocation unit contains 32,768 consecutively numbered file system blocks.[18] CFS tracks the space on each device managed by a file system using three structural files listed in Table 6-2 and represented in Figure 6-5. Because these structural files are typically cached, allocating and freeing space is fast and efficient.

Figure 6-5 CFS free space management



Extent Allocation Unit structural files

For each storage device that it manages, a CFS file system maintains an *Allocation Unit State* structural file that indicates one of four states for each of the device's allocation units:

- **Completely unallocated**. All file system blocks in the allocation unit are free
- **Completely allocated**. The entire allocation unit is allocated to a single extent
- **Expanded**. Blocks in the allocation unit have been allocated to multiple extents (some blocks may be free)

18. If the size of a volume assigned to a file system is not a multiple of 32,768 file system blocks, its last (highest numbered) allocation unit contains fewer file system blocks.

- **"Dirty"**. The allocation unit has been subject to recent changes that may not yet be reflected in the Allocation Unit Summary structural file (the Extent Map file and its cached image are always accurate)

Thus for example, when attempting to allocate 32,768 file system blocks or more in a single operation, CFS can determine immediately from Extent Allocation Unit files which, if any, allocation units can be allocated in their entirety.

Allocation Unit Summary structural files

Allocation Unit Summary structural files give a more detailed picture of allocation unit state. For each allocation unit on a device, this file includes a record that indicates how many free extents of 1, 2, 4, 8,...16,384 file system blocks the allocation unit contains. When allocating space, CFS attempts to use an extent whose size is the smallest power of two larger than the required amount (for example, 1,000 file system blocks can be allocated from a 1,024-block free extent). Cached Allocation Unit Summary records allow CFS to quickly determine whether an extent of a given size can be allocated from a particular allocation unit.

When CFS allocates storage space, or frees space by deleting or truncating a file, it marks the affected allocation unit's state "dirty" in the Extent Allocation Unit file. CFS disregards Allocation Unit Summary information for dirty allocation units, and refers directly to extent maps. A background execution thread updates Allocation Unit Summary files to reflect the actual number of free extents of various sizes in each allocation unit.

Extent MAP structural files

Each Extent Map structural file contains a record corresponding to each allocation unit on the device it represents. An allocation unit's Extent Map record contains a set of bit maps that enable CFS to quickly determine which sequences of 1, 2, 4,...2,048 file system blocks within the allocation unit are free. Extent maps allows CFS to quickly locate the largest available extent that can contain the amount of space it is attempting to allocate.

Extent maps also make de-allocation of storage space fast and efficient. To free an extent, CFS updates the Extent Map for its allocation unit. In addition, it marks the allocation unit "dirty" in its Extent Allocation Unit file so that subsequent allocations will ignore its Allocation Unit Summary records. A CFS background thread eventually updates Allocation Unit Summary records for "dirty" allocation units to reflect the correct number of free extents of each size.

Because CFS can extend the size of its space management metadata files as necessary, it is easy to add storage to a CFS file system, either by increasing the size of its volumes to add more allocation units or by assigning additional volumes to it. Moreover, because a CFS file system's volumes retain their individual identities, it is possible to relocate files between storage tiers

transparently, as described in Chapter 10 on page 171.

CFS storage allocation algorithms are “thin-friendly” in that they tend to favor reuse of storage blocks over previously unused blocks when allocating storage for new and appended files. With CFS, thin provisioning disk arrays that allocate physical storage blocks to LUNs only when data is written to the blocks, need not allocate additional storage capacity because previously allocated capacity can be reused.

Inside CFS Extents

A set of consecutively numbered CFS file system blocks is called an *extent*. Three pieces of information describe the range of file system blocks that make up a CFS extent:

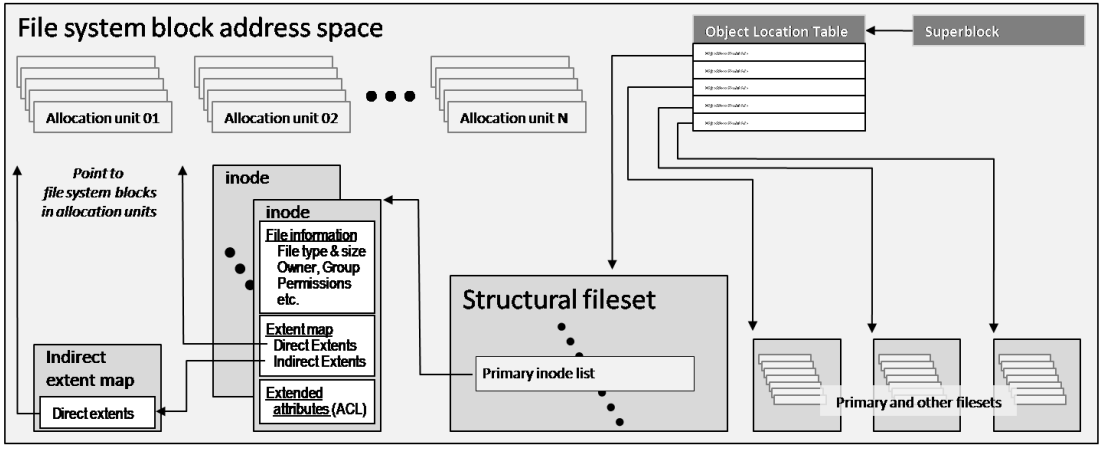
- **Volume identifier**. An index to the volume that contains the extent
- **Starting location**. The starting file system block number within the volume
- **Size**. The number of file system blocks in the extent

An extent may be *free* (not allocated to a file), or it may be allocated (part of a file’s block address space). Free extents are identified by the structures listed in Table 6-2 on page 122.

Using extents to describe file data locations

Extents allocated to files are described by *extent descriptors* that are either located in or pointed to by the files’ inodes. CFS inodes for files in the primary fileset are located in a structural file called an Inode List File. Figure 6-6 illustrates the structure of CFS inodes.

Figure 6-6 CFS primary fileset inode list



When it creates a file, CFS assigns an inode to it. File system inodes are 256 bytes in size by default. An administrator can specify a 512-byte inode size when creating a file system; this may be useful for file systems in which a large percentage of the files have non-inherited access control lists. (If two or more files inherit an ACL from the directory in which they reside, CFS links their inodes to a single copy of the ACL contents, which it stores in blocks allocated from an *Attribute Inode List* structural file). Whichever inode size is selected at file system creation time, inodes are numbered according to their positions in the inode list. A file in the primary fileset is associated with the same inode number throughout its lifetime. When a file is deleted, CFS marks its inode as available for reuse unless it qualifies for aging (see page 221).

Administrative hint 17

CFS holds inodes for active files in a dedicated inode cache in main memory. It computes the size of the cache based on system memory size. The *Veritas™ File System Administrator's Guide* for each supported platform describes how administrators can adjust a tunable to force a larger or smaller inode cache size. number of extents required to map large files.

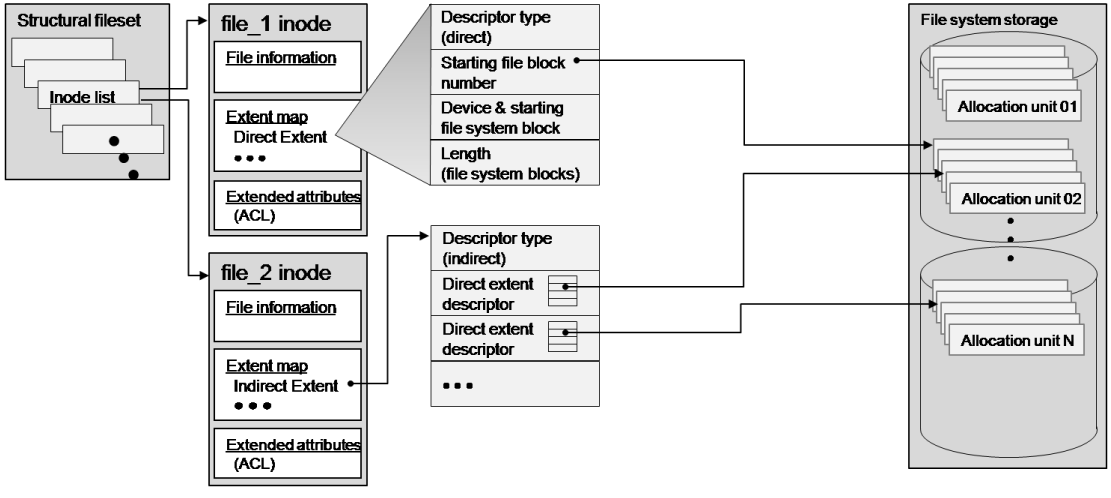
Each active inode contains a file's *attributes*—file-related metadata items, some defined by the POSIX standard, and others specific to CFS. These include the file's name, size, owner, access permissions, reserved space, and so forth. In addition, an inode may contain *extended attributes* assigned by CFS. CFS stores access control list (ACL) entries that specify individual users' file access rights as extended attributes. If a file's ACL is too large to fit in its inode, CFS stores additional entries in one or more inodes which it allocates from the *Attribute Inode List* structural file (separate from the inode list), and links to the file's primary inode.

Finally, each file's inode contains a *block map* (BMAP) that specifies the locations of the file system blocks that contain the file's data. Data locations are specified by a series of *extent descriptors*, each of which describes a single range of consecutively numbered file system blocks. CFS includes two types of extent descriptors:

- **Direct extent descriptor**. Direct extent descriptors point directly to the locations of file system blocks that contain file data
- **Indirect extent map pointer**. Indirect extent map pointers point to blocks of storage that contain additional direct extent descriptors and, if required, further indirect extent map pointers. CFS allocates space and creates an indirect extent map when a file is extended and there is insufficient space for more direct extent descriptors in its primary inode or in an already-existing indirect extent map

Figure 6-7 illustrates the structure of CFS inodes containing both direct and indirect extent descriptors.

Figure 6-7 Structure of a CFS inode



When the number of extents allocated to a file exceeds the number of descriptors that fit in an inode, CFS moves the file's direct extent descriptors to an indirect extent descriptor and uses the indirect descriptor structure for further allocations.

File block addresses and file system block addresses

The space that CFS allocates to a file may consist of a single extent or of multiple non-contiguous extents. Non-contiguous extents occur for two primary reasons:

- **File system occupancy**. When a file is pre-allocated or first written, the file system may not have a single extent or contiguous group of extents large enough to contain the entire file
- **Appending**. When data is appended to a file, or adjacent file blocks are added to a sparse file, the space adjacent to the file's original file system blocks may no longer be available

Like all UNIX file systems, CFS represents each file to applications as a single stream of consecutively numbered bytes. The bytes are stored in (possibly non-contiguous) extents of file system blocks allocated to the file. Collectively, the file system blocks allocated to a file constitute an ordered *file block address space*.

In order to represent the file block address space, each extent allocated to a file has one additional property: the starting file block number represented by the starting file system block number of the extent's location. CFS maintains the correspondence, or *mapping*, between consecutively numbered file blocks and extents of file system blocks. Figure 6-8 illustrates CFS file data mapping for a file mapped entirely by direct extent descriptors.

Figure 6-8 CFS file data mapping

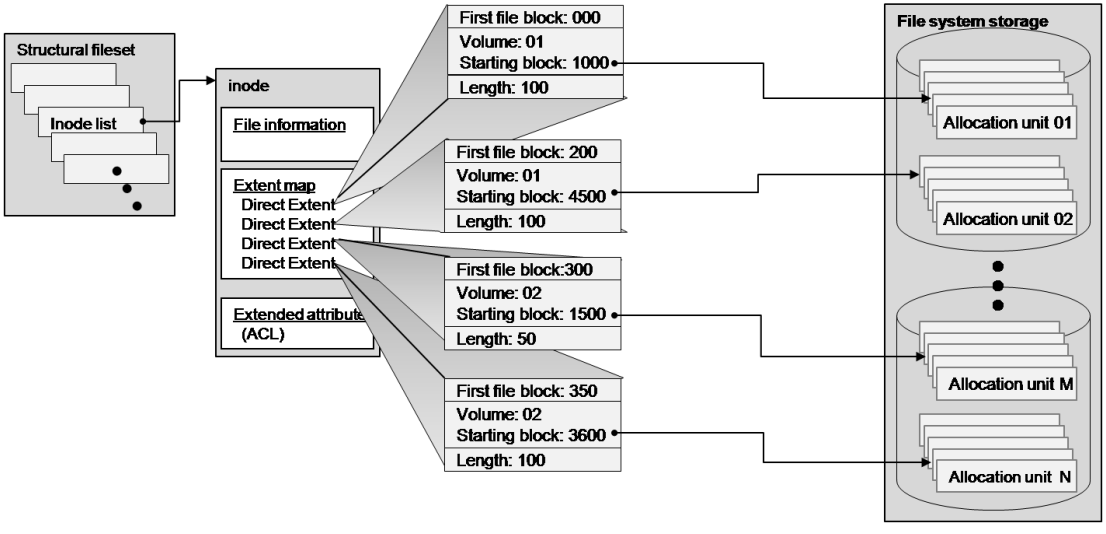


Figure 6-8 represents a file whose data is stored in four non-contiguous extents on two volumes of the file's volume set, and in addition, which contains a *hole* (file block addresses 100-199) to which no storage is allocated. Table 6-3 summarizes the mapping of the 450 file blocks containing the file's data to file system blocks.

Table 6-3 Mapping file blocks to file system blocks

File blocks	Volume containing extent	Starting file system block of extent
0-99	Volume 01	File system block 1,000 of Volume 01
100-199	None	No storage allocated (hole indicated by first file block of second extent)
200-299	Volume 01	File system block 4,500
300-349	Volume 02	File system block 1,500
350-449	Volume 02	File system block 3,600

This simple example illustrates three key points about CFS file data mapping:

- **Multi-volume storage**. The extents that hold a file's data can be located on different CVM volumes. Collectively, the volumes assigned to a file system are called its *volume set* (VSET)

- **Variable-size extents**. File data extents can be of any size. (They are limited only by the 2^{56} file system block maximum imposed by the size of the largest extent descriptor length field)
- **Unlimited number of extents**. There is no limit to the number of extents in which a file's data is stored (although fewer extents results in better data access performance). A CFS inode has space for up to ten extent descriptors depending on its format. When these have been used, CFS allocates an 8 kilobyte block (or smaller, if no 8-kilobyte block is available) in which it creates an indirect extent map, moves the file's extent descriptors to it, and links it to the file's inode. Indirect extent map structures can be cascaded as long as there is space in the file system

Cluster-specific aspects of the CFS data layout

CFS instances running on multiple cluster nodes execute multiple client requests concurrently, some of which require modification of structural file contents. Access to the data structures that CFS uses to manage a file system must be strictly serialized, for example, so that file system blocks do not become "lost" (neither free nor allocated to a file), or worse, allocated to two or more files at the same time. If one CFS instance is updating a Free Extent Map record because it is allocating or freeing file system blocks, other instances must not access the map until it has been completely updated.

File system resource control

Computer systems typically use some sort of locking mechanism to serialize access to resources. A program locks access to a resource by making a request to a central authority called a lock manager that grants or withholds control resource based on the state of other petitioners' requests. For example, a file system execution thread that is allocating space for a file must obtain exclusive access to the data structures affected by the allocation so that other threads' actions do not result in lost or multiply-allocated blocks. A lock manager denies a request for exclusive access if the data structures are already locked by other threads, and grants it otherwise. Once a lock manager has granted exclusive access to a resource, it denies all other requests until the grantee relinquishes its exclusive access permission.

CFS resource management

Within a single computer, the 'messages' that flow between programs requesting access to resources and a lock manager are API calls that typically execute no more than a few hundred instructions. The CFS cluster environment is more complicated, however, for two reasons:

- **Distributed authority**. There is no single locking authority, because such an authority would be a 'single point of failure,' whose failure would incapacitate the cluster. Requests to access resources in a CFS file system can originate with any CFS instance running on any node in the cluster and be directed to any instance
- **Message latency**. Within a single node, CFS lock manager overhead is comparable to that of conventional lock managers. When messages must flow between nodes, however, the time to request and grant a lock rises by two or more orders of magnitude

CFS mitigates inter-node messaging frequency by implementing a few simple, but effective rules for controlling file system resources that can be manipulated by multiple instances at the same time:

- **Per-instance resources**. Some file system resources, such as intent and file change logs, are instance-specific; for these, CFS creates a separate instance for each node in a cluster
- **Resource partitioning and delegation**. Some resources, such as allocation unit maps, are inherently partitionable. For these, the CFS primary instance delegates control of parts of the resource to instances. For example, when an instance requires storage space, CFS delegates control of an allocation unit to it. The delegation remains with the instance until another instance requires control of it, for example, to free previously allocated space

Administrative hint 18

In clusters that host multiple CFS file systems, a best practice is to distribute file system mounting among nodes so that primary file system instances are distributed throughout the cluster.

- **Local allocation**. Each CFS instance attempts to allocate resources from pools that it controls. An instance requests control of other instances' resources only when it cannot satisfy its requirements from its own. For example, CFS instances try to allocate storage from allocation units that have been delegated to them. Only when an instance cannot satisfy a requirement from allocation units it controls does it request delegation of additional allocation units
- **Deferred updates**. For some types of resources, such as quotas, CFS updates master (cluster-wide) records when events in the file system require it or when a file system is unmounted

For purposes of managing per-instance resources, the first CFS instance to mount a file system becomes the file system's *primary instance*. The primary instance delegates control of partitionable resources to other instances.

Instance-specific resources

Some CFS resources are inherently instance-specific. For example, CFS transactions are designed so that each instance's intent log is independent of other instances' intent logs for the same file system. If a cluster node running a CFS instance fails, a cluster reconfiguration occurs, during which CFS freezes file I/O. After reconfiguration, the primary CFS instance replays the failed instance's intent log to complete any file system transactions that were unfinished at the time of the failure.

Similarly, each CFS instance maintains a separate file change log (FCL) for each file system it mounts, in which it records information about file data and metadata updates. CFS time-stamps all FCL records, and, for records from different instances that refer to the same file system object, sequence numbers them using a cluster-wide *Lamport timestamp*. Every few minutes, the primary instance merges all instances' private FCLs into a master FCL so that when applications retrieve FCL records, records from different nodes that refer to the same object are in the correct order.

Delegated resources

Access to other file system resources, for example allocation units and inodes, is inherently required by all CFS instances because, for example, any instance can create a file and allocate space to it. From a resource standpoint, this means:

- **File creation**. To create a file, a CFS instance must locate a free inode in the ilist, mark it as allocated, and populate it with file metadata
- **Space allocation**. To allocate space to a file, a CFS instance must control one or more allocation units, adjust the Free Extent Map(s) to indicate the allocated space, and record the location(s) the allocated space in the file's inode

In both cases, the CFS instance performing the action must control access to the affected metadata structures while it identifies free resources and allocates them. If a required resource is controlled by another CFS instance, there is necessarily at least one private network message exchange between the requesting and controlling instances, for example a message requesting allocation of space and a response indicating which space was allocated. Using inter-node message exchanges to manage resources for which CFS instances contend frequently limits performance in two ways:

- **Bottlenecking**. A node that manages resources that are frequently accessed by instances on other nodes can become a performance bottleneck. Cluster performance can become bounded by the speed with which one node can respond to requests from several others

- **Latency**. The *latency*, or time required for a CFS instance to allocate a resource controlled by another instance, necessarily includes the private network "round trip time" for one or more message exchanges

CFS minimizes these limitations by *delegating* control of sub-pools of file system resources that are both partitionable and likely to be manipulated by all instances, among the instances.

The first CFS instance to mount a file system becomes its primary instance. A file system's primary instance controls the delegation of certain resources, chief among them, allocation units for data and structural files (including the file system's inode list). The primary instance delegates control of these resources to other CFS instances as they are required.

Thus, for example, when a CFS instance must allocate storage space to satisfy an application request to append data to a file, it first searches the allocation units that are delegated to it for a suitable extent. If it cannot allocate space from an allocation unit it controls, it requests delegation of a suitable allocation unit from the file system's primary instance. The primary delegates an additional allocation unit to the requester, retrieving it from another instance if necessary. Once an allocation unit has been delegated to a CFS instance, it remains under control of the instance until the primary instance withdraws its delegation.

Freeing storage space or inodes is slightly different, because specific file system blocks or specific inodes must be freed. If the allocation unit containing the space to be freed is delegated to the CFS instance freeing the space, the operation is local to the instance. If, however, CFS instance A wishes to free space in an allocation unit delegated to instance B, instance A requests that the primary instance delegate the allocation unit containing the space to it. The primary instance withdraws delegation of the allocation unit from instance B and delegates it to instance A, which manipulates structural file records to free the space. Delegation remains with instance A thereafter. The change in delegation is necessary because freeing space requires both an inode update (to indicate that the extent descriptors that map the space are no longer in use) and an update to the structural files that describe the state of the allocation unit. Both of these must be part of the same transaction, represented by the same intent log entry; therefore both must be performed by the same CFS instance.

A CFS file system's primary instance maintains an in-memory table of allocation unit delegations. Other instances are aware only that they do or do not control given allocation units. If the node hosting a file system's primary CFS instance fails, the new primary instance selected during cluster reconfiguration polls other instances to ascertain their allocation unit delegations, and uses their responses to build a new delegation table.

Because a CFS file system's primary instance is a focal point for delegated resources, a best practice in clusters that support multiple file systems is to distribute file system primary instances among the cluster's nodes using either the **fsclustadm setprimary** command (to change the primary node while the file system is mounted) or the **cfsmntadm setprimary** command (to change the

primary node permanently). This enhances operations in two ways:

- **Load balancing**. Resource delegation-related traffic is distributed among the cluster's nodes
- **Limited impact of failure**. If a cluster node fails, only file systems for which the failed node's CFS instance was the primary instance require re-delegation

Asynchronously updated resources

The third type of per-instance resource that CFS controls is that whose per-instance control structures can be updated asynchronously with the events that change their states. Structural files that describe resources in this category include:

- **User quota files**. During operation, the CFS instance that controls the master quota file delegates the right to allocate quota-controlled space to other instances on request. Each CFS instance uses its own quota file to record changes in space consumption as it allocates and frees space. The primary CFS instance reconciles per-instance quota file contents with the master each time a file system is mounted or unmounted, each time quota enforcement is enabled or disabled, and whenever the instance that owns the master quota file cannot delegate quota-controlled space without exceeding the user or group quota. Immediately after reconciliation, all per-instance quota file records contain zeros
- **Current usage tables**. These files track the space occupied by filesets. As it does with quota files, CFS reconciles them when a file system is mounted or unmounted. When an instance increases or decreases the amount of storage used by a fileset, it adjusts its own current usage table to reflect the increase or decrease in space used by the fileset and triggers background reconciliation of the current usage table files with the master
- **Link count tables**. CFS instances use these files to record changes in the number of file inodes linked to an extended attribute inode. Each time an instance creates or removes a link, it increments or decrements the extended attribute inode's link count in its link count table. A file system's primary instance reconciles per-instance link count table contents with the master file whenever the file system is mounted or unmounted, when a snapshot is created, and in addition, periodically (approximately every second). When reconciliation results in an attribute inode having zero links, CFS marks it for removal. Immediately after reconciliation, all per-instance link count tables contain zeros

Administrators can query CFS for information about current space usage against quotas, as well as usage of clone space. In the course of responding to these queries, CFS reconciles the per-node structural files that contain the requested information.

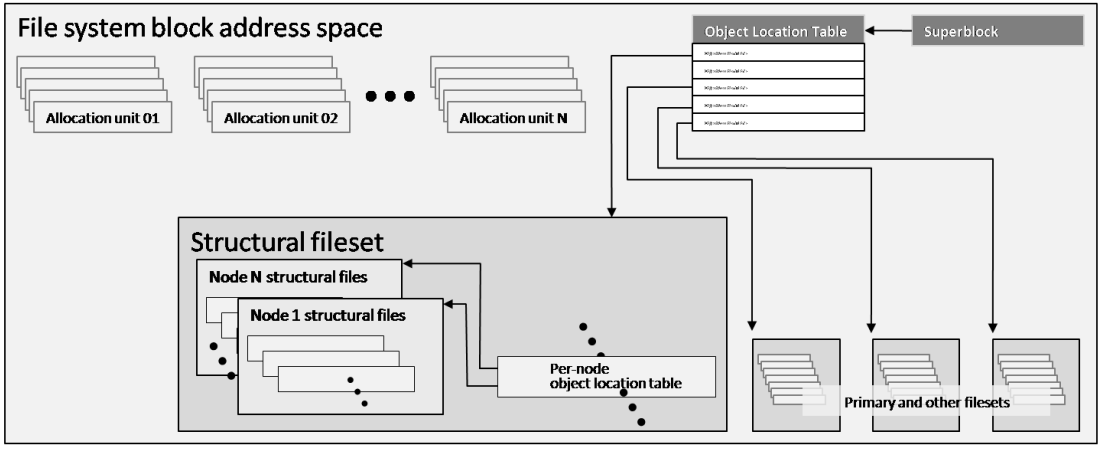
Reconciling per-node resource control structures removes the compute, I/O, and

message passing time from client response latency, while at the same time maintaining a cluster-wide view of file system resource consumption that is as timely as it needs to be for correct operation.

Per-instance resource management data structure layout

CFS organizes per-instance file system metadata as collections of files in a file system's structural fileset. As nodes are added to a cluster and shared file systems are mounted on them for the first time, CFS creates the required per-instance structural files. Figure 6-9 is a slightly more detailed representation of the file system structural overview presented in Figure 6-3 on page 119, illustrating the *Per-Node Object Location Table* (PNOLT) structural file that contains an object location table for each CFS instance's per-node structural files. Each of these object location tables contains pointers to per-node structural files for the CFS instance it represents.

Figure 6-9 CFS per-instance structural files



Chapter 7

Inside CFS: transactions

This chapter includes the following topics:

An important reason for CFS's high level of file system integrity is its transactional nature. CFS groups all operations that affect a file system's structural integrity, both those resulting from client requests and those required for its own internal "housekeeping," into *transactions* that are guaranteed to leave file system metadata intact, and file system structure recoverable if a failure occurs in the midst of execution. Before executing a transaction, a CFS instance *logs* its intent to do so in its intent log structural file by recording the metadata changes that make up the transaction in their entirety.

A cluster node may crash with one or more CFS transactions partially complete, causing a cluster reconfiguration. If the failed node had been host to a file system's primary CFS instance, another instance assumes the primary role. The new primary instance recovers from the failure by *replaying* the failed instance's intent log, completing any outstanding transactions.

Transactions in information technology

In digital information technology, the term "*transaction*" describes any set of operations that must be performed either in its entirety or not at all. Computer system transactions are sometimes described as having *ACID properties*, meaning that they are:

- **Atomic**. All operations in a transaction execute to completion, or the net effect is that none of them execute at all
- **Consistent**. Data on which a transaction operates is consistent after the transaction executes, whether execution is successful (committed) or unsuccessful (aborted)
- **Isolated**. The only system states that are observable from outside the transaction's context are the before and after states; intermediate stages in its execution are not visible
- **Durable**. Once completion of a transaction has been signaled to its initiator, the results persist, even if, for example, the system executing it fails immediately afterwards

The transaction metaphor is apt. In business, a seller delivers something of value (goods or services) as does the buyer (money). Unless both deliveries occur, the transaction is incomplete (and the consequences are undesirable).

The same is true for a file system. For example, removing file system blocks from the free space pool and linking them to a file's inode must be performed as a transaction. If both operations complete, the transaction is satisfied and the file system's structure is intact. If blocks are removed from the free pool, but not linked to a file, they are effectively "lost" (an undesirable consequence). If they are not successfully removed from the free space pool, but *are* linked to a file, they may later be removed again, and doubly allocated to another file (a *really* undesirable consequence).

Ideally, all transactions would complete—sellers would deliver products or services, and buyers would pay for and be satisfied with them. But the unanticipated happens. Goods are out of stock. Shipments are lost. Services are unsatisfactory. In such cases, the outcome is not ideal, but can be satisfactory as long as the transaction can be *recovered* (completed, as for example, goods restocked and delivered, lost shipments found, or unsatisfactory services performed again correctly) from the event that disturbed it.

Again, the same is true of file systems. If a transaction cannot be completed, for example because the computer crashes while executing it, the only acceptable outcome is to complete it eventually. For example, if file system blocks are removed from a file system's free space pool but the computer crashes before they have been linked to a file, the blocks should not be lost. When the computer restarts, the file system should "remember" that the blocks were removed from the free space pool, and link them to the file.

Administrative hint 19

To improve performance, an administrator can delay the logging of metadata updates briefly by mounting a file system in **delaylog** mode. The **delaylog** mode is the default CFS mounting mode. To force immediate logging of every metadata update as described in this section, an administrator specifies the **log** option when mounting a file system.

As the example suggests, the most frequent cause of file system transaction failure to complete (and the greatest challenge to file system designers) is a crash of the computer that hosts the file system at an instant when one or more transactions are partially complete.

Protecting file system metadata integrity against system crashes

File system transactions are essentially sequences of I/O operations that modify structural metadata. File systems must protect against the possible consequences of system crashes that occur during these sequences of operations. They do this in a variety of ways. One of the most common, which is used by CFS, is to maintain a persistent *journal* or *log*, that records the file system's intent to perform the operations that make up a transaction before it executes them.

CFS logs are aptly called *intent logs*, since they record CFS instances' intentions to execute transactions. A CFS instance records a log entry that describes an entire transaction before executing any of the metadata updates that make up the transaction.

Each CFS instance maintains an independent intent log in a structural file for each file system that it mounts. A file system's primary CFS instance allocates an intent log file when it mounts the file system for the first time. CFS determines the size of this intent log based on file system size. As other nodes mount the file system for the first time, CFS creates intent logs for them, with a size equal to that of the current size of the primary instance's log.

Administrators can adjust the sizes of individual intent logs in the range of 16-256 megabytes. When an intent log fills with uncompleted transactions, file

system operations stall until some transactions complete and log space can be freed. For this reason, larger intent logs are recommended for cluster nodes in which metadata activity (file creations, appends, renames, moves, and deletions) is frequent; nodes whose predominant file I/O activity is reading existing files can usually make do with smaller logs.

CFS transactional principles

CFS performs all operations that alter a file system's structural metadata as transactions. The design of its data structures and algorithms follows three principles that facilitate the structuring of file system updates as transactions, and make CFS file systems particularly resilient to the complex failure scenarios that can occur in clusters:

- **Resource control**. CFS instances only execute transactions that affect resources that they control. They gain control of resources either by locking them, as with file inodes, or by delegation from the primary instance, as with allocation units. This principle makes CFS instances' intent logs independent of each other, which in turn makes it possible to recover from the failure of an instance solely by replaying its intent log

  For each type of transaction, a CFS instance either requests control of the required resources from other instances, or requests that the instance that controls the required resources execute the transaction on its behalf. For example, the first time a CFS instance writes data to a newly-created file, it locks the file's inode and executes the transaction itself. Later, if another instance appends data to the file, it may request that the instance controlling the file's inode perform the append on its behalf.

  CFS instances gain control of some resources, such as inodes, by means of GLM APIs (Global Lock Manager, discussed in Chapter 8 on page 147); others, such as allocation units, they control by requesting delegation from the primary instance as described on page 133

- **Idempotency**. CFS metadata structures are designed so that every operation that can be part of a file system transaction is *idempotent*, meaning that the result is the same if it is executed twice (or more). This principle allows CFS to recover after a node crash by re-executing in their entirety all transactions in the failed cluster node's intent log that are not marked as complete, even though some of the operations within them may have been executed prior to the crash

- **Robust log format**. Storage devices generally do not guarantee to complete multi-sector write operations that are in progress when a power failure occurs. CFS therefore identifies every intent log sector it writes with a monotonically increasing *sector sequence number*. During recovery, it uses the sequence number to identify the newest transaction in the intent log file. In addition, every transaction record includes both a transaction sequence

number and information that describes the totality of the transaction. During recovery, CFS uses this information to replay transactions in proper sequence and to avoid replaying partially recorded transactions

CFS transaction flow

CFS transactions are logged and executed in an order that guarantees file system metadata recoverability if a crash occurs during transaction execution. Unless explicitly specified by the file system's mount mode, or in an application's I/O request, completion of a CFS transaction does not imply that application data has been written persistently. For example, when an application requests that data be appended to a file, the typical sequence of CFS operations is:[19]

1) **Resource reservation**. The CFS instance identifies free space in an allocation unit delegated to it (or if necessary, requests delegation of an additional allocation unit from the primary instance). In addition, it locks access to the file's inode to prevent other instances from executing transactions that involve it
2) **Block allocation**. The CFS instance identifies the required number of free file system blocks from allocation units delegated to it and updates in-memory allocation unit metadata to indicate that the blocks are no longer free, preserving enough information to undo the operation if necessary
3) **Block assignment**. The CFS instance updates the cached image of the file's inode by linking the newly-allocated blocks to an extent descriptor, again preserving enough information to restore the inode's prior state if the transaction must be undone. (The file's time of last modification are updated in the cached inode image as well.) If no suitable extent descriptor is available, the instance allocates an indirect extent map for this purpose and links it to the inode
4) **Intent log write**. The CFS instance constructs an intent log record, assigns a transaction number, and records the metadata update operations that make up the transaction in its intent log. At this point, the transaction is said to be *committed*, and can no longer be undone
5) **CFS transaction completion**. Once its record has been recorded in the intent log, the CFS transaction is complete. At this point, neither the metadata updates that make up the transaction nor the user data is guaranteed to have been written persistently
6) **Data movement**. The CFS instance allocates pages from the operating system page cache and copies the appended data from the application's buffer

19. This simplified description represents CFS operation when the **log** mount option is in effect and when the application I/O request does not specify that data be written directly from its buffer. Direct I/O, the **delaylog** (default) and **tmplog** mount options, and the **sync** and **dsync** write options result in slightly different behavior.

into them. Once this is done, the application may reuse its buffer for other purposes

7) **Completion signal and data flush**. Once the CFS instance has transferred the application's data to page buffers, further behavior depends on whether the application request specified that data was to be written synchronously. If it did, the instance schedules the data to be written to disk and waits for the write to complete before signaling request completion to the application. If synchronous writing was not specified, the instance signals completion immediately, and writes the application data "lazily" at a later time

8) **Metadata update completion.** When the metadata updates that make up the transaction have been written to disk, the CFS instance writes a "transaction done" record containing the transaction's sequence number in the intent log. It uses this record during recovery to help determine which transaction log records need to be replayed

The cluster node hosting a CFS instance may crash at any point in this sequence of operations. If a crash occurs prior to step 4 of this procedure, there is no transaction, because nothing has been written in the intent log, nor have any persistent metadata structures or user data been modified. If a crash occurs at any time between steps 4 and 7, CFS recovers by reading the transaction record from the intent log and repeating the operations it contains.

If the application specified a synchronous append, and had received notification that its request was complete before the crash, the appended data is on disk. Otherwise, CFS blocks applications from reading the contents of the newly allocated blocks until they have been successfully written by an application.

If crash occurs after step 8, the entire transaction is reflected in the file system's on-disk metadata. If metadata updates from prior transactions had not been written to disk at the time of the crash, the transaction must still be replayed during recovery, because restoring a CFS file system to a consistent state after a crash requires that transactions be replayed in the order of their original occurrence.

CFS transactions and user data

CFS transactions are designed to preserve the structure of a file system; in essence to maintain the correctness of file system metadata. Completion of a CFS transaction does not necessarily guarantee that when CFS signals an application that its write request is complete, metadata and file data updates have been written to disk. Moreover, the default file system mount option for logging permits CFS to delay logging for a few seconds in order to coalesce multiple intent log writes.

Administrators and application programs can employ a combination of CFS mount options and POSIX caching advisories to specify file data and metadata persistence guarantees. Table 7-1 summarizes the effect of the CFS **delaylog** and

log mount options and the POSIX **O_SYNC** and **O_DSYNC** caching advisories on data and metadata persistence.

Table 7-1 CFS Intent log, metadata, and data persistence at the time of request completion

If the mount option is → …and the file cache advisory option is ↓	delaylog (default) or tmplog mount option	log mount option
Asynchronous (default)	**Log record:** may still be in cache **Metadata:** may still be in cache **File data:** may still be in cache	**Log record:** on disk **Metadata:** may still be in cache **File data:** may still be in cache
O_SYNC	**Log record:** on disk **Metadata:** on disk **File data:** on disk	**Log record:** on disk **Metadata:** on disk File data: on disk
O_DSYNC	**Log record:** on disk **Metadata:** may still be in cache (non-critical metadata only) **File data:** written to disk	**Log record:** on disk **Metadata:** on disk **File data:** on disk

By default, CFS caches both file data and metadata and intent log records to optimize performance, and writes them to disk "lazily," after it has signaled write completion to applications. It flushes metadata updates to its intent log every few seconds, but in general, if default mount and write options are in effect, file data, metadata, and the intent log record that describes the write may still be in cache when an application receives notification that its write request is complete.

This behavior allows CFS to schedule disk writes optimally, and is suitable for many applications. For some, however, guaranteeing that intent log records have been written and that on-disk metadata and/or file data are up to date have higher priority than file I/O performance. If for whatever reason, application code cannot be modified, for example, by adding **fsync()** calls at critical points, an administrator can guarantee that log records are persistent before applications progress specifying the **log** option when mounting a file system.

Administrative hint 20

An administrator can specify the mount option **convosync=delay** to override application programs **O_SYNC** and **O_DSYNC** cache advisories. This may improve performance, but at the risk of data loss in the event of a system crash.

Some applications make synchronous file I/O requests. For those that do not, if source code is available for modification, the POSIX file control function can be inserted to set the write mode for critical files to either:

- **Synchronous (O_SYNC)**. With this option, both metadata and file data disk writes are complete when CFS signals completion of the application request
- **Data synchronous (O_DSYNC)**. With this option, file data disk writes are complete, but metadata may still be in cache when CFS signals completion of the application request

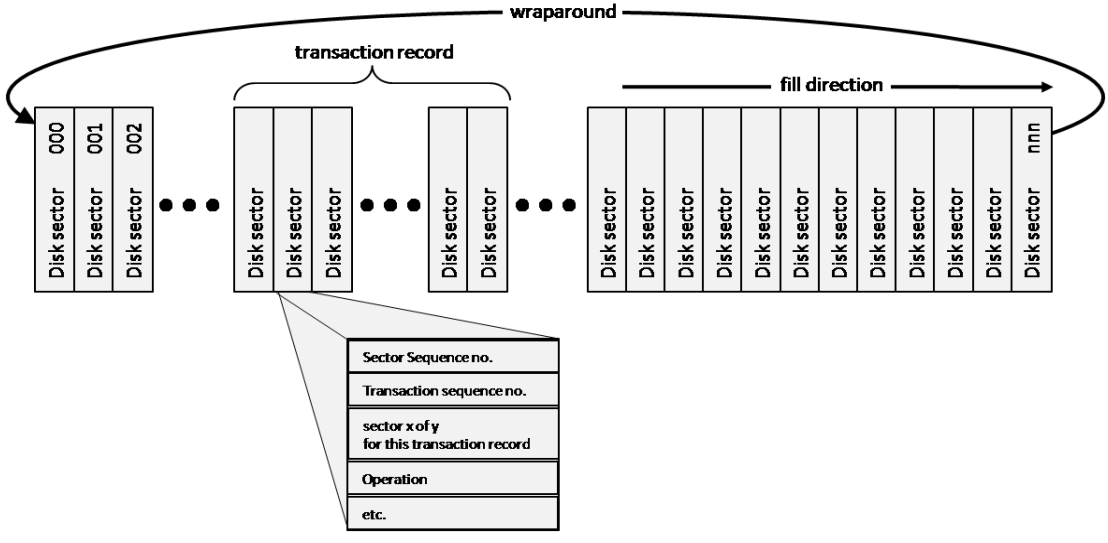
Applications can use either or both of these options selectively to ensure that updates to critical files are written synchronously, and allow less critical updates that can be recovered at application level to be written asynchronously for improved application performance.

Alternatively, applications can use explicit POSIX **fdatasync()** or **fsync()** system calls to cause CFS to flush file data only or both file data and metadata to disk at key points during execution. When a CFS instance signals completion of **fdatasync()** and **fsync()** system calls, all file and/or metadata has been written to disk.

CFS intent logs

Each CFS file system contains an intent log structural file for each instance that mounts it. Intent logs are contiguous, and are written *circularly*—a CFS instance records transactions sequentially until the end of the log is reached, and then continues recording from the beginning. When a file system's I/O load includes frequent metadata updates, an intent log may fill. Continued recording would overwrite transaction records for which some metadata updates were not yet written to disk. When this occurs, CFS delays further recording (and consequently, delays signaling completion to applications) until sufficient intent log space has been 'freed' by writing older transaction records to disk.

Figure 7-1 CFS intent log structure



If the performance of an update-intensive application becomes sluggish, an administrator can increase the size of the intent log to enable more transaction records to be buffered. Different CFS instances' intent logs may be of different sizes. For example, if an application on one node is adding files to a file system frequently (transaction-intensive) and another is reading the files to report on their contents (largely non-transactional), an administrator might increase the intent log size for the former instance and leave the latter at its default value.

Administrative hint 21

An administrator uses the **logsize** option of the **fsadm** console command to alter the size of a CFS instance's intent log. Larger intent logs take longer to replay, and can therefore elongate crash recovery times.

The **logvol** option of the **fsadm** command can be used to place the intent log on a specific volume in the file system's VSET.

CFS intent logs are *contiguous*, that is, they occupy consecutively numbered file system blocks, so with rare exceptions, each write to an intent log results in a single disk write. This minimizes the impact of intent log writes on application performance. In multi-volume file systems, administrators can place intent logs on separate volumes from user data; by reducing seeking, this further minimizes the impact of intent log writes on application I/O performance. Flash-based solid state disks might seem a good choice for intent log volumes, but because the number of writes they can sustain before wearing out is limited, the high write frequency of intent logs may make them less than optimal for this purpose.

"Crash-proofing" CFS intent logs

The basic purpose of CFS intent logs is to enable rapid recovery from crashes, so intent logs themselves must be "crash-proof." In particular, during recovery, CFS must be able to identify unambiguously:

- **Log boundaries**. The logical beginning and end of a circular intent log
- **Transaction status**. For each logged transaction, whether the transaction is "done" (that is, whether all metadata updates have been written to persistent disk storage)
- **Transaction content**. The complete set of metadata operations that make up each logged transaction
- **Transaction order**. The order in which logged transactions were originally executed

These must all be unambiguously identifiable from an intent log, no matter what was occurring at the instant of a crash. Four structural features of CFS intent logs make this possible:

- **Disk sector sequence**. Each sector of a CFS intent log includes a monotonically increasing sector sequence number. Sector sequence numbers enable CFS recovery to determine the most recently written intent log sector, even if a system crash results in a disk write failure
- **Transaction sequence**. Each logged transaction includes a monotonically increasing transaction sequence number. These numbers enable CFS recovery to replay transactions in order of their original execution
- **Transaction content**. Each intent log disk sector contains information about only one transaction. Information in each sector of a transaction record enables CFS recovery to determine whether the log contains the complete transaction record, or was truncated, for example because it was being written at the instant of the crash. CFS does not recover incompletely logged transactions
- **Transaction state**. When a transaction's metadata updates have been completely written to disk, CFS writes a "transaction done" intent log entry signifying that the transaction's metadata updates are persistent. CFS recovery uses these records to identify the oldest logged transaction whose metadata updates may not be completely persistent. During recovery, it replays that and all newer transaction records

Thus, as long as the volume containing an intent log structural file survives a system crash intact, CFS recovery can interpret the log's contents and reconstruct transactions that may have been "in flight" at the time of the crash. Replaying the idempotent operations that make up these transactions guarantees the structural integrity of the recovered file system.

Chapter 8

Inside CFS: the Global Lock Manager (GLM)

This chapter includes the following topics:

- General requirements for a lock manager
- GLM architecture
- GLM operation

Any modern file system must coordinate multiple applications' concurrent attempts to access key resources, in order to prevent the applications, or indeed, the file system itself, from corrupting data or delivering incorrect responses to application requests. For example, if Application A writes data to multiple file blocks with a single request and Application B reads the same file blocks with a single request at around the same time, Application B may retrieve either the pre-write or post-write contents of the blocks, but should not ever see part of the data written by Application A and part of the blocks' prior contents.

Correct behavior in the presence of multiple applications is particularly important for CFS, which is highly likely to be employed in environments in which multiple applications are active on multiple cluster nodes. All CFS instances on all cluster nodes must respect file system resources that are locked to prevent concurrent access.

Like most file systems, CFS uses "*locks*"—in-memory data structures—to coordinate access to resources. A lock indicates that a resource is in use by a file system execution thread, and places restrictions on whether or how other threads may use it. CFS locks must be visible and respected throughout the cluster. The CFS component that manages resource locking across a cluster is called the Global Lock Manager (GLM).

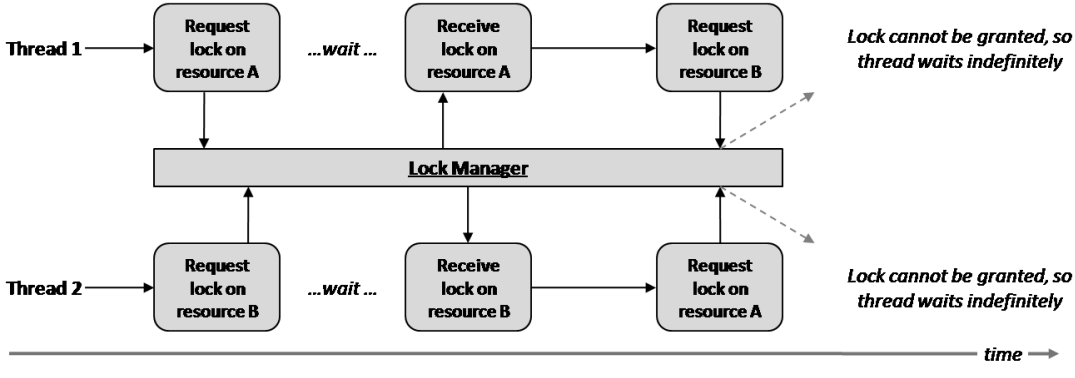
General requirements for a lock manager

Controlling access to file system resources is simple to describe, but complex to implement. Within CFS, for example, execution threads must observe a protocol when they contend for shared or exclusive access to a file system's resources. The protocol is a "gentlemen's agreement" among trusted entities, in the sense that it is not enforced by an external agency. If all CFS threads do not strictly adhere to the protocol, orderly control over resources breaks down.

The basic requirements for any file system locking protocol are:

- **Unambiguous resource identification**. The resources to be controlled must have unique names. To control access to a directory, a file's inode, or a range of byte addresses within a file, a file system must be able to identify the resources uniquely
- **Deadlock prevention**. Deadlocks occur when two or more execution threads wait indefinitely for events that will never happen. One simple form of deadlock, illustrated in Figure 8-1, occurs when a thread (**Thread 1**) controls one resource (**Resource A**) and attempts to gain control of a second resource (**Resource B**), which another thread (**Thread 2**) already controls. Meanwhile, **Thread 2** is attempting to gain control of **Resource A**. The result is that both threads wait indefinitely. Several solutions to the deadlock problem are known; a lock management protocol must adopt one of them

Figure 8-1 Classic deadlock scenario



- **Range locking**. In some cases, for example updating file metadata, blocking access to the entire file by locking its inode is appropriate. In others, particularly those in which many application threads update small amounts of data within large files, locking entire files would serialize application execution to an intolerable degree. Lock managers should include mechanisms for locking a range of bytes within a file rather than the entire file

- **Locking levels**. In some cases, as for example when it is updating inode metadata, a thread requires exclusive access to a resource. In others, however, exclusive access is unnecessarily restrictive and the serialization it causes can impact performance adversely. For example, a CFS execution thread that is listing directory contents can share access to directories with other threads as long as the directories' contents do not change. Thus, lock managers must generally include both exclusive and shared locking capabilities or *levels*

Cluster lock manager requirements

GLM must present the same view of lock state to all of a cluster's CFS instances at all times. Whenever a CFS execution thread gains or relinquishes control of a resource, its action must be reflected in all CFS instances' subsequent attempts to gain or relinquish control of that and other related resources. This inherently requires that GLM instances communicate some amount of state change information with each other. Thus, in addition to the basic requirements for any lock manager, CFS has three additional ones:

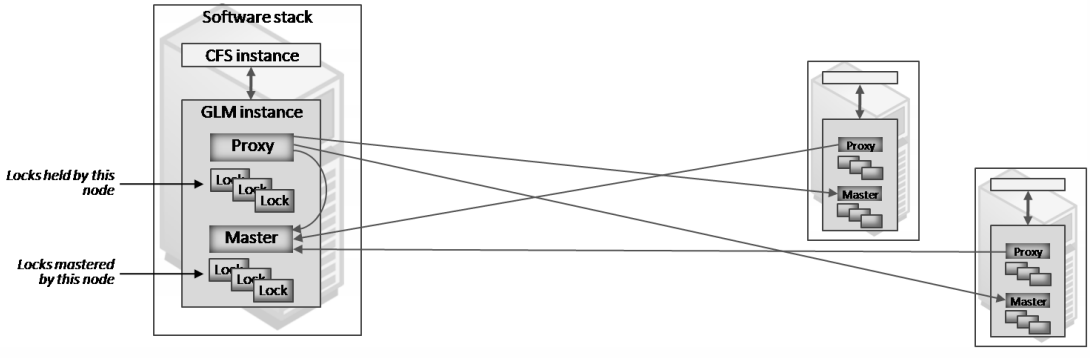
- **Message overhead sensitivity**. In single-host file systems, resource locking imposes relatively little overhead because lock request "messages" amount to at most a few hundred processor instructions. In a cluster, however, a message exchange between nodes can take hundreds of microseconds. Minimizing a lock management protocol's inter-node message overhead is therefore an important requirement
- **Recoverability**. A key part of the CFS value proposition is *resiliency*—continued accessibility to file data when, for example, a cluster node fails. From a resource locking point of view, if a node fails, CFS instances on other nodes must dispose of the failed instance's locks appropriately, and adjust the cluster-wide view of resource lock state so that applications can continue to access file systems with no loss of data or file system structural integrity
- **Load balancing**. Ideally, the messaging, processing, and memory burden of managing locks should be distributed across a cluster, partly so that ability to manage locks scales along with other file system properties, but also to facilitate recovery from node failure by minimizing the complexity of lock recovery

GLM architecture

The Global Lock Manager (GLM) cooperates with other CFS components to present a consistent view of the lock state throughout a cluster, even when member nodes are added or removed. Figure 8-2 illustrates the overall GLM architecture. As the figure suggests, the GLM service is *distributed*—an instance runs on each cluster node. Collectively, the instances provide a *distributed*

master locking service—each cluster node is responsible for *mastering* locks for a unique part of the system's *lock name space* (the set of all possible lockable resource names).

Figure 8-2 Global Lock Manager (GLM) architecture



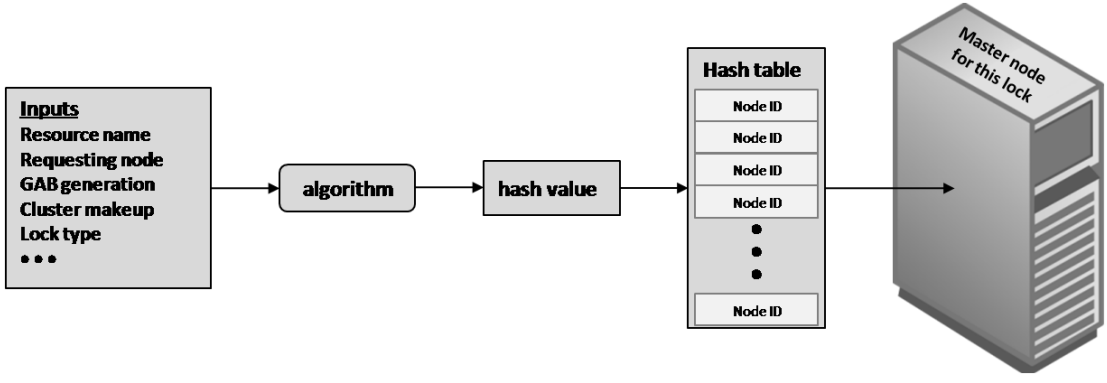
As Figure 8-2 suggests, each GLM instance has two executable components:

- **Master**. An instance's master component manages an in-memory database of existing locks for which the instance is the master. A GLM master component *grants* locks to its own proxy component and to other nodes that request control over resources for which it is responsible
- **Proxy**. When a CFS execution thread first requests control of a resource, GLM forwards the request to the resource's master. In return it receives a delegation that allows it to act as the master's proxy, and grant locks of equal or less-restrictive level to other threads

The GLM name space and distributed lock mastering

When a CFS execution thread requests a lock on a resource, GLM computes the resource's 32-byte *lock name* using an algorithm guaranteed to produce a unique name for each possible file system resource. It hashes the resulting lock name to produce an index into a table of node ID numbers. The node ID in the indexed table cell designates the GLM master for the resource.

Figure 8-3 Determining the master node for a GLM lock



When it begins to execute, GLM builds a lock name hash table based on cluster membership. The table is typically between 100 and 1,000 times larger than the number of nodes in a cluster, so each node ID appears multiple times in the table.

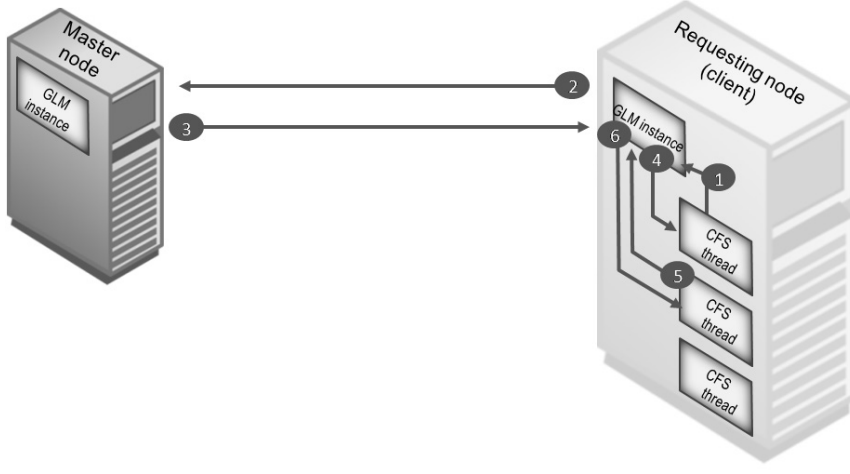
The GLM locking hierarchy

Cluster-wide resource locking inherently requires that messages be passed among nodes, increasing *latency*, or time required to grant and release locks. One objective of any distributed lock manager design must therefore be to minimize the number of messages passed among nodes. In furtherance of this objective, GLM uses a two-level locking hierarchy consisting of:

- **Node grants**. When a CFS execution thread requests a shared or exclusive lock on a resource, its local GLM instance requests that the resource's master issue a *node grant* for the resource. If there no conflicts, the GLM master issues the node grant, making the requesting node's GLM instance a *proxy* for the resource. The proxy is empowered to grant non-conflicting locks to CFS execution threads on its node
- **Thread grants**. A GLM instance acting as proxy for a resource can issue non-conflicting *thread grants* for locks of equal or less-restrictive level than its node grant to CFS execution threads on its node without exchanging messages with other nodes

Figure 8-4 illustrates the steps in issuing node and thread lock grants.

Figure 8-4 Node and thread lock grants



1) A CFS execution thread requests a lock from its local GLM instance
2) The local GLM instance computes the master node ID for the resource and requests a node grant from its GLM instance
3) The master GLM instance for the resource issues a node grant to the requesting instance
4) The requesting GLM instance issues a thread grant to the requesting CFS execution thread
5) Another CFS thread on the same cluster node requests a non-conflicting lock on the resource
6) The local GLM instance grants the lock without referring to the resource's master GLM instance

A *node grant* on a resource effectively delegates proxy authority to grant locks requested by CFS execution threads to the GLM instance on the node where the requests are made. As long as a node holds a node grant, its GLM instance manages local lock requests that are consistent with the node grant without communicating with other nodes. When a different node requests a conflicting lock from the resource's master node, the resource master's GLM sends *revoke* messages to all proxies that hold conflicting node grants on the resource. The proxies block further lock requests and wait for threads that hold thread grants on the resource to release them. When all thread grants have been released, the proxy sends a *release* message to the resource's master, which can then grant the requested new lock.

GLM lock levels

CFS execution threads request control of resources at one of three levels:

- **Shared**. The requesting thread requires that the resource remain stable (not change), but is content to share access to it with other threads that also do not change it. GLM permits any number of concurrent shared locks on a resource
- **Upgradable**. The requesting thread can share access to the resource, but reserves the right to request exclusive access at a later time. GLM permits any number of concurrent shared locks plus one upgradable lock on a resource. The upgradable lock level permits a thread to prevent a resource from being modified, while deferring prevention of other threads from reading the resource until it actually requires exclusive access
- **Exclusive**. The requesting thread requires exclusive access to the resource, usually to update it, for example, to modify file metadata in an inode or to write a range of file blocks. As the name implies, an exclusive lock cannot co-exist with any other locks

GLM-CFS execution thread collaboration

Like any lock management scheme, GLM must avoid "deadlocks"—situations such as that illustrated in Figure 8-1 on page 148, in which two or more execution threads vie for control of resources held by other threads. Several techniques for avoiding deadlocks are known, including lock timeouts and analysis of lock requests against existing locks. One of the most straightforward techniques, which is used by CFS, is the *cooperative client*. CFS execution threads are designed to order their lock requests so as to avoid most situations that might result in deadlocks. For situations in which deadlocks would be possible, GLM includes a *trylock* mechanism that allows an execution thread to test whether a lock can be granted and request the grant in a single operation.

Most requests to GLM are synchronous; GLM blocks execution of the requesting thread until it has satisfied the request. In the scenario depicted in Figure 8-1, this would cause both CFS execution threads to wait indefinitely for the release of resources locked by the other thread. CFS execution threads use the trylock mechanism to avoid situations like this.

A GLM trylock is a conditional request for a thread grant on a resource. If no existing locks conflict with the request, the proxy grants the lock. If the request does conflict with an existing lock, the proxy responds to the calling thread that the resource is "busy," rather than forcing the caller to wait for the lock to become grantable, as would be the case with a normal lock request.

Thus, referring to the example of Figure 8-1, **Thread 1** might use a trylock to request control of **Resource B**. On discovering that its request had failed because **Resource B** is already locked by **Thread 2**, **Thread 1** might release its

lock on **Resource A**, which would free **Thread 2** to obtain the lock and continue executing to completion.

CFS execution threads request trylocks in one of three "strengths:"

- **Weak**. The proxy grants the lock if it can do so without exchanging messages with the resource master, otherwise it responds to the requester that the resource is busy. Weak trylocks are sometimes used by CFS internal background threads operating asynchronously with threads that execute application requests
- **Semi-strong**. The proxy exchanges messages with the resource master if necessary. The master grants the lock if it can do so without revoking locks held by other nodes; otherwise it responds that the resource is busy
- **Strong**. The proxy exchanges messages with the resource master if necessary. The master revokes conflicting locks if possible, and grants the lock as long as no conflicting locks remain in effect. CFS typically uses strong trylocks when executing application requests

GLM trylocks are a simple and useful method for avoiding deadlocks, but they depend on proper behavior by CFS execution threads. GLM is thus not a general-purpose distributed lock manager that could be used by arbitrary applications whose proper behavior cannot be guaranteed. Largely for this reason, the GLM API is not exposed outside of CFS.

Minimizing GLM message traffic

GLM includes several mechanisms that minimize lock-related message traffic among cluster nodes. Two of these—node grants and trylocks—have already been discussed. Node grants effectively delegate locking authority within a single node to the GLM instance on the node. Trylocks, particularly weak trylocks, enhance the delegation mechanism by making it possible for execution threads to avoid intra-node deadlocks without inter-node message exchanges. GLM *masterless locks* are a third mechanism for minimizing lock-related inter-node message traffic.

A CFS instance requesting a lock on a resource can specify that the lock be *masterless* as long as it can guarantee that it will be the only instance to request locks on the resource. When it holds a masterless lock on a resource, a GLM proxy may grant on its own authority locks for which it would ordinarily have to consult the resource master.

For example, a CFS instance must hold at least a shared *hlock* (hold lock) on a file's inode before it is permitted to request an *rwlock* (read-write lock) on the inode. A CFS instance that holds an exclusive hlock on an node can request thread grants for masterless rwlocks. There is no need to communicate with the resource's GLM master because no other node can request an rwlock unless it holds an hlock, which cannot be the case, because the local node's hlock is exclusive.

A lock can remain masterless as long as no threads on other nodes request access to the resource. When this is no longer the case (for example, if the CFS instance holding an exclusive hlock downgrades it to shared), threads that hold masterless locks affected by the downgrade must request that GLM *normalize* them, allowing other nodes to access to the resources.

Masterless locks are another example of cooperative client behavior. CFS execution threads only request masterless locks on resources that they "know" other nodes will attempt to access. Moreover, threads that hold masterless locks must normalize them when the conditions that make them masterless are no longer in effect; otherwise, resources could remain inaccessible to other threads indefinitely.

Block locks

For certain types of resources, it is highly probable that a single CFS instance will attempt to lock several of them. For example, when creating new files, an instance often allocates (and locks) consecutively numbered inodes from a block delegated to it by the file system's primary instance.

The GLM *block lock* is a mechanism for making locks on similar resources such as inodes masterless. When an instance expects to be the exclusive user of multiple resources of similar type, such as consecutively numbered inodes, it creates a *block name* that uniquely identifies the resources, and includes it in each of its requests for locks on them. Internally, GLM attempts to obtain an exclusive lock on the block name when it first encounters it. If it succeeds, each lock request that contains the block name can be masterless, eliminating the need for GLM to communicate with other instances before granting it. For example, as an instance allocates free inodes when creating files, it includes the block name that identifies the block of consecutive inode numbers delegated to it within its lock request. If the first block lock request succeeds, CFS can make all inode locks from this block masterless.

Range locks

The primary CFS resource to which GLM locks apply is the file inode. Before performing any operation that affects a file's inode, a CFS execution thread secures a *hold lock* (*h-lock*) on the inode. Threads secure additional locks in order to read or write file data.

For single-writer applications, such as those that write files sequentially, locking an entire file while data is written to it is adequate. Some applications, however, such as multi-threaded business transaction processors, manipulate relatively small records within a few large CFS files concurrently. Locking an entire file each time one of these applications updated a record would serialize their execution to an unacceptable degree.

For the most part, multi-threaded applications' threads do not interfere with

each other—two threads that update non-overlapping records in a file simultaneously can execute concurrently without corrupting data. But it is possible for two or more threads of such an application to attempt to update records in the same file block at the same time. To prevent data corruption (for example, wrong record written last, or part of one update and part of another in the resulting record) when this happens, a thread must have exclusive access to a record for the duration of its update.

GLM *range locks* allow multiple execution threads to access a file concurrently, but they limit access to specific file blocks while threads update or read them. Range locks control access to specific ranges of file block addresses. As with all locks, CFS instances compute range lock names using an algorithm that guarantees uniqueness. Range locks are somewhat more complicated than ordinary locks, however, in that ranges of file blocks can overlap; for example, it is possible for one CFS execution thread to request a lock on file blocks 1-10, and another to request a lock on blocks 6-15 of the same file. Depending on the lock level requested, the two may be incompatible.

In most respects, range locks behave identically to ordinary locks—they can be taken at node or thread level, they are subject to the same revocation rules, and trylocks for ranges of file blocks are supported. When a thread requests a "greedy" range lock, however, GLM enlarges the range to a maximum value if possible. This gives the thread control of a large range of the file unless another thread requests it.

GLM operation

When a cluster node boots, GLM starts executing before CFS. Upon starting, CFS instances register with GLM. GLM instances use a shared port within the VCS Group Atomic Broadcast (GAB) protocol to communicate with each other.

To lock a resource, a CFS execution thread first creates, or *initializes*, the lock. Next, it typically requests a *thread grant*. GLM always grants thread locks and upgrade requests for existing locks as long as they do not create conflicts. Typically, GLM calls are thread-synchronous—a CFS execution thread requesting a GLM service (upgrade, downgrade, and unlock) waits for GLM to perform the service before continuing to execute. GLM does support asynchronous grant requests, which CFS uses, for example, to request multiple grants concurrently. GLM assumes, however, that its CFS clients are well-behaved in the sense that they request lock services in the proper order—initialize, grant, upgrade or downgrade, and release.

A GLM node grant for a resource remains in effect until both of the following occur:

- **Revocation**. GLM requests revocation of the grant because a CFS execution thread on another node has requested a conflicting thread grant on the resource

- **Release**. All CFS execution threads holding thread grants on the resource release them

In this respect as well, GLM assumes that its clients are well-behaved—that they do not hold thread grants indefinitely, but release them when they no longer need to control the resources.

GLM locks occupy memory; once a CFS execution thread no longer requires a resource, and has released all data memory associated with the lock, it usually deletes, or *deinitializes* the lock. This minimizes the amount of memory consumed by GLM instances' lock databases, as well as the amount of lock database searching required when a CFS execution thread requests a grant.

GLM and cluster membership changes

Because GLM distributes lock mastership among all cluster nodes, changes in cluster membership may require that some locks be remastered. Cluster membership may change because of:

- **Node failure**. Because each GLM instance keeps its database of lock information in local memory, failure of a node causes its lock information to be lost. In particular, node locks granted to other nodes by a failed node must be recovered and a new master assigned
- **Node addition**. No lock information is lost in this case, but because lock mastership is a function of lock name and cluster membership, changes in cluster membership may result in changes in mastership for locks that exist at the time of membership change

A GLM instance's lock database contains two types of information related to instances on other nodes:

- **Mastership**. Locks for which the instance is the master, and which have been granted to instances on other nodes
- **Proxy**. Node grants that have been issued to the instance for resources mastered by other instances

When a cluster node fails, GLM must properly reconstruct its GLM information.

Node grants for which the failed node itself is master only affect execution threads on the failed node. GLM does not recover locks contained completely within a failed node; they are lost, along with the execution threads that created them.

A node failure causes a cluster reconfiguration, during which VCS suspends interactions between nodes briefly while the remaining nodes construct a new cluster. When reconfiguration is complete, each node's GAB instance sends a message to its local CFS instance containing the new cluster membership and an instruction to *restart*. During its restart processing, CFS invokes GLM's restart API.

GLM must recover relevant lock information lost from the failed node's memory. Each instance enters a *recovery* phase during which it blocks incoming lock requests. All instances send their hash tables to a designated *recovery manager* instance. The recovery manager computes a new hash table for the cluster and returns it to other instances.

Each GLM instance compares its original (pre-failure) hash table with the new (post-failure) table sent by the recovery manager to determine which of its locks must be remastered. Locks must be remastered if the new hash table maps their names to different masters. For each such lock, GLM requests a grant from the new master. Each GLM instance informs the recovery manager when it completes remastering. When all instances have reported, the recovery manager directs them to resume normal operation.

GLM recovery after a cluster reconfiguration is functionally transparent to CFS execution threads, but lock requests made during recovery do not complete until recovery is finished.

Clusters also reconfigure when nodes are added. Adding a node causes a cluster reconfiguration during which locking is inhibited while all nodes converge on a common view of the enlarged cluster. When the reconfiguration completes, the CFS instance on the new node starts and mounts file systems, and GLM instances on all nodes reinitialize themselves, using the new node population to create new hash tables for remastering and subsequent use. From that point on, GLM operates as it did prior to the reconfiguration, including the new node in its lock mastership calculations.

Ghost locks

CFS instances sometimes request that other instances perform tasks on their behalf. For example, an instance (the *updater* instance) may have to update the access time of a file that is owned by another instance (the *file owner* instance). Rather than taking ownership of the inode, the updater instance requests that the file owner instance update the file's metadata on its behalf.

The updater instance locks the file's inode before making its request. If the updater instance's node fails before the file owner instance has completed the metadata update, the file owner instance requests a *ghost lock* on the inode, so that during recovery GLM will transfer the lock held by the failed updater instance to the file owner instance.

A ghost lock request made by a CFS instance informs GLM that the instance is using a resource on which a failed instance had held the regular lock. The request causes GLM recovery to transfer ownership of the regular lock so that the node using the resource can complete its task.

In the example of the preceding paragraph, if the updater instance's node fails before the file owner instance has finished updating file metadata, the cluster reconfigures, and the file owner instance makes a ghost lock request for the

file's inode. During recovery, the GLM instance *fires* the ghost lock (converts it to regular lock held by the file owner instance). No new lock is created.

Chapter 9

Inside CFS: I/O request flow

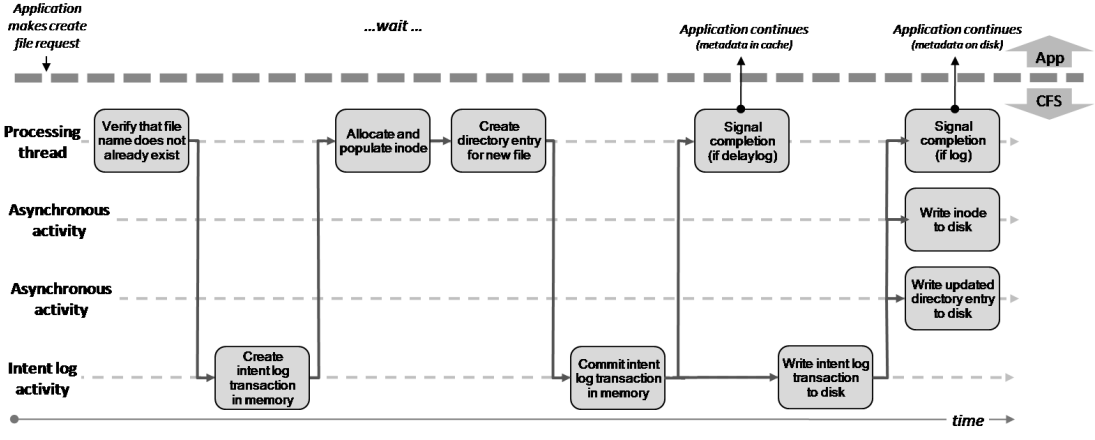
This chapter includes the following topics:

While outwardly presenting a relatively simple POSIX interface to applications, CFS is internally complex, because it must maintain POSIX semantics with an arbitrarily large number of clients on multiple cluster nodes. This section presents examples of how CFS performs some basic I/O operations to illustrate CFS's internal complexity. The examples are simplified, particularly in that they do not show GLM lock requests and grants, only the operations enabled by successful GLM lock grants.

Creating a CFS file

Figure 9-1 shows a simplified timeline for the CFS create file operation. The horizontal lines in the figure represent activities that can occur asynchronously. The arrows connecting the boxes represent dependencies on preceding events.

Figure 9-1 Creating a CFS file



The first step in creating a new file is to verify that its name is unique in the file system name space. CFS traverses the directory hierarchy until it locates the directory that would contain the file's name if it existed. If the name is indeed unique (does not already exist), CFS creates a transaction record in memory. Next, it allocates an unused inode and populates it with information about the file, such as its owner and group, and timestamps, then CFS creates an in-memory directory entry for the new file.

CFS then commits the transaction. (If the node crashes and its intent log is replayed during recovery, only transactions marked as 'committed' are re-executed).

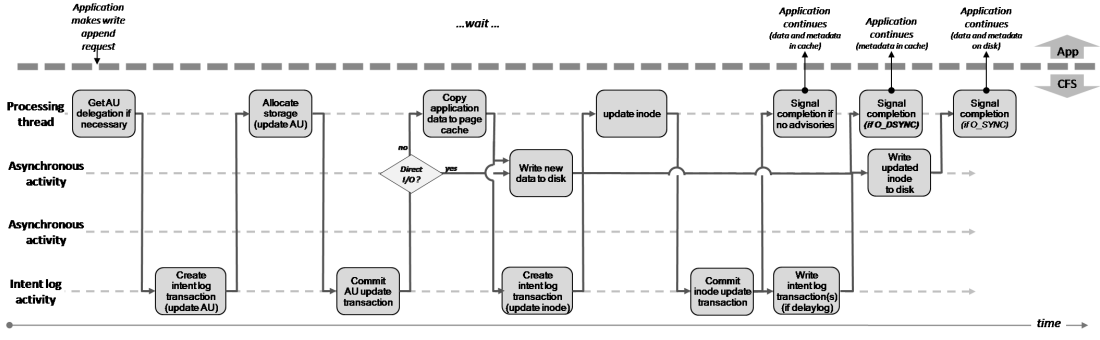
If the file system is mounted with the **log** mount option, CFS delays signaling completion to the application until it has written the transaction record to disk. If either of the **delaylog** or **tmplog** mount options are in effect, however, the intent log disk write may be deferred. CFS then signals the application that its request is complete. Once the intent log record has been written to disk, CFS performs two additional actions asynchronously to complete the file creation:

- **Directory write**. CFS writes the directory file containing the name and inode number of the newly created file
- **inode update**. CFS writes the newly created file's inode to disk storage

Appending data to a CFS file

Figure 9-2 represents the simplest sequence of actions in executing an application's request to append data to the end of a file. A CFS instance begins an append operation by acquiring ownership of an allocation unit with sufficient space if it does not already own one, and creating a transaction to allocate storage for the appended data. Once storage is allocated, the instance commits the transaction.

Figure 9-2 Appending data to a CFS file



Once the storage allocation transaction has been committed, data can be transferred. If no advisories are in effect, the instance allocates pages from the operating system page cache, moves the data to them, and schedules a write to the CVM volume. If a **VX_DIRECT** or **VX_UNBUFFERED** advisory is in effect, or if the amount of data being appended is larger than **discovered_direct_iosz** (page 219), the instance schedules the write directly from the application's buffers. (When writing directly from application buffers, the CFS instance must also invalidate any cached pages left over from earlier buffered writes that overlap with the file blocks being written.)

The instance then creates a second transaction to update the file's inode with the larger file size and the modification time. Once that transaction has been committed (and written, if the **log** mount option is in effect), the instance signals the application that its request is complete, provided that no advisories are in effect. Otherwise, it must wait for:

- **Data write**. If either of the **O_DSYNC** or **VX_UNBUFFERED** advisories is in effect for the request, CFS does signal completion until the data has been written
- **inode write**. If the **O_SYNC** or **VX_DIRECT** advisories is in effect for the request, the CFS instance does not signal completion until both data and updated inode have been written

Figure 9-2 illustrates the simplest case of an appending write. If the size of the

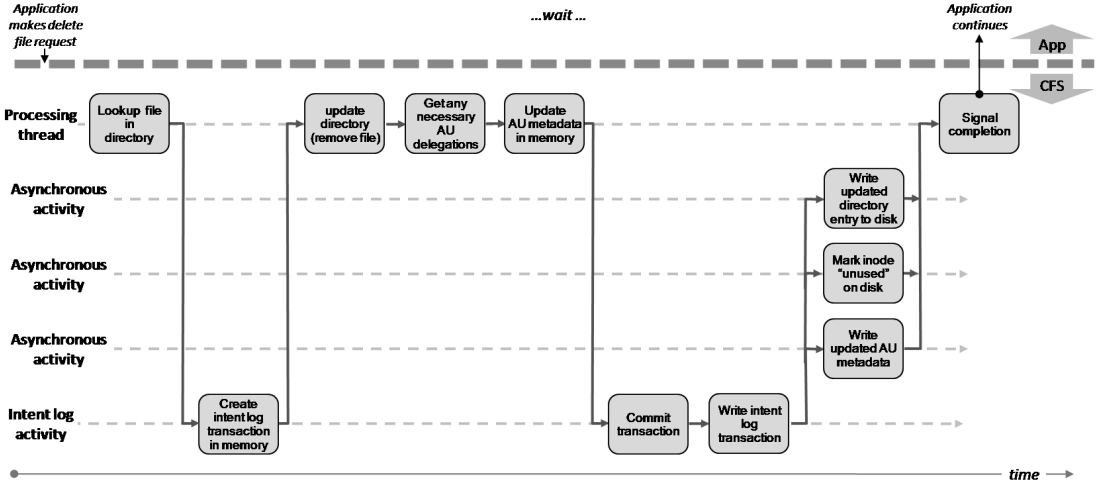
appended data is larger than **max_direct_iosz**, or if CFS has detected sequential writing, the operation is more complex.

Deleting a CFS file

Figure 9-3 illustrates the CFS sequence of actions for a simple file deletion. It assumes that no other applications have locks on the file, and that no hard links point to it. (If the "file" being deleted is actually a hard link, CFS deletes the directory entry and decrements the link count, but leaves the file inode and data in place.)

CFS starts by verifying that the file exists. It then creates a transaction for the deletion, and locks the file's directory entry. It next reads the file's inode if it is not cached (for example, if the file has not been opened recently), locks it, and uses information in it to obtain any necessary delegations for allocation unit(s) in which the file's data resides. Once it has ownership of the necessary allocation units, it updates their metadata to reflect the space freed by deleting the file.

Figure 9-3 Deleting a CFS file



CFS then commits and writes the intent log transaction. Committing the delete transaction enables the three actions illustrated in Figure 9-3 to be performed asynchronously:

- **Directory update**. CFS overwrites the directory block with the file's entry marked "deleted"
- **inode status update**. CFS updates the inode's disk image to indicate that it is not in use, and flags the inode as unused in the inode table

- **Allocation unit metadata update**. CFS updates the metadata for allocation units in which file system blocks were freed by the deletion

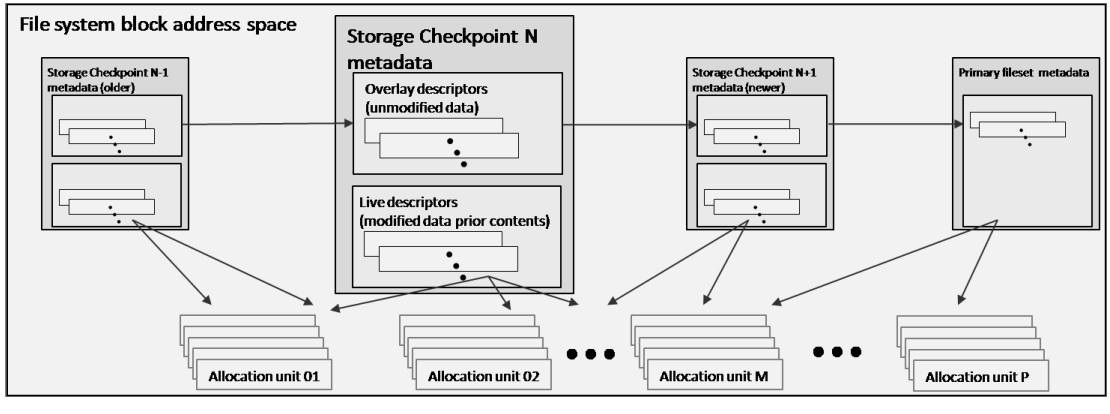
These operations occur asynchronously with each other, but they must all complete before CFS signals completion to the application, no matter which logging mount option is in effect for the file system.

CFS Storage Checkpoint structure

An important feature of CFS is its ability to create Storage Checkpoints—point-in-time space-optimized snapshots of file system contents that can be mounted and accessed as if they were file systems, including being updated by applications. Storage Checkpoints are useful as source data for backups and data analysis, for development and testing, and so forth.

With any copy-on-write snapshot technology, snapshot metadata and data are necessarily bound to that of the primary fileset, and they therefore have an impact on primary fileset I/O. The three examples that follow illustrate three important Storage Checkpoint operations—creation, deletion, and writing to the primary fileset when one or more Storage Checkpoints are active. As a preamble to the discussions of I/O operation flow in the presence of Storage Checkpoints, Figure 9-4 illustrates the main principle of Storage Checkpoint metadata organization.

Figure 9-4 Storage Checkpoint metadata linkage



As Figure 9-4 suggests, the metadata structures that describe Storage Checkpoints form a time-ordered chain with the file system's primary fileset at the head. Each time a new checkpoint is created, it is inserted into the chain between the primary fileset and the prior newest link.

Storage Checkpoints are described by two types of metadata:

- **Overlay descriptors**. These describe data and metadata that have not changed since the checkpoint was created. They point to corresponding metadata (for example, extent descriptors) in the next (newer) element in the chain
- **Live descriptors**. These describe data and metadata that have altered since the creation of the checkpoint. These point to file system blocks in which the prior state of the altered data and metadata are stored. A file system's primary fileset only contains live metadata descriptors

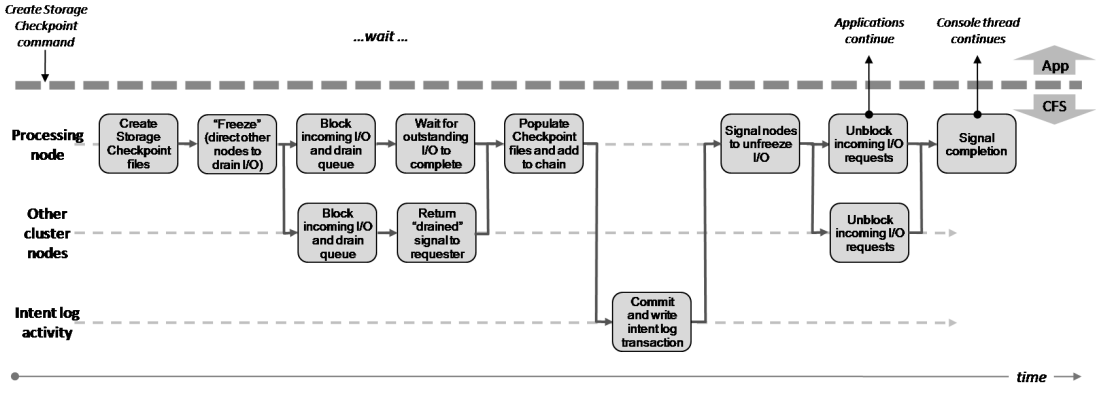
With this structure, the actual storage consumed by any given checkpoint amounts to primary fileset data and metadata that changed while the checkpoint was the newest active one. As soon as a newer checkpoint is taken, the contents of the older one are effectively frozen; prior contents of any primary fileset blocks that change thereafter are linked to the newer Storage Checkpoint's metadata. Thus, ignoring the per-Storage Checkpoint metadata itself, the total amount of storage consumed by all of a file system's active checkpoints is equal to the amount of change in the primary fileset during the lifetime of the oldest active checkpoint. (The situation is slightly different for writable *clones*, which are based on the Storage Checkpoint technology, but which can be updated independently of their primary filesets. The Symantec Yellow Book Using Local Copy Services, available at no cost on the Symantec web site,[20] gives an exhaustive description of the data structures that support Storage Checkpoints and file system clones.)

Creating a CFS Storage Checkpoint

Storage Checkpoints are copy-on-write snapshots. They consume storage only in proportion to the amount of primary fileset data that is modified while they are active. For unmodified data, they point to primary fileset extent contents. Storage Checkpoints occupy storage in a file system's space pool. Figure 9-5 illustrates a (necessarily compressed) Storage Checkpoint creation timeline.

20. http://eval.symantec.com/mktginfo/enterprise/yellowbooks/using_local_copy_services_03_2006.en-us.pdf

Figure 9-5 Creating a Storage Checkpoint



A CFS instance begins Storage Checkpoint creation creating the metadata files for the checkpoint and temporarily marking them for removal. It then "freezes" the file system, blocking incoming I/O requests and requesting that other instances do the same. It waits until all I/O operations in progress on all instances have completed.

When the file system is frozen, CFS populates the new checkpoint's fileset with metadata and inserts the fileset into an age-ordered chain headed by the primary fileset and having a link for each active Storage Checkpoint. When a new Storage Checkpoint is created, old data resulting from subsequent modifications to the primary fileset are recorded in it. all older checkpoints refer forward to it if necessary when applications access data in them. Thus, the amount of storage consumed by all Storage Checkpoints is approximately the number of file system blocks occupied by all primary fileset data modified since creation of the oldest Storage Checkpoint.

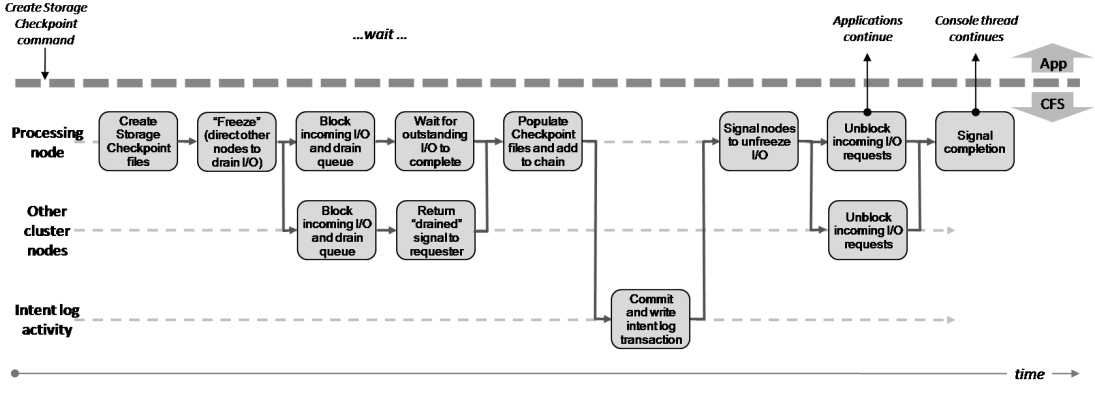
Storage Checkpoints contain two types of metadata—pointers to file system blocks and so-called "overlay" metadata that points to the next newer checkpoint in the chain (or, for the newest Storage Checkpoint, the primary fileset). When CFS uses Storage Checkpoint metadata to locate data, items flagged as overlays cause it to refer to the next newer checkpoint in the chain. If the corresponding metadata in that checkpoint is also an overlay, CFS refers to the next newer one, until it reaches the primary fileset (indicating that the data item was unmodified since the creation of the oldest checkpoint). CFS accesses such data items from the file system's primary fileset (live data) when they are read in the context of a Storage Checkpoint.

When one or more Storage Checkpoints are active, application writes to a file system's primary fileset are performed as outlined in the following section.

Writing data while a Storage Checkpoint is active

CFS Storage Checkpoints are *space-optimized*—the only storage space they consume is that used to save the prior contents of primary fileset blocks that applications modify while they are active. The technique is called "copy-on-write"—upon an application request that modifies file system data or metadata, CFS allocates file system blocks from the file system's space pool, copies the data or metadata to be overwritten into them, and links them to the Storage Checkpoint's metadata structures. Figure 9-6 presents a condensed timeline for an application write to a file system with an active Storage Checkpoint.

Figure 9-6 Writing data while a Storage Checkpoint is active



In order to focus on the main points of writing to a CFS file system in the presence of one or more Storage Checkpoint, Figure 9-6 omits the detail related to allocation unit delegation, mount options, and cache advisories that is shown in Figure 9-2. Essentially the same operations occur when Storage Checkpoints are active, however.

The first write to a given range of file system blocks after a given Storage Checkpoint is taken results in copy-on-write. Subsequent updates to the same file system block range are executed identically, whether Storage Checkpoints are active or not.

CFS first executes a transaction to allocate storage for the prior contents of the blocks to be updated. It then reads the data to be overwritten and copies it to the allocated blocks. Its third step is to copy the new data from application buffers to pages that it allocates from operating system page cache. (It bypasses this step and writes directly from application buffers if the **VX_DIRECT** or **VX_UNBUFFERED** cache advisory or the corresponding **convosync** or **mincache** mount option (Table 13-2 on page 216) is in effect.) When the prior contents have been safely preserved on disk, it overwrites the vacated file system blocks with the new application data.

CFS then updates Storage Checkpoint metadata to point to the newly allocated file system blocks that contain the prior contents of the overwritten ones in the primary fileset. Because copy-on-write occurs upon the first modification of the affected file system blocks, CFS must convert the Storage Checkpoint's overlay extent descriptors (and possibly other metadata) that refer forward to primary fileset data into live extent descriptors that point to the newly-allocated file system blocks that now contain the prior contents of the updated blocks.

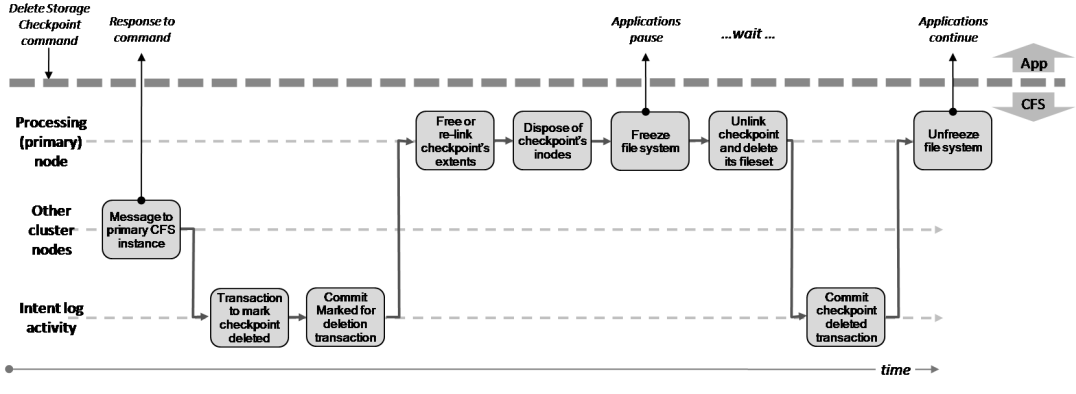
Finally, CFS completes the operation by executing an intent log transaction that captures the primary fileset inode update (change in modification and access times) and the update to the Storage Checkpoint inode (extent descriptor change, but no access time changes).

Similar scenarios occur when CFS truncates files and so forth.

Deleting a CFS Storage Checkpoint

When CFS deletes a Storage Checkpoint, the file system blocks it occupies must either be freed or linked to more recent Storage Checkpoints that remain active, and its inodes must be disposed of properly. Figure 9-7 illustrates the sequence of events in the deletion of a Storage Checkpoint.

Figure 9-7 Deleting a CFS Storage Checkpoint



A CFS instance that receives a command to delete a Storage Checkpoint sends a message to the file system's primary instance, which performs all deletions. The primary instance creates and commits an intent log transaction in which it marks the checkpoint deleted. Once this transaction is committed, the primary instance starts a background thread to dispose of the checkpoint's file data extents and inodes.

Extents containing data from files that were modified, truncated, or deleted are transferred to the next newer checkpoint in the chain. Extents that contain data written to this checkpoint are released to the free space pool. As the background

thread processes inodes, it marks them as “pass-through” so that other threads ignore them when locating data to satisfy user requests made against Storage Checkpoints.

When the background thread has processed all of the Storage Checkpoint’s inodes, it freezes the file system momentarily, and creates an intent log transaction that removes the checkpoint from the file system’s checkpoint chain and frees the space occupied by its structural files. When this transaction is committed, the thread unfreezes the file system, and application activity against it resumes.

Chapter 10

CFS Differentiator: multi-volume file systems and dynamic storage tiering

This chapter includes the following topics:

Most enterprise data has a natural lifecycle with periods of high and low activity and criticality. Recognizing that there can be a substantial differential in cost per terabyte (4:1 or more) between enterprise-class high-performance disk array storage and high-capacity storage of lesser performance, some data centers adopt *multi-tier* strategies to lower the average cost of keeping data online by moving data sets between storage tiers at different points in their life cycles.

Lower blended storage cost through multi-tiering

Typically, a relatively small percentage of a mature application's data is active. Current business transactions, designs, documents, media clips and so forth are all likely to be accessed frequently. As a business or project activity diminishes, its data is accessed less frequently, but remains valuable to keep online. As applications age, it is fair to say that the majority of their data is seldom accessed. An enterprise can reduce its average storage cost by relocating its inactive data to less expensive bulk storage devices without losing the advantages of having it online and readily accessible. Since the relocated data is accessed infrequently, the lesser performance of the bulk storage devices has little or no impact on overall application performance.

As an example, Table 10-1 presents a *pro forma* cost savings calculation for an application with 100 terabytes of data, 20% of which is active, and 80% of which

is idle and could be moved to low-cost bulk storage without significantly impacting overall application performance.

Table 10-1 Pro forma example of cost savings with a two-tier storage strategy (100 terabytes

	Storage cost per terabyte	Cost of storage	Storage cost per terabyte	Savings
Single-tier strategy	Tier 1: $7,500	Tier 1: $750,000	Tier 1: $7,500	
Two-tier strategy	Tier1: $7,500 Tier 2: $2,000	Tier1: $150,000 Tier 2: $160,000 Total: $310,000	Tier1: $7,500 Tier 2: $2,000	$440,000

It is clear from this admittedly simplistic example that the cost savings from a two-tier storage strategy can be substantial. For small active data sets with very high performance requirements, a solid state primary tier would show even greater savings and better quality of service.

Barriers to storage cost savings

But there is a significant barrier to actually realizing the potential cost savings from multi-tier storage. For a multi-tier storage strategy to be practical, it must not impact application performance negatively; data must reside on the high-performance primary tier while it is actually active, and migrate to lower-cost tiers when it becomes idle.

Keeping a few files on the most appropriate storage tier based on I/O activity is easy enough. An administrator can monitor I/O activity, and move files and adjust application procedures as necessary to optimize storage and I/O resources. The complication arises in data centers with millions of files that fluctuate between high and low I/O activity and greater and lesser criticality. It is impossible for humans to track I/O activity on this scale, let alone relocate files and adjust applications and scripts accordingly. When solid state storage is present, the need to use it appropriately is even more pronounced. Not only must important, active files be placed on it, but as files become activity, they must be relocated away from it to free the high-cost space for other, more active files. The difficulty and expense of matching large numbers of files with changing activity levels leads some data centers to adopt inflexible placement policies (for example, “if it’s 90 days old, relocate it to tier 2, period”), and others to forego multi-tier storage entirely.

CFS Dynamic Storage Tiering (DST)

The Dynamic Storage Tiering (DST) feature of CFS solves the problem of matching large numbers of files to appropriate storage tiers without

administrative intervention. DST completely automates the relocation of any or all of the files in a file system between storage tiers according to a flexible range of administrator-defined *policies*. DST is a "set and forget" facility. Once an administrator has defined a file system's placement parameters and a relocation schedule, file movement is both automatic and transparent to applications.

When creating a CFS file system, an administrator specifies:

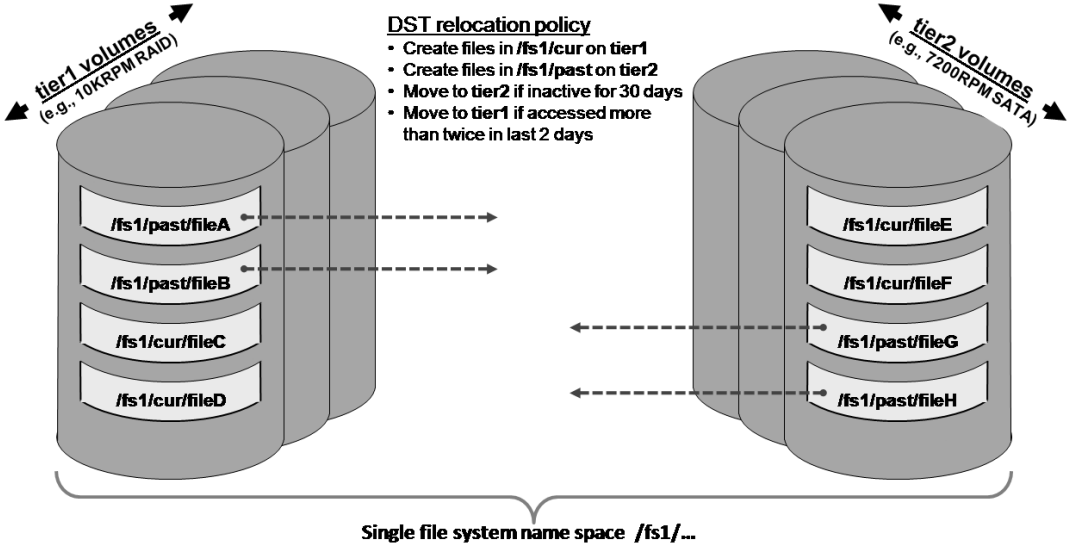
- **Storage**. The CVM volumes that make up each of the file system's storage tiers
- **Tiering policy**. Optional specification of placement and relocation rules for classes of files

From that point on, DST automatically manages file locations based on I/O activity levels or other criteria specified in the placement rules, making proper file placement completely transparent to applications and administrators.

A DST Example

Figure 10-1 illustrates the action of a simple DST placement and relocation policy.

Figure 10-1 Typical DST relocation policy actions



According to the file placement policy illustrated in Figure 10-1, files in the **/cur** directory are created on **tier1** volumes, while those created in the **/past** directory are created on **tier2** volumes. Regardless of where they were created, DST relocates files from **tier1** volumes to **tier2** volumes if they are not accessed

for 30 days (according to the **atime** POSIX metadata). Similarly, if files on **tier2** volumes are accessed more than twice over a two-day period, DST relocates them to **tier1** volumes, regardless of where they were created.

Thus, with no action on the administrator's part, "busy" files automatically move to high-performance storage, and inactive files gradually migrate to inexpensive storage. As migration occurs, files' extent descriptors change to reflect their new locations, but the files' positions in the directory hierarchy does not change, so relocations are completely transparent to applications that access files by path name.

DST advantages

While other technologies for automating data movement between storage tiers exist, DST is unique in four respects:

- **Configuration flexibility**. DST storage tier definitions are completely at the discretion of the administrator. There is no artificial limit to the number of tiers that may be defined, nor to the number or type of CVM volumes that may be assigned to a tier
- **Managed objects**. DST operates at the file level. DST policies act on files rather than volumes, placing the entities that correspond most closely to business objects on whichever type of storage is deemed most appropriate by the administrator
- **Policy flexibility**. DST includes a wide range of file placement and relocation options. While the most frequent application of DST is I/O activity-based relocation, file placement policies can be based on other criteria, such as location in the name space, size, or ownership
- **Application transparency**. DST file relocation is transparent to users and applications. DST moves files between storage tiers as indicated by the policy in effect, but leaves them at their original logical positions in the file system's name space. No adjustment of applications or scripts is necessary

Enabling DST: CFS multi-volume file systems

A CFS file system's VSET may consist of as many as 8,192 CVM volumes. The volumes can be of any type that CVM supports (simple, concatenated, striped, or mirrored) and of any practical capacity.[21] An administrator organizes a file system's volumes into *tiers* by affixing tags to them. The composition of a tier is completely at administrator discretion, but an obvious best practice is for each

21. The theoretical maximum volume size possible with CVM data structures is 18,446,744,073,709,551,104 bytes (2^{64}-512, or 18.5 exabytes). Practical considerations impose significantly smaller limits. CFS limits maximum file system size to 256 terabytes.

tier to consist of identical, or at least similar, volumes. For example, an administrator might assign volumes that consist of LUNs based on Fibre Channel disks mirrored by a disk array to a tier called **tier1**, and volumes consisting of high-capacity SATA disks mirrored by CVM to a tier called **tier2**. Tier names are completely at the discretion of the administrator. For this example, tiers called **gold** and **silver** would serve equally well.

DST policies specify file placement and relocation at the tier rather than the volume level. This gives the administrator the flexibility of expanding or reducing the amount of storage in each file system tier independently.

Administrative hint 22

An administrator can use the `fsmap` console command to determine the volume(s) on which a particular file resides.

File-level placement and relocation

DST acts on the files in a CFS file system at two points in their life cycles:

- **Creation**. When a file is created, DST places its data on a volume chosen by DST that is part of a storage tier specified in an administrator-defined placement policy statement
- **State changes**. When the state of a file changes (for example, the file grows beyond a size threshold, is renamed, is not accessed for some period of time), DST automatically relocates it to a volume (chosen by DST) in another tier as specified in an administrator-defined relocation policy statement

Each CFS file system may have a single DST policy associated with it at any point in time. A policy consists of an unlimited number of *policy rules*. Each rule specifies:

- **Applicability**. The files to which it applies
- **Initial placement**. The storage tier in which space for files to which the rule applies is to be allocated when the files are first written
- **Relocation criteria**. Criteria for relocating the files to which the rule applies to other tiers

Continuing with the two-tier example of the preceding section, an administrator might define a DST policy with two rules, one pertaining to **jpg** and **png** files and the other to all other files in the file system's name space. The rules might specify placement for new files as:

- **Rule 1**. Space for new **jpg** and **png** graphic images is to be allocated on **tier2** volumes
- **Rule 2**. Space for new **doc** files is to be allocated on **tier1** volumes
- **Rule 3**. Space for all other new files is to be allocated on a **tier2** volume

In addition to initial file placement, DST policy rules specify criteria for automatic file relocation. For example, the two-tier file system might specify the following relocations:

- **Rule 1**. Never relocate **jpg** and **png** graphic files to **tier1** volumes (implicitly specified by the absence of relocation statements in **Rule 1**)
- **Rule 2a**. Relocate **doc** files on **tier1** volumes that are not accessed for 10 days to **tier2** volumes
- **Rule 2b**. Relocate **doc** files on **tier2** volumes that are accessed more than five times in one day to **tier1** volumes
- **Rule 3a**. Relocate all other files on **tier1** volumes that are not accessed for 30 days **tier2** volumes
- **Rule 3b**. Relocate all files other than **doc**, **jpg**, and **png** on **tier2** volumes that are accessed more than 10 times in one day to **tier1** volumes

An administrator can schedule the DST *sweeper* to scan an entire file system name space periodically (for example, on a daily basis), or execute it on demand. The sweeper relocates files that meet the file system's relocation policy criteria to volumes in the specified tiers. Because solid state device-based volumes are typically much smaller than disk-based ones, DST can scan them selectively on a more frequent basis (e.g., hourly) to identify and relocate inactive files on them, which it then replaces by more active ones.

An administrator can specify a DST policy in either of two ways:

- **Graphically**. The *Veritas Enterprise Administrator* (VEA) includes a graphical interface for specifying the parameters of certain pre-defined DST policy types. VEA automatically assigns policies to file systems as they are created or edited
- **Direct editing**. Administrators can extract policy source files written in XML and edit them to reflect required policy changes. This technique makes the full generality of the DST policy structure available, but requires a basic knowledge of XML and the DST grammar, described in the book *Using Dynamic Storage Tiering*, which can be downloaded at no cost from www.symantec.com/yellowbooks

Administrative hint 23

Administrators use the **fsppadm** command to **assign**, **unassign**, **dump**, and otherwise manage DST file placement and relocation policies.

Administrators can change a file system's DST policy by dumping its policy file, editing it, and reassigning the edited file to the file system.

DST policy rule ordering

What sets DST apart from other data placement automation schemes is flexibility. DST policies can range from the extremely simple ("create files on **tier1** volumes; move any that aren't accessed for 30 days to **tier2**") to highly sophisticated, including relocation up and down a multi-tier storage hierarchy, with policy rule granularity down to the directory, and even file level.

DST policy rules are specified in an XML file created by the administrator. As described in a preceding section, a DST policy rule has three elements:

- **Applicability**. The files to which it applies
- **Initial placement**. The storage tier in which space for files to which the rule applies is to be allocated when the files are first written
- **Relocation criteria**. Criteria for relocating the files to which the rule applies to other tiers

The order in which rules are specified in a DST policy file is significant. When allocating space for a new file or relocating an existing one, DST scans its policy rules in the order in which they occur in the policy source file, and acts in accordance with the first rule in the sequence that applies to the file upon which it is operating. Thus, if a policy contains more than one rule that applies to a given file, only the rule that appears first in the policy source file is effective. For example, a DST policy might contain three rules that apply to:

- **Location in the name space**. Files in directory **/home/user1**
- **File type**. Files of type **jpg** or **png**
- **User and/or group**. Files owned by userID **[100,100]**

With this scenario, **jpg** and **png** files in directory **/home/user1** would never be subjected to the second rule, because DST would always act on them in accordance with the first rule. Similarly, **jpg** and **png** files owned by userID **[100,100]** would never be subjected to the third rule, because DST would act on them in accordance with the second rule.

If a file's properties change, for example, if it is extended, renamed, moved to an alternate directory, or changes ownership, it may become subject to a different DST rule. Changes in the policy rules to which a file is subject take effect only during file system relocation scans.

DST policy rule selection criteria

As suggested in the preceding section, DST policy rules can be applied to files selectively based on:

- **Location in the name space**. The directory subtrees in which files reside

- **File name pattern**. The pattern of the files' names (for example, **`*.jpg`** or **`*.dat`**)
- **File ownership**. The file owners userIDs and/or groupIDs
- **File tags**. Administrators and users can "tag" files with arbitrary character strings that are visible to DST

DST uses these criteria to select files to which to apply rules. Within each rule there are further selection criteria based on files' states—for example, size, time of last access or update, and so forth—that apply specifically to relocation. A DST policy rule may contain a *relocation clause* specifying that files to which the rule applies be relocated if they meet certain criteria:

- **Current location**. They are located on a storage tier that requires pruning. For example, in a three-tier system, files residing in **tier1** may be subject to relocation, while those in **tier2** are not
- **Most recent access**. They have not been accessed or modified for a specified period (based on their POSIX **atime** or **mtime** parameters)
- **Size**. They have grown or been truncated beyond a threshold
- **Activity level**. Their *I/O temperature*, or rate at which they have been read, written, or both during a defined interval, has exceeded or dropped below a threshold

Any or all of these selection criteria may be specified in a single policy rule. If a file that meets the selection criteria for the rule as a whole also meets one of the relocation criteria, DST relocates it to one of the tiers specified in the rule.

File placement and relocation flexibility

A potential shortcoming of any multi-tier storage solution is that one tier may fill to capacity, causing allocations to fail, while unused space remains in other tiers. In some situations, this may be the desired behavior, whereas in others, "spilling over" to an adjacent tier would be more appropriate.

DST places the decision about whether to confine file placement and relocation to a single tier or whether to allow placement in other tiers if the preferred one is full in the hands of the administrator. An policy rule may specify either a single tier or a list of tiers as a file placement or relocation destination. If a single tier is specified, DST only attempts initial placement on or relocation to volumes in that tier. If multiple tiers are specified, DST attempts to allocate or relocate qualifying files in the tiers in the listed order.

When placing and relocating files, DST works entirely with tiers. It is not possible to designate specific volumes directly as sources or destinations for file placement and relocation. This allows the administrator to expand the capacity of individual storage tiers independently of each other, by adding volumes to them. If placement on and relocation to specific volumes is desirable, it can be

achieved by making each volume a separate tier.

For storage tiers that contain multiple volumes, the DST *balance size* option distributes the extents of each qualifying file across the volumes in a tier. When a policy rule specifies a balance size for a tier, DST places and relocates files into the tier in extents of the balance size, which it distributes randomly among the tier's volumes. Specifying a balance size has an effect similar to striping, except that the "stripes" are randomly distributed among the tier's volumes rather than in a regular geometric pattern. Balancing is particularly beneficial for transactional workloads that are characterized by a high frequency of relatively small reads and writes.

Application transparency

Applications express file read and write requests in terms of data addresses in a file address space of logically contiguous file blocks. CFS extent descriptors map ranges of file blocks to file system block addresses that consist of volume indexes and file system block number within the volume. When DST relocates a file, it copies the data in each extent to a volume in the destination tier, and updates the extent descriptor in the inode with the new volume index and file system block number. This design makes it possible for DST to relocate files in a way that is transparent to applications that open files by specifying their pathnames. No matter which volume(s) a file's data resides on, its logical position in the file system directory hierarchy remains the same.

Additional capabilities enabled by multi-volume file systems

In addition to automatic transparent file placement, multi-volume file systems and DST provide two other noteworthy capabilities:

- **Metadata isolation**. When adding a volume to a CFS file system's VSET an administrator can specify that it be a data-only volume, on which CFS is not to store any metadata. The first volume in a VSET must be eligible to contain metadata, but any additional volumes can be specified as solely for file data storage.
  A multi-volume CFS file system can remain mounted as long as all of its metadata-eligible volumes are present; any data-only volumes may be missing (for example, failed). I/O to files whose data is on missing data-only volumes will fail, but the file system as a whole can function

Administrative hint 24

Because CFS forces certain types of critical metadata to the first volume in a VSET, It is a best practice to specify a highly reliable, high-performance volume as the first volume, and not name it in any DST placement or relocation rules. This effectively isolates metadata from file data and permits business-based storage choices of file data storage technology.

- **Intent log isolation**. Because CFS file system performance is strongly influenced by intent log performance, it can be useful to isolate intent logs on small, high-performance volumes. As with metadata isolation, an administrator can add volumes to a file system's VSET and not specify it as a placement or relocation destination in any DST policy rule. File system instances' intent logs can be placed on these volumes, keeping them isolated from file data I/O activity

Administrative hint 25

The first time a file system is mounted on a cluster node, CFS creates its intent log structural file. Administrators can add dedicated volumes to a file system's VSET, and use the **fsadm** command with the **logvol** option to move instances' intent log files to the volumes.

Chapter 11

CFS Differentiator: database management system accelerators

This chapter includes the following topics:

CFS is used frequently as a storage substrate for relational databases. Database management systems manage their own storage at the block-level, and most can make direct use of virtual volumes or even disks. When they do use files as data storage, they typically treat each file as a disk-like container, and manage its contents internally.

Using files as storage containers has several important advantages for database management software and for database administrators:

- **Allocation flexibility**. Files can be created, expanded, truncated, and deleted at will, without recourse to a system or storage administrator. Moreover, because CFS files can be extended automatically, database administrators can allocate small database container files to start with, and let CFS extend them as actually required, rather than having to predict storage requirements
- **Administrative flexibility**. Files can be copied from one device to another much more simply than virtual volumes or disks. Again, database administrators can usually manage storage without involving system or storage administration
- **Data protection flexibility**. Database management systems generally include tools for backing up database objects. Many users find it more convenient to

back up the underlying storage, to reduce backup times, or to integrate more fully with overall data center practice. Using files as database storage opens the possibility of using file-based data protection methods, such as CFS Storage Checkpoints coupled with Symantec's NetBackup, to protect database data

About the only factor that has inhibited database administrators from specifying file-level storage for their databases in the past is I/O performance. Because file systems are designed for concurrent use by multiple applications and users, they necessarily include extensive mechanisms to make it appear to each client that it is the file system's only user. The principal mechanisms are:

- **Data copying**. Most file systems by default execute application read requests by reading data into operating system page cache and copying it to application buffers. Similarly, they execute write requests by copying data from application buffers to operating system page cache and writing it from there. This minimizes the constraints placed on application buffers, but consumes "extra" memory and processing power compared to transferring data directly between application buffers and storage devices
- **Write serialization**. Most UNIX file systems serialize application write requests by default. If a write request arrives while a previous one is being executed, the second request waits until execution of the first is complete. This simple constraint satisfies the POSIX rule of presenting only the results of complete writes to subsequent readers, but means that (a) only a single write to a file is in progress at any instant, and (b) all reads to the file are serialized behind any outstanding write

Because database management systems tend to be I/O intensive, data copying and write serialization can seriously impact the performance they deliver to their clients.

When they use file systems to store data, however, database management systems are the only user, so file system mechanisms that make multiple users transparent to each other are unnecessary. In recognition of this, CFS includes two mechanisms that permit database management systems (and other applications that manage their own concurrent I/O operations) to bypass unnecessary file system protection mechanisms and take control of their own I/O scheduling and buffering, as if they were operating directly on "raw" disks:

- **Oracle Disk Manager**. For Oracle databases, all CFS versions include a run-time library that implements Oracle's Oracle Disk Manager (ODM) APIs. For legacy applications and databases, CFS also continues to include the native *Quick I/O for Databases* feature from which the ODM library evolved
- **Concurrent I/O**. For database management systems, whose vendors do not implement private API specifications like ODM, CFS includes a *concurrent I/O* (CIO) facility that eliminates in-memory data movement and increases parallel I/O capability

In essence, these capabilities allow database management systems to treat CFS

files as disk-like containers to which they make asynchronous I/O requests. CFS bypasses file write locking and transfers data directly to and from database management system buffers. Thus, CFS provides database management systems with raw disk I/O performance and file system administrative convenience.

The Oracle Disk Manager (ODM)

Oracle Corporation publishes an API specification called the Oracle Disk Manager (ODM) for its Oracle and Real Application Cluster (RAC) cluster database management software. Newer Oracle database management software versions invoke ODM APIs to perform storage-related functions. The underlying storage infrastructure implements the functionality expressed in the APIs. For Oracle developers and database administrators, ODM provides consistent, predictable, database behavior that is portable among different storage infrastructures.

CFS includes an ODM library that uses CVM and CFS capabilities to implement the ODM API functions. CFS's ODM library is an evolution of an earlier, and still supported, Storage Foundation capability called Quick I/O for databases (QIO), discussed in "Quick I/O for Databases" on page 186.

Using the CFS ODM library enhances Oracle I/O performance in three ways:

- **Asynchronous I/O**. Oracle threads are able to issue I/O requests and continue executing without waiting for them to complete
- **Direct I/O**. Data is transferred directly to and from Oracle's own buffers. When ODM is in use, CFS does not copy data to operating system page cache before writing it to disk, nor does it execute Oracle's read requests by reading data from disk storage into page cache and copy it to Oracle's own cache
- **Write lock avoidance**. Oracle's writes bypass operating system file write locking mechanisms. This increases parallel execution by allowing multiple requests to pass through to CVM and thence to the hardware I/O driver level

These optimizations are possible with Oracle, because Oracle itself ensures that it does not issue potentially conflicting I/O commands concurrently, or reuse buffers before I/O is complete. CFS's GLM locking (Chapter 8) comes into play only when file metadata changes, for example when an administrator resizes database container files or creates new ones.

The CFS ODM library is cluster-aware. Instances of it run in all nodes of a VCS cluster and communicate with each other to maintain the structural integrity of database container files and to keep database storage administration simple. For example, before creating a new data file in response to a request from Oracle, an ODM instance queries other instances to verify that the file name is unique throughout the cluster.

Volume resilvering with ODM

One important feature of the CFS ODM library that is especially significant is that it enables Oracle to *resilver*[22] a mirrored volume after system crash.

It is possible that writes to a volume mirrored by CVM may have been in progress at the time of a failure. The contents of the disks that make up mirrored volumes may be inconsistent for either of two reasons:

- **Incomplete writes**. A multi-sector write may have been interrupted while in progress. Disks (and disk array LUNs) generally finish writing the sector in progress when power fails, but do not guarantee to complete a multi-sector write. After the failure, a multi-sector Oracle block may be "torn"—containing partly old and partly new content
- **Unprocessed writes**. Writes to some of the disks of a mirrored volume may not have been executed at all at the instant of failure. After the failure, all mirrors will contain syntactically valid Oracle blocks, but some mirrors' block contents may be out of date

Normally, CVM would alter its read algorithms during recovery to insure that all mirrors of a mirrored volume contain identical contents. ODM overrides this mode of operation, since it has more precise knowledge of which file blocks might be at risk.

Oracle uses leading and trailing checksums on its data blocks to detect torn blocks after recovery from a failure. To detect unprocessed writes to mirrored volumes, it uses an I/O sequence number called the *system control number* (SCN) that is stored in multiple locations in a database. When Oracle detects either of these conditions in a database block, it uses ODM APIs to request a re-read of the block from a different mirror of the volume. If the re-read content is verifiable, Oracle uses the ODM API to overwrite the incomplete or out-of-date content in the original mirror, making the database block consistent across the volume.

ODM advantages

ODM library instances communicate with each other to coordinate file management. This enables Oracle itself to manage the creation and naming of data, control, and log files by specifying parameters in a database's Oracle initialization file, a feature referred to as Oracle-Managed Files (OMF). OMF also supports automatic deletion of data files when a database administrator removes the tablespaces that occupy them.

CFS adds high availability, scalability, and centralized management to the VxFS file system on which it is based. A CFS storage infrastructure enhances Oracle

22. The *resilvering* metaphor is apt. After a failure that may leave a mirror tarnished, resilvering restores its perfectly reflective quality.

storage in the following ways:

- **Superior manageability**. Without a file system, Oracle uses disk partitions as data storage containers. Compared to CFS container files, disk partitions are inflexible to configure and resize, typically requiring coordination between the database administrator and a storage or system administrator. With CFS, files, and indeed entire file systems can be resized dynamically as requirements dictate
- **Less susceptibility to administrator error**. System administrators cannot readily determine that partitions are being used by Oracle, and may therefore mistakenly format file systems over database data. Using CFS for database container file storage eliminates this possibility
- **Flexible data protection options**. Databases that use disk partitions for storage are limited to using the database management system vendor's backup mechanism. Using CFS makes it possible to take snapshots of and back up database container files using tools and techniques that are used for non-database data elsewhere in the data center

Cached ODM

When ODM is in use, CFS normally bypasses the file system cache and writes and reads directly to and from disk. The newest releases of CFS include a *Cached ODM* feature, which can improve ODM I/O performance. With Cached ODM, CFS caches data read by Oracle conditionally, based on hints given in I/O requests that indicate what use Oracle expects to make of the data. The CFS ODM library uses these hints to determine whether to enable caching and read ahead or to read directly into Oracle's buffers, as would the non-cached ODM mount option.

Administrators configure Cached ODM at two levels of granularity:

- **File system**. Using the **vxtunefs** command, an administrator can enable or disable Cached ODM for an entire file system
- **Per-file**. Using the **odmadm** command, an administrator can enable or disable Cached ODM for individual files within a file system, or alternatively, specify that I/O to the file should follow the current Cached ODM setting for the file system

By default, Cached ODM is disabled when CFS file systems are mounted. Unless Cached ODM is enabled, CFS ignores cache hints that Oracle passes to it in I/O commands. Administrators enable or disable Cached ODM for an entire file system by using the **vxtunefs** command to set the **odm_cache_enable** advisory after the file system is mounted.

Administrators set cached ODM options for individual files in a *cachemap* that CFS creates when Cached ODM is enabled for the file system in any form. The cachemap specifies caching advisories for file type and I/O type combinations. Administrators use the **odmadm setcachefile** command to specify Cached ODM

behavior for individual files:

- **ON**. When Cached ODM is set to **ON** for a file, CFS caches all I/O to the file regardless of Oracle's hints. Disabling Cached ODM for the entire file system overrides this option
- **OFF**. When Cached ODM is set to **OFF** for a file, CFS does not cache any I/O to the file, even if Oracle I/O requests hint that it should do so
- **DEF**. The **DEF** (default) setting for a file causes CFS to follow the caching hints in Oracle's I/O requests. Disabling Cached ODM for the entire file system overrides this option

CFS cachemaps are not persistent across file system mounts. To make per-file Cached ODM behavior persistent, an administrator creates a file called **odmadm** in the **/etc/vx** directory created by the Common Product Installer containing the Cached ODM per-file settings. CFS searches for an **/etc/vx/odmadm** file when it mounts a file system, and if one is found, uses the advisories in it to populate the file system's cachemap.

Administrative hint 26

Administrators should consult the man pages for the **setcachemap** and **setcachefile** commands for the syntax of specifications in the **odmadm** file.

Quick I/O for Databases

The CFS Quick I/O for databases (QIO) feature provides advantages similar to those of ODM for any database management system that coordinates its I/O request synchronization and buffer usage internally. QIO pre-dates Oracle's publication of the ODM specification, and while it continues to be supported, is gradually being supplanted by ODM in Oracle environments. It remains a viable option for other database management systems, however, the more modern Concurrent I/O (CIO) (page 189) option is generally preferable.

Administrative hint 27

Applications in which CFS is used as the storage substrate for Oracle databases should by upgraded from using Quick I/O to using the Oracle Disk Manager library, for improved integration with the Oracle database management system.

With QIO, database management systems access preallocated CFS files as raw character devices, while operating system utilities and other applications move, copy, and back them up as files. The result is the administrative benefits of using files as database data containers without the performance degradation associated file systems designed for concurrent use by multiple applications.

Quick I/O uses a special naming convention to identify files so that database managers can access them as raw character devices.

Quick I/O provides higher database performance in the following ways:

- **Bypassing UNIX kernel serialization**. Database managers schedule their I/O requests to avoid simultaneous I/O to overlapping data areas. This renders the file access locking done by UNIX and Linux kernels unnecessary. Quick I/O uses UNIX *kernel asynchronous I/O* (KAIO) to bypass kernel file locking and issue I/O requests directly to the raw Quick I/O device
- **Bypassing data movement**. Quick I/O uses the CFS direct I/O capability when it issues I/O requests to volumes. With direct I/O, CFS writes data directly from or reads it directly into application buffers, bypassing the usual copying from application buffers to kernel buffers. This works well for database managers, which typically coordinate access to large areas of memory dedicated as cache for the data they manage

Kernel asynchronous I/O

Some operating systems support asynchronous I/O to block-level devices, but not to files. On these platforms, the operating system kernel locks access to file while writes are outstanding to them. Database managers that use container files as storage cannot optimize performance by issuing asynchronous I/O requests to files on these platforms. Quick I/O bypasses these kernel locks and allows database managers to make asynchronous I/O requests to files accessed via the Quick I/O raw device interface.

Kernel write lock avoidance

Each POSIX **`write()`** system call to a file locks access to the file until the I/O is complete, thus blocking other writes, even if they are non-overlapping. Write serialization is unnecessary for database management systems, which coordinate I/O so that concurrent overlapping writes do not occur. The Quick I/O raw device interface bypasses file system locking so that database managers can issue concurrent writes to the same file.

Direct I/O

By default, file data read by means of POSIX **`read()`** system calls is read from disk into operating system page cache and copied to the caller's buffer. Similarly, file data written using the **`write()`** system call is first copied from the caller's buffer to operating system page cache and written to disk from there. Copying data between caller and kernel buffers consumes both CPU and memory resources. In contrast, raw device I/O is done directly to or from the caller's buffers. By presenting a raw device interface, Quick I/O eliminates in-memory copying overhead.

Using Quick I/O

Quick I/O files behave differently from ordinary files in a few key respects:

- **Simultaneous block and file I/O**. A Quick I/O file can be accessed simultaneously as a raw device and a file. For example, a database manager can use a file through its Quick I/O interface, while a backup or other application accesses it via the normal POSIX interface simultaneously.
- **Contiguity requirement**. Quick I/O must consist of contiguously located disk blocks. Quick I/O cannot be used with sparse files. I/O to a file's Quick I/O interface fails if the request addresses blocks represented by a hole in the file block space.
- **Appending**. A Quick I/O file cannot be extended by an application I/O request that appends data to it. The qiomkfile administrative command must be used to change the size of a Quick I/O file

If the Quick I/O feature is installed, it is enabled by default when a file system is mounted. Files may be accessed by the following ways:

- **POSIX file I/O**. Utility programs can move, extend, copy, and back up a Quick I/O file just as they would any other file
- **Character (raw) device I/O**. Database managers and other applications can perceive a file as a raw character device, to which asynchronous, non-locking, direct I/O is possible

The CFS **qiomkfile** utility creates a a file with preallocated, contiguous disk space, a raw character device whose name is the file name with the character string **::cdev::vxfs** appended, and a symbolic link between the two so that the file can be accessed via the File Device Driver (FDD) built into CFS. The database management system or application does I/O to the character device; operating system and other utilities use the regular file.

Administrative hint 28

Administrators of Oracle databases that continue to utilize Quick I/O should use the **–h** option with the **qiomkfile** utility to create Oracle container files with the correct header size and alignment.

- **Legacy database management systems.** Most 32-bit database management systems can run on 64-bit computers, but are limited to 4 gigabytes of memory addressability. Cached Quick I/O uses the larger memory capacity of a typical 64-bit computer as an extended cache
- **Multi-database hosts**. For computers that host multiple databases, Cached Quick I/O forms a pooled cache resource

On 32-bit computers, for example, a database is limited to a maximum cache size of 4 gigabytes of physical memory (minus operating system, database manager,

and application code and buffer requirements) because that's all that 32 bits can address. For read operations, Cached Quick I/O stores blocks of database data in file system cache. This reduces the number of physical I/O operations required.

On 64-bit systems, where memory addressability is less of a limitation, using the file system caching still increases performance by taking advantage of the read-ahead functionality. To maintain the correct data in the buffer for write operations, Cached Quick I/O keeps the page cache in sync with data written to disk.

The CFS Concurrent I/O feature

For database management systems that do not provide API specifications, as well as for other applications that manage their own multi-threaded I/O, CFS includes a Concurrent I/O (CIO) feature. An administrator can specify CIO as a mount option to cause all files in a file system, except for those for which the option is specifically overridden, to be accessed directly and concurrently, bypassing file system data copying and write locking. Alternatively, software developers can use **ioctl** system calls specifying the **VX_CONCURRENT** cache advisory to enable and disable CIO for individual files.

When CIO is enabled for a file, write requests cause CFS to acquire shared locks, rather than exclusive ones. This allows multiple application read and write requests to the file to execute concurrently. The presumption is that applications coordinate their accesses to data internally so that data is not corrupted by overlapping writes, and so that reads to not return the results of partial updates. As with other forms of direct I/O, CFS requires that application buffer memory addresses be aligned on disk sector-size boundaries.[23] If application buffers are not sector size-aligned, CFS buffers the I/O. CIO applies only to application read and write requests; other requests obey the customary POSIX semantics.

23. For example, if disk sector size is 512 bytes, application I/O buffers' starting memory byte addresses must be multiples of 512.

Part III

Installing and configuring CFS

- Installing and configuring CFS
- Tuning CFS file systems

Chapter 12

Installing and configuring CFS

This chapter includes the following topics:

CFS operates in conjunction with several other Storage Foundation software components:

- **Cluster Volume Manager (CVM)**. CVM instantiates the shared volumes used for file storage
- **Veritas Cluster Server (VCS)**. VCS provides monitoring and failover services for CFS. In the SFCFS-HA, SFSYBCE, SFRAC and SFCFSRAC bundles, VCS also provides failover services for database management systems and for applications that use shared file systems

All three of these components must be installed and configured in order for applications and database managers to access shared file systems. Additional installations may be required, for example, if CFS is part of an SFRAC (Storage Foundation for Real Application Cluster) Oracle installation. Finally, installation of SFCFS, SFCFS-HA, SFSCE, or SFRAC may require coordination with other Symantec products, such as Symantec Security Services and Storage Foundation Manager, if those are in use in the data center.

The Storage Foundation Common Product Installer

Storage Foundation products generally consist of multiple installable packages. The interactive *Common Product Installer* included with all Storage Foundation products simplifies installation and initial configuration of products and bundles to the greatest extent possible. The Common Product Installer does the following:

- **Identifies the task**. Prompts the administrator to determine which Storage Foundation bundles or products are to be installed
- **Gathers information**. Prompts the administrator for information required to install the selected products or bundles
- **Installs**. Selects and installs the individual packages that make up the products or product bundles
- **Configures**. Creates and populates configuration files required by the installed products

For the most part, administrators that install Storage Foundation products and bundle are not concerned with the details of individual product installation and integration.

Best practices for installing Storage Foundation products

Observing a few simple best practices can expedite Storage Foundation installation and configuration. These practices generally fall into one of two areas:

- **Preparation**. Making sure that the systems on which the software is to be installed and the surrounding hardware and software environment are properly prepared for installation
- **Information gathering**. Acquiring the information required during installation, so that installation can proceed uninterrupted from start to finish

Preparing for installation: the Storage Foundation pre-check utility

The Storage Foundation Common Product Installer includes a *pre-check* utility that can be run directly from the installation media. Given a list of the products to be installed, the utility produces a report of the operating system, patch level, and other requirements.

In addition to running the pre-check utility, administrators should refer to the

Release Notes documents included with all Storage Foundation products. Release Notes contain information such as patch requirements and restrictions that is discovered after product content is frozen for release. Administrators should apply all required upgrades and patches for operating systems and other software prior to installing Storage Foundation products.

Preparing for installation: the Veritas Installation Assessment Service

Alternatively, administrators can access the free web-based *VERITAS Installation Assessment Service* (VIAS, found at https:// vias.symantec.com/vias/vias/) to determine the readiness of a designated set of systems for installation of Storage Foundation products.

To use the VIAS service, an administrator first downloads the *VIAS Data Collector* from the Symantec web site. The VIAS Data Collector runs on a single system. It collects hardware and software configuration information from a designated list of target systems and consolidates it into a file. The file is transmitted to Symantec, where its contents are compared against current hardware and software compatibility lists, and a report is generated listing requirements that must be met for a successful installation or upgrade of the designated Storage Foundation products or bundles.

For installations whose operating policies prohibit communication of configuration information outside the organization, the VIAS service includes downloadable checklists that exhaustively specify the prerequisites for successful installation of Storage Foundation products.

The free online VIAS service is the preferable means of verifying the readiness of systems for Storage Foundation product installation, in part because it is simple to operate, but most importantly because Symantec updates VIAS hardware and software compatibility lists dynamically, so the online service always evaluates configuration information against the latest known requirements for successful installation.

Information gathering

Certain information about the data center environment is required to install Storage Foundation products. For example, the Common Product Installer requires data center domain names, VCS cluster names and ID numbers, node names, a default CVM disk group name, and so forth, in order to create accurate configuration files. Some “information” is actually in the form of decisions, for example, which network interfaces are to be used as a cluster’s private network, or whether CVM enclosure-based naming is to be used to identify disk locations. These decisions should be made prior to starting installation to avoid the need for hasty “in-flight” data center policy decisions.

To assure a smooth installation process, administrators should read the installation guides for the products to be installed to determine what information and decisions are required (for the most part, the *Common Product Installer Guide* is adequate for this purpose). Storage Foundation product installation guides can be found on:

- **Distribution media**. All Storage Foundation distribution media contain installation guides for the products on them
- **Symantec Web site**. Installation guides and other documentation are available from http://www.symantec.com/business/support/index.jsp

Downloading documentation is useful for pre-purchase product evaluation, as well as in preparing for installation prior to product delivery.

General installation considerations

Observing a few general installation practices for CFS-related Storage Foundation products simplifies installation and results in reliable and trouble-free cluster operation:

- **Remote installation**. Storage Foundation products can be directly installed on the systems on which they are to run, or they can be installed remotely using secure shell (**ssh**) or remote shell (**rsh**) connections. All Storage Foundation bundles that include CFS (SFCFS, SFHA, SFSCE, and SFRAC) also include VCS, for which **ssh** or **rsh** is an installation requirement. Remote console shells must be configured to operate without passwords during installation. Administrators must be authorized to run **ssh** or **rsh** on the systems from which they install, and must have superuser (**root**) access to the systems on which they install Storage Foundation products
- **Time synchronization**. Both CFS and VCS require that all cluster nodes have a synchronized notion of time. Symantec does not recommend manual time synchronization, because it is difficult to accomplish, fragile, and error-prone. A cluster-wide or data center-wide Network Time Protocol (NTP) service is preferable for this purpose. NTP should be installed in the data center and operating prior to Storage Foundation product installation
- **Cluster and data disk fencing**. VCS requires at least three coordinator disks that support SCSI-3 Persistent Group Reservations (PGR) to resolve cluster partitions that result from failures of the private network. (Alternatively, a *coordinator server* can be configured in place of one of the disks.) If a cluser's data disks are PGR-capable, CVM uses data disk fencing ("Feature 1: Cluster and data disk fencing" on page 33) to protect against data corruption when a partition occurs. Volumes for CFS file systems can be configured from non-PGR disks, but PGR-based data disk fencing is the most trustworthy mechanism for avoiding data corruption if a cluster partitions. If a cluster's

disks are PGR-capable, Symantec strongly recommends that Storage Foundation products be installed with data disk fencing enabled

- **Response files**. During installation, the Common Product Installer generates several files, among them a *response file*, in which all administrator responses to installer queries are captured. An administrator can edit the response file from an installation to change system-specific information and use it as a script to drive subsequent installations. Response files are particularly useful in larger clusters and in data centers with a number of similar clusters
- **Event notifications**. Administrators can configure Storage Foundation products to deliver event notifications to electronic mail accounts, to one or more Simple Network Management Protocol (SNMP) consoles, or to a combination of the two. During installation, administrators supply configuration information for whichever of these services are to be employed (server names or IP addresses, port numbers, electronic mail addresses, and so forth)
- **Symantec Security Services**. In data centers that use Symantec Security Services, administrators can configure Storage Foundation products to use their services for central user authentication and to encrypt inter-system communication traffic. Symantec Security Services must be operating during Storage Foundation installation in order for this option to be elected

Installation overview

The Storage Foundation Common Product Installer insulates administrators from most details of component product installation and basic configuration. When the Common Product Installer completes installation of one of the CFS products (SFCFS, SFHA, SFSCE, and SFRAC), it has configured and verified a cluster, and created service groups for CVM and CFS. When product installation is complete, the administrator next executes the system and application-specific configuration steps:

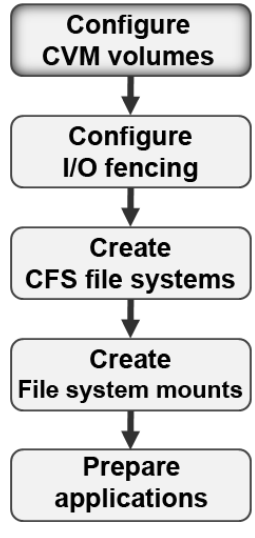
- **Volume configuration**. Adding the disks or LUNs that make up the cluster's pool of shared storage to the CVM default volume group, and create the needed volumes
- **Cluster fencing configuration**. Creating a volume group to contain the LUNs used as cluster fencing coordinator disks and adding three or more suitable LUNs to it (alternatively, coordinator servers can be configured in place of one or more of the coordinator disks)
- **File system creation**. Creating the file systems required by applications that will run on the cluster's nodes with the parameters most suitable to the applications they will serve

- **Mount configuration**. Specifying the mount configuration, volumes that contain shared file systems, the mount points at which they are to be mounted, and the mount parameters (for example, sharing and writability)
- **Application preparation**. Installing applications, configuring them as VCS service groups if necessary, and specifying any required configuration parameters

While the Storage Foundation Common Product Installer does insulate the administrator from most VCS and CVM details, a working knowledge of both products is helpful during initial configuration of a CFS cluster. Similarly, a working knowledge of applications that will co-reside with CFS on cluster nodes helps make their installation and initial configuration trouble-free, particularly if they are to be configured as VCS service groups.

Volume configuration

CFS shared file systems use CVM volumes to store data. Architecturally, CVM volumes are organized identically to those managed by Symantec's (single-host) VxVM volume manager. Each volume is a layered structure built up from *plexes*, which in turn consist of subdisks, or subdivisions of the block address spaces presented by physical disks or LUNs.



The Common Product Installer runs the **cfscluster** utility to perform initial configuration of the CVM cluster service. Once CVM is configured, the administrator uses CVM commands to create the default disk group and any additional disk groups required. For each disk group, the administrator uses the **cfsdgadm** command configure it as a cluster resource. The administrator then *scans* each node's disk I/O interfaces to discover disks and LUNs, and adds each one to a disk group. Disks are added to the default disk group phase if no alternate group is specified;

CVM disk groups are either shared among all cluster nodes or private to a single node. In order for a CFS file system to be shared among cluster nodes, the volumes it occupies must be allocated from a shared disk group. An administrator can use the **cfsdgadm** command to specify different disk group activation modes for different cluster nodes. A cluster node can activate a disk group to share read or write access with other nodes, to be the exclusive writer of volumes in the disk group, or to become a reader and deny all other nodes ability to write to the disk group's volumes. Volume activation modes apply to all volumes in a disk group, and can therefore be used to regulate cluster nodes'

access to all file systems whose volumes are in the disk group.

Once disks and LUNs have been assigned to disk groups, the administrator uses CVM administrative commands to create volumes in anticipation of file system requirements. Best practices for shared volumes are essentially the same as for single-host volumes:

- **Disk grouping**. It is usually advantageous to place all of an application's disks in the same disk group to simplify administration and especially, migration to another system
- **Similar disk types**. For consistent performance and reliability, volumes should normally consist of disks or LUNs of the same, or at least closely similar, type in terms of raw I/O performance and reliability. An exception might be made for solid state disks, which can be mirrored with rotating disks using the CVM *preferred plex* option to direct read requests to the higher performing device
- **Mirrored volume allocation**. While they do offer performance benefits for certain I/O loads, in most cases mirroring and RAID are employed to protect against data loss due to disk failure. For many applications' data, the mirroring or RAID protection provided by disk arrays is adequate. If disk array mirroring is not available, or if the criticality of data warrants protection beyond that offered by a single disk array, CVM can mirror data between two or more disks or LUNs. For maximum protection, the disks that make up a mirrored volume should utilize different physical resources—disks at a minimum, but in addition, separate I/O paths and, in cases where multiple disk arrays are employed, LUNs from different disk arrays
- **Thin provisioning**. If the disk arrays used to store CFS file system data are capable of thin provisioning, they should be configured to make use of the feature. CFS space allocation algorithms are "thin provisioning-friendly" in the sense that they reuse storage space that has been used previously and then deallocated in preference to allocating space that has never been used before. For thin-provisioned disk arrays, this results in fewer provisioning operations and less physical storage consumption.
- **DST storage classes**. When Dynamic Storage Tiering is configured for a file system, CFS transparently moves files between different storage tiers based on criteria like recent activity, location in the name space, owner and group, and so forth. The primary benefit of DST is reduction of overall storage cost with minimal impact on application performance, although its use may also be justified based on other criteria. When creating and tagging volumes that will become

Administrative hint 29

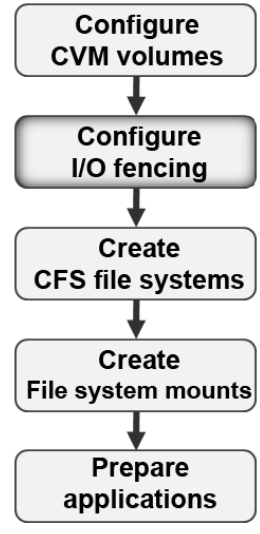
Administrators can use the DST analyzer tool, available at no cost from the Symantec web site, to determine the cost impact of different multi-tier storage strategies on overall file system storage cost.

members of multi-volume file system VSETs, the administrator should ensure that volumes meet the cost, data availability, and I/O performance requirements of the storage tiers of which they will become part

Cluster fencing configuration

VCS requires three or more coordinator disks or coordination servers to enable a single cluster node to unambiguously gain control of a majority in the event of a cluster partition. Three or any higher odd number of coordinator disks or servers are required; all disks must be directly visible to all cluster nodes. For maximum resiliency, each coordinator disk should be presented by a different disk array. CVM does not use coordinator disks to store data, so storage administrators should configure LUNs of the minimum capacity supported by the disk array. LUNs to be used as VCS coordinator disks must support SCSI-3 Persistent Group Reservations (PGR).

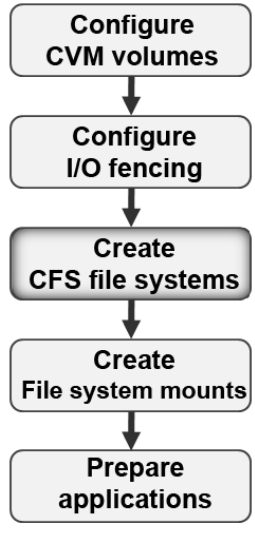
The administrator uses an option of the CVM **vxdg** command to create a dedicated disk group for a cluster's coordinator disks, and adds the coordinator disks to it.



Administrative hint 30

Administrators can use the Storage Foundation **vxfentsthdw** utility to determine whether a given disk or LUN supports SCSI-3 Persistent Group Reservations.

File system creation



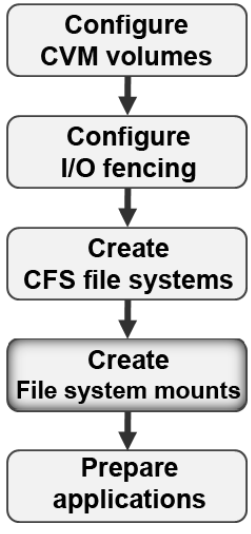
An administrator uses the UNIX **mkfs** command, or alternatively the CFS-specific **mkfs.vxfs** command to create a CFS file system, specifying certain options that can affect the file system's performance. Some creation time options cannot be changed, so administrators should choose them carefully:

- **File system block size**. The atomic unit of CFS space management. CFS supports file system's block sizes of 1, 2, 4, or 8 kilobytes. File systems with larger block sizes can allocate space for large files faster; file systems with smaller block sizes utilize space more efficiently when storing small files. The block size chosen for a file system that will contain database table space files should be the same as the database block size
- **inode size**. CFS persistent inodes are either 256 or 512 bytes in size. The main benefit of larger inodes is that they can hold more access control list entries (But inherited access control lists are stored in separate inodes and linked to files' inodes. Administrators should almost always accept the default 256 byte inode size
- **First volume**. To make storage tiering (Chapter 10) possible, a CFS file system must occupy more than one volume. An administrator can add volumes to or remove them from a file system's volume set (VSET), except for the volume on which the file system is originally created. CFS storage tiering best practice is to create the file system on a resilient, high-performing volume, and specify file placement policy rules that limit it to storing metadata

In addition to these parameters that cannot be changed, an administrator can override the default size of the intent log. A larger intent log can be useful in file systems that are expected to be subject to high frequency metadata activity. Both of these parameters can be changed after file system creation, so they are not as critical as file system block size, inode size, and first volume, which are fixed at file system creation.

During file system creation, CFS queries CVM to determine the geometry of its volume (volume 0), and uses the response to set default values for alignment and sequential read-ahead and write-behind parameters (**read_pref_io**, **read_nstream**, **write_pref_io**, and **write_nstream**, "Sequential read-ahead and write-behind " on page 220). The defaults set by CFS are based on the geometry that CVM reports for the file system's first volume. The administrator can use the **vxtunefs** utility to change these parameter values to optimize for other volumes.

Mount configuration



An administrator mounts each CFS file system either for *shared* or *local* access. File systems mounted in shared mode must use shared CVM volumes for storage (local file systems can use private volumes, or indeed, operating system raw devices, for storage). While mounted as local, a file system cannot be mounted on other cluster nodes.

Administrative hint 31

If a file system is to be mounted with different read-write access on different cluster nodes, the primary mount must include the **crw** option.

File systems that are initially mounted for shared access can be mounted by some or all of a cluster's nodes. A shared file system can be mounted for read-write access on its primary node can be mounted either for read-write or read-only access on other cluster nodes, provided that the crw mount option is specified when it is first mounted. A file system mounted for read-only access on its primary node can only be mounted for read-only access on other nodes. File system mounts whose write access option differs from node to node are called asymmetric mounts.

The CFS instance on the first cluster node to mount a shared file system becomes the file system's primary instance. Other instances that mount the file system become its secondary instances.

Administrative hint 32

CFS reserves the CVM volumes used by shared file systems, and thus protects them from inappropriate access by Storage Foundation administrative commands. There is no similar protection against operating system commands, however. Administrators should use caution with UNIX commands such as **dd** on shared volumes, to avoid corrupting data in shared file systems.

Most CFS file system management functions, including intent logging, space allocation, and lock management are performed by both primary and secondary instances. Certain key functions, including delegation of allocation unit control and Storage Checkpoint creation and deletion, are performed only by a file system's primary instance. In clusters that support multiple CFS file systems, therefore, it is usually advisable to distribute the file systems' primary instance roles among nodes by issuing mount

commands on different nodes.

Obviously, a CFS file system must have a primary instance at all times. If a file system's primary instance fails (for example, because the node on which it is running fails), CFS elects one of the secondaries as the file system's new primary instance.

Administrative hint 33

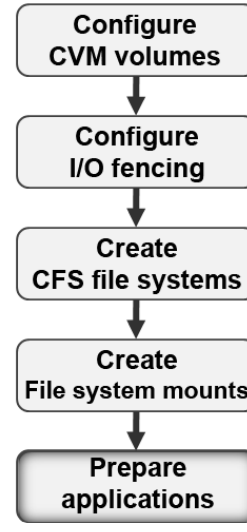
After initial configuration of CFS and shared file systems, administrators use the **cfsdgadm** and **cfsmntadm** commands for on-going administration of shared disk groups and file systems.

Administrators run the **cfsmntadm** command, installed during CFS installation, prior to mounting a shared file system. The **cfsmntadm** command configures the required VCS resources and service groups for the file system, and sets up automatic file system mounting and unmounting as nodes join and leave a cluster. Using options of the **cfsmntadm** command, an administrator can specify which cluster nodes are eligible to mount a file system, the nodes on which the file system should be mounted automatically, the preferred primary node (another node takes on the primary role if the preferred primary node is not running at mount time), and the file system's mount options.

Administrative hint 34

Administrators can use an option of the **fsclustadm** command to learn which node is hosting the primary CFS instance of a given file system and to change the node hosting the primary instance.

Application preparation



In general, applications running on nodes of a CFS cluster use file systems just as they would if they were running on individual non-clustered systems. Cluster-specific preparation is necessary for an application to run as a failover or parallel service group.

To prepare an application to operate as a VCS service, a developer defines the resources it requires and the dependencies among them. To be part of a service group, a resource requires a VCS type definition and an agent that can start, stop, and monitor it. VCS includes type definitions and agents for the most common types of resources, such as network interfaces, virtual IP addresses, CVM disk groups and volumes, and some popular applications such as the Apache web server.

The administrator uses either command line or graphical VCS configuration tools to create a VCS service group definition for the application. The service group specifies the application's resources and dependencies among them, as well as any inter-service group dependencies, and is inserted into the cluster's **main.cf** file.

The **cfsdgadm** and **cfsmntadm** utilities structure CVM volumes and CFS file systems as parallel VCS service groups; applications that use them are normally structured as failover service groups, and should have group dependencies on the file systems and volumes they require. The VCS service group names of file systems and volume groups can be found in the cluster's **main.cf** file after the utilities run.

Chapter 13

Tuning CFS file systems

This chapter includes the following topics:

The process of adjusting the operating parameters of a file system to optimize space utilization and I/O performance is commonly called *tuning*. CFS has a number of parameters that an administrator can adjust to tune a file system to match the needs of different I/O workloads.

CFS tunables are stored persistently in two locations:

- **/etc/vx/tunefstab**. CFS stores parameters specified by the **vxtunefs** administrative command in the **/etc/vx/tunefstab** file. Tunables may be file system-specific or may apply to all file systems in a cluster. CFS propagates changes in the **tunefstab** file to all cluster nodes
- **Operating system configuration files**. Certain CFS tunables are stored in operating system configuration files, such as **/etc/system** on Solaris platforms. An administrator modifies these by editing the file on the node for which the tunables are to be changed. Changes to driver-level tunable values take effect after driver reload; others take effect after node reboot

This chapter presents general guidelines tuning CFS to optimize I/O performance and space efficiency for the most frequently-encountered file types and access patterns. For definitive information and detailed instructions about the use of tunables, mount options, and application program advisories, the reader is referred to the *Veritas Storage Foundation Cluster File System Administrator's Guide* and the man pages for the applicable operating system.

To tune or not to tune

CFS file systems have been deployed in production applications for nearly a decade. During that time, much has been learned about optimizing storage space efficiency and I/O performance. As a result, the default values for most CFS tunables tend to provide optimal storage utilization and I/O performance for workloads that contain a balance of:

- **File I/O types**. Random and sequential, small and large, and read and write I/O requests
- **Data and metadata operations**. Metadata (file creation, deletion, renaming, permission changes, and so forth) and data (reading and writing) operations
- **File sharing**. Single-client and shared file access

If a file system's I/O load includes all of these, leaving tunables at the default values (set when a file system is created) generally results in near-optimal performance. For example, CFS tunable default values usually provide good performance for file systems that contain the home directories (personal files) of large numbers of users. Default tunables are also usually acceptable for Oracle transactional databases for a different reason: the Oracle database management system uses the ODM APIs to do its own "tuning."

Many CFS file systems are deployed in applications with specific file sizes, I/O

loads, and data consistency needs. File systems used as storage containers for databases of business transactions are a good example. Once an initial set of database container files has been created, these file systems experience relatively few file system metadata operations—files are opened when a database starts up and usually remain open indefinitely. Occasionally, a database administrator creates new container files, or extends existing ones. I/O requests are addressed to random database blocks; their sizes are typically a small multiple of the database block size. Thus, a CFS file system that will be used for transactional database container storage should be tuned to handle many small random I/O operations, and relatively few file creations and deletions.

File systems that provide workstation backing storage for groups of media artists or product designers would typically have somewhat different I/O loads. Metadata operations would be more frequent, but not greatly so. I/O requests would typically be large (multiple megabytes) and sequential, as users "check out" entire files to work on, and check them in again when they finish. File systems used in these applications should be tuned for large sequential I/O operations, and strong consideration should be given to minimizing fragmentation.

CFS file systems are deployed in a wide variety of applications with a wide variety of file storage requirements and I/O characteristics. Especially popular are seven classes of applications, each of which makes unique demands on a file system:

- **Transactional databases**. Applications that keep records of sales, product and service deliveries, registrations, and so forth, fall into this category, whether they use relational database management systems or other indexing techniques[24] to organize their data. Transaction records are typically small in size (a few kilobytes of data), and are read and written in random order, with read activity dominating in most cases. I/O load tends to be "bursty," characterized by busy peaks whose timing is not always predictable, followed by idle periods. The nature of transaction processing is such that I/O resources must be provisioned to handle peak activity with little or no increase in latency
- **Data mining**. Applications that analyze large bodies of records fall into this category. They process thousands or millions of records at a time, searching for trends or patterns, or simply accumulating statistics. Their I/O usually consists predominantly of large sequential reads—they typically scan data sets from beginning to end—with very little writing
- **Personal file serving.** Most enterprises provide some form of managed central storage for data created and used by their employees. The nature of personal business data varies from industry to industry, but in general

24. For example, see the discussion on page 50 about using sparse files to simplify data organization for large index spaces.

personal file serving is characterized by large numbers of clients and frequent metadata activity. Individual file accesses tend to be sequential, but the number of clients results in random I/O to the file system. Because clients tend to read files, work with them, and return them to the data store, overall I/O to the file system is usually balanced between reading and writing

- **Media.** Audio-visual media are usually stored centrally on high-capacity file servers, and downloaded to workstations for editing or to "server farms" for transformation and rendering. I/O workloads are dominated by high-bandwidth sequential transfers of multi-gigabyte files. As with personal file serving, media workloads have a balance of reading and writing. Distribution of the finished product, however, is dominated by reading from the file server
- **Extract, Transform, and Load (ETL)** As enterprises seek to derive value from their digital assets, applications that extract data from transactional databases, transform it for analysis, and load it into specialized databases, are becoming popular. CFS is particularly suitable for applications of this type, because different phases of an application can run on different cluster nodes and share access to data, either simultaneously or sequentially. The extraction phase of a typical ETL application writes somewhat less data than it reads, because it preserves only those data elements needed for later analysis. The loading phase tends to be dominated by large sequential writes as the application lays out data for efficient analysis
- **Build server.** Enterprises that develop software, either for their own use or for sale, often dedicate servers to compiling and building complete packages on a daily basis. Compiling and linking software is "bursty"—periods of intense I/O activity are interspersed with lulls as computations are performed. Data (source code files, temporary files, and binary module images) is accessed randomly, with a balance of reads and writes
- **Messaging.** Many enterprises use dedicated messaging servers to integrate disparate applications into a cohesive information processing whole. Applications communicate with each other by sending messages to the messaging server, which queues them for processing by their destination application servers. Messages may indicate major events (e.g., close of business day), or may be as simple as a single online transaction that triggers shipping, billing, accounting, and customer relationship management processes. As the central coordination point for all IT, messaging servers must be absolutely reliable. Driven by external events, I/O is typically random, and the request load can be heavy during active periods. Messages themselves are typically small

Each of these seven classes of application places unique demands on a file system. Table 13-1 summarizes the seven applications' relative characteristics as a backdrop for the discussion of tuning CFS file systems for efficient space

utilization and optimal I/O performance.

Table 13-1 Typical properties of common CFS application I/O workloads

Property	Database (transaction)	Database (mining)	File serving	Media	Extract, Transform, Load	Build server	Messaging
Typical file size	Large	Large	Mixed	Very large	Mixed	Small	Small
Typical number of files	Small	Small	Very large	Tens of thousands	Moderate	Thou-sands	Tens of thousands
Metadata I/O load	Negligible	Negligible	Very heavy	Moderate	Moderate	Heavy	Small
Read-write load	Heavy	Heavy	Heavy	Moderate	Heavy	Heavy	Heavy
Typical I/O size	Small	Large	Mixed	Very large	Mixed	Small	Small
I/O type	Random	Sequential	Random	Sequential	Sequential	Random	Random
Read-write mix	70%-30%	95%-5%	50%-50%	50%-50%	40%-60%	50%50%	80%-20%
Example	Oracle	Sybase	NFS, CIFS	NFS, CIFS	Informatica	Various	Tibco EMS

The sections that follow describe how CFS can be tuned to utilize storage efficiently, provide appropriate data integrity guarantees, and perform optimally for these and other workloads.

An overview of CFS tuning

A few CFS tuning considerations are not specific to data types or application workloads, but are applicable in all situations. Chief among them is distribution of the file system primary role in clusters that host multiple file systems. While CFS is largely symmetric, in the sense that all instances can perform most functions, a few key functions, such as file system resizing and online layout upgrades, are performed only by a file system's *primary CFS instance*. By default, a file system's primary instance is the one that mounts it first. Administrators may override this, however, and designate specific cluster nodes to fill the primary role for specific file systems.

Administrative hint 35

An administrator can use the **setpolicy** option of the **cfsmntadm** console command to permanently designate the order in which cluster nodes assume the mastership of a file system. The **setpolicy** option of the **fsclustadm** command can also be used for this purpose, but it operates only on the running cluster; it does not change the VCS **main.cf** configuration file permanently.

In clusters that host multiple file systems, administrators may wish to distribute file systems' primary instance roles among nodes to distribute primary instance processing load evenly.

CFS tuning points

Administrators and developers can affect CFS file system tuning in six areas:

- **Hardware configuration**. While CFS and CVM can be configured for greater or lesser I/O performance and resiliency to failures, it is also true that for a given software configuration, higher performing or more resilient hardware components will out-perform or outlast lesser ones
- **CVM volume creation**. The number of columns and mirrors in a volume can affect both business transaction (random access) and sequential streaming I/O performance
- **File system creation**. Administrators specify file system options such as file system block size, inode size, and intent log size, at file system creation time. Some of these are irreversible choices that cannot be altered during a file system's lifetime
- **File system mounting**. Administrators can specify options that affect I/O performance and data integrity guarantees when they mount file systems. These include Quick I/O and concurrent I/O (discussed in Chapter 11 on page 182), data and metadata logging guarantees, and caching behavior. Some mount options override default behaviors that are assumed by or

programmed into applications, and so should be used only with full knowledge of the consequences on application behavior

- **Ongoing operation**. An administrator can use the CFS **vxtunefs** utility program to alter the values of certain file system performance parameters, particularly those that affect sequential I/O performance and File Change Log behavior, while a file system is mounted and in use. In addition, the **fsadm** utility can be used to reorganize (defragment) files and directories, resize and move intent logs, and enable or disable large file support
- **Application development**. Application developers can include CFS libraries in their applications. These libraries allow them to program advisories that affect CFS behavior into their applications. Advisories include both those that control file system cache behavior and those that affect storage allocation for individual files

The sections that follow discuss CFS tuning in each of these areas.

Hardware configuration: tuning the CFS environment

Tuning a CFS file system starts with its storage. Different types of disks, disk array logical units, I/O interfaces, access paths, volume configurations, and other factors can affect file system performance. The two primary factors in tuning the CFS environment are the hardware components and configuration and the CVM volume configuration.

Hardware configuration tuning considerations

Choosing and configuring the hardware that provides persistent storage for a CFS file system is a classic three-way balance between cost, resiliency to failures, and I/O performance requirements.

Beginning with disks, the choice has conventionally been between high-RPM disks of moderate capacity (200-400 gigabytes) and lower-RPM, high-capacity (1-2 terabytes) ones. Disk drive vendors claim greater reliability for high-RPM disks, but this distinction is blurring as the increasing amounts of online data drive the market toward high-capacity disks, which motivates vendors to enhance their quality.

Recently, solid-state disks (SSDs) have matured to the point where they can realistically be considered for enterprise-class file storage. But aside from their cost per byte, which can be an order of magnitude greater than that of rotating disks, SSDs perform relatively better with workloads consisting primarily of random reads.

Most CFS file systems store their data on disk array logical units (LUNs), and

most disk arrays can be configured with a mixture of disk types. This flexibility enables a file system designer to choose the most appropriate disk type for each application. Generally, this means high-RPM disks for transactional applications, SSDs for smaller data sets where data criticality justifies the cost, and high-capacity disks for other data.

CFS Dynamic Storage Tiering makes it possible to distribute a file system across several CVM volumes, each consisting of disks of a different type. Administrators can define file relocation policies that cause CFS to relocate files between different types of storage as their states or usage changes.

Disk arrays organize disks into mirror or RAID groups that provide a first line of protection against disk failure. File system designers can choose among disk array resiliency options, realizing that CVM can provide a second layer of protection by mirroring LUNs, even those presented by different disk arrays. The main performance-related choices are the number of columns (disks) across which the disk array stripes data, and the number of access paths for communicating with LUNs (the latter is also a resiliency concern).

The final hardware configuration choice is the amount of memory in each cluster node. CFS instances use dedicated cache memory to hold active inodes and directory entries, but the greatest usage is for data cached in operating system page cache. When configuring memory, file system designers should consider the expected demands on each node, both those of file systems and those of applications and the operating system itself.

Volume configuration tuning considerations

CFS requires CVM volumes for persistent storage, even if no CVM mirroring, striping, or multi-path capabilities are configured. File system designers can configure multi-LUN CVM volumes to increase flexibility, resiliency, and I/O performance in the following ways:

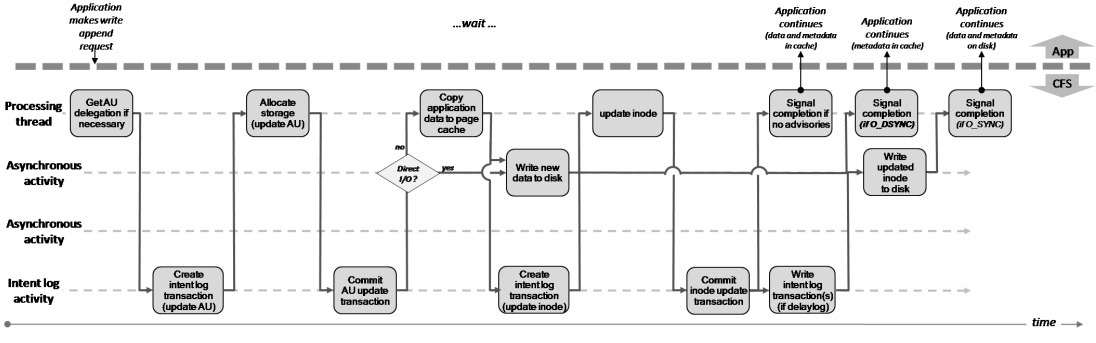
- **Flexibility**. During operation, mirrors can be split from CVM volumes, and deported to other systems for backup, data analysis, or testing, while the main volumes remain in production use. Using the Portable Data Container facility (PDC, page 43), volumes formed from split mirrors can even be imported and used on platforms of different types
- **Resiliency**. CVM can mirror LUNs presented by different disk arrays, and support multiple access paths to a LUN to increase resiliency above and beyond what a disk array can provide. CVM can take either full-size or space-optimized snapshots of volume contents to protect against data corruption. Finally, its volume replication (VVR) facility, can replicate the contents of a set of volumes across long distances for recoverability from site disasters
- **I/O performance**. CVM can configure volumes in which data is striped across multiple columns (LUNs). Striping tends to improve performance beyond that of a single LUN, both for sequential streaming applications and for

transactional applications that access data randomly. File system designers should coordinate hardware and CVM configuration choices with CFS tuning parameters so that the two interact synergistically

Tuning CFS file systems

Both the file system designer, the administrator, and the application developer have access to tuning parameters that can affect CFS file system performance, in some cases in conflicting or overriding ways. To appreciate the effects of tuning parameters, it may be helpful to review the sequence of individual events and actions that make up a simple CFS I/O operation. Figure 13-1, which repeats Figure 9-2 for convenience, summarizes the sequence of actions that CFS performs to append data to a file. Some details, such as resource locking, are omitted, because the purpose of the figure is to illustrate how I/O performance can be affected by tuning parameter values.

Figure 13-1 A representative CFS I/O operation



As Figure 13-1 suggests, CFS starts executing the operation by creating a transaction, allocating storage for the new data, and committing the transaction. If the file system is configured to write intent log transactions immediately (mounted with the **log** mount option), CFS writes the intent log record for the transaction at this point. Otherwise, writing can be delayed.

Next, if either of the **VX_DIRECT** or **VX_UNBUFFERED** tuning options that cause data to be written directly from application buffers is in effect, CFS schedules the data to be written directly from application buffers. Otherwise, it allocates operating system cache pages and copies the data into them before writing.

Next, CFS creates a transaction for updating the file's inode (to reflect the new file size, data location, and access time) and the allocation unit's space map (to reflect the storage allocated to the file). For recoverability, the order of operations is important. The data must be written and the transaction must be committed before the inode and allocation unit metadata are updated.

The upward-pointing vertical arrows in the figure indicate points in the sequence of actions at which I/O completion can be signaled to the application requesting the append, depending on which of the tuning options discussed in the sections that follow are in effect.

File system creation time tuning considerations

When using the **mkfs.vxfs** console command to create CFS file systems, administrator specify (either explicitly or implicitly by allowing them to default) options that can affect the file system's performance. Some of these cannot be changed once a file system has been created, so they should be chosen carefully. The immutable parameters chosen at file system creation time are:

- **File system block size**. The file system block size is the unit in which CFS manages the space assigned to it. A file system's block size can be specified as 1, 2, 4, or 8 kilobytes. File system block size determines the largest file system that can be created (32 terabytes with 1 kilobyte file system blocks; 256 terabytes with 8 kilobyte file system blocks) and the efficiency of space utilization (all files occupy at least one file system block, no matter how little data they contain)
- **inode size**. As stored on disk, a CFS file system's inodes are either 256 or 512 bytes in size (When CFS caches inodes in memory, it appends additional metadata to them). The primary use for larger inodes is to store more unique access control list entries (Access control lists that are common to multiple files are stored in separate inodes that are linked to files' inodes.) Under most circumstances, administrators should accept the CFS default inode size of 256 bytes, particularly if system memory is limited, or if the file system is expected to host multiple Storage Checkpoints
- **Volume zero**. A CFS file system can occupy multiple CVM volumes, for example to support storage tiering. With the exception of the first volume assigned when a file system is created (Volume 0), volumes can be added to and removed from a file systems volume set (VSET) at any time. When a file system is expected to occupy multiple volumes, a best practice is to choose a highly resilient, high-performing Volume 0, and specify file placement policy rules that limit Volume 0 to storing file system metadata

Two other parameters that affect file system tuning can be specified at file creation:

- **Large file support**. Limiting a file systems to file sizes of 2 gigabytes or less simplifies data structures and manipulation, and is primarily useful for platforms with very limited memory. Large file support is enabled by default for all platforms supported by CFS
- **Intent log size**. An administrator may specify the size of a file system's intent log (between 256 kilobytes and 256 megabytes, with a default of 16

megabytes) when creating it. Each CFS instance's intent log can be sized separately. When a node mounts a file system for the first time, its CFS instance creates an intent log with a size equal to that of the primary instance's log. In file systems subject to heavy metadata activity, larger intent log sizes may improve performance, because they reduce the chance that a full intent log will cause application requests to stall until transactions have been made persistent in the on-disk log

Both of these parameters can be changed during a file system's lifetime, so they are not so critical to define correctly at creation time as are the immutable ones.

Mount-time file system tuning considerations

Administrators can affect application I/O performance through the options they specify when mounting file systems. Mount options remain in effect only until a file system is unmounted by all CFS instances. They can affect file system tuning in three important ways:

- **Database I/O acceleration**. Either Quick I/O or Concurrent I/O (CIO) database I/O acceleration (Chapter 11 on page 182), but not both, can be enabled by mount options. Quick I/O and CIO improve database management system I/O performance by bypassing kernel write locking and in-memory data copying and by making it possible for database management systems to issue asynchronous I/O requests
- **Cache advisory overrides**. File data and metadata caching advisories encoded in applications can be overridden by specifying the **convosync** and **mincache** mount options
- **Intent log behavior**. Mount options can be used to alter the time at which the intent log is written as well as suppress logging of atime and mtime metadata updates

POSIX data and metadata persistence guarantees

By default, CFS signals applications that their write requests are complete when both data and any consequent metadata updates are in page or buffer cache. It performs disk writes after the completion signal. The POSIX standard includes two cache advisories that enable applications to direct CFS to persist data and metadata before signaling write request completion:

- **O_SYNC**. Both data and any metadata updates implied by the request have been stored persistently when request completion is signaled
- **O_DSYNC**. Data, but not necessarily implied metadata updates have been stored persistently when request completion is signaled

CFS data and metadata persistence guarantees

The CFS **mount** command includes two options that affect cache behavior:

- **convosync**. The **convosync** (convert **O_SYNC**) mount option overrides file system cache behavior for application write requests that specify the **O_SYNC** or **O_DSYNC** advisories
- **mincache**. The **mincache** mount option overrides file system cache behavior for application read and write requests that do not specify the **O_SYNC** or **O_DSYNC** advisories

Table 13-2 lists the values that can be specified for these mount options and the resulting modifications in caching behavior. The **convosync** option affects metadata caching for write requests; mincache can affect both reads and writes.

Specifying the **convosync** mount option causes CFS to override all **O_SYNC** and **O_DSYNC** advisories attached to application I/O requests. This option is generally used to improve overall I/O performance, but can affect data integrity if a system fails with unwritten data from **O_SYNC** or **O_DSYNC** requests in cache. Application recovery procedures might assume that data reported as having been written is in the file system's disk image. Administrators should therefore use the **convosync** mount option carefully, in full consultation with application developers and support engineers.

Specifying the **mincache** mount option causes CFS to treat application I/O that do *not* explicitly specify a cache advisory as indicated in Table 13-2. In general, it applies more stringent persistence guarantees to writes, but in the case of the unbuffered option, applications must leave I/O buffers untouched until CFS reports I/O completion to ensure that the data in them at the time of an I/O request is what is actually written.

Table 13-2 Effect of mount options on CFS cache advisories

Mount option value ↓	convosync option (effect on application write requests that specify O_SYNC or O_DSYNC)	mincache option (effect on application I/O requests without O_SYNC or O_DSYNC)
direct	Transfers data directly from application buffers When CFS signals write completion, file data, but not metadata, is guaranteed to be persistent	Transfers data directly to and from application buffers When CFS signals I/O completion, file data, but not metadata, is guaranteed to be persistent
dsync	Converts O_SYNC requests to VX_DSYNC (equivalent to O_DSYNC) When CFS signals I/O completion, file data, but not metadata, is guaranteed to be persistent	Treats all application I/O requests as though they had specified the VX_DSYNC advisory

Table 13-2 Effect of mount options on CFS cache advisories (Continued)

Mount option value ↓	convosync option (effect on application write requests that specify O_SYNC or O_DSYNC)	mincache option (effect on application I/O requests without O_SYNC or O_DSYNC)
unbuffered	Transfers data directly from application buffers Neither file data nor metadata are guaranteed to be persistent when CFS signals write completion	Transfers data directly to and from application buffers Neither file data nor metadata are guaranteed to be persistent when CFS signals I/O completion
closesync	Nullifies applications' O_SYNC and O_DSYNC advisories Writes file data persistently only when the last application to have a file open closes it	Same behavior as convosync=closesync
delay	Nullifies applications' O_SYNC and O_DSYNC advisories. Neither file data nor metadata are guaranteed to be persistent when CFS signals I/O completion	n/a
tmpcache	n/a	Does not specify when file data and metadata are persistent

Other mount options that affect file system tuning

Other ways in which mount options can affect I/O performance include:

- **Suppressing time stamp updates**. The POSIX standard specifies that the access (**atime**) and modification (**mtime**) times recorded in a file's inode should be updated each time the file is accessed and modified respectively. Inode update transactions can result in substantial I/O overhead, however. Because many applications do not require accurate **atime** and **mtime**, CFS provides **noatime** and **nomtime** mount options. The **noatime** option suppresses **atime**-only inode updates. The **nomtime** option causes CFS to update **mtime** only at fixed intervals
- **Erase on allocate**. CFS provides a **blkclear** mount option to prevent *scavenging* (allocating storage and reading its contents to discover what had previously been written). The **blkclear** causes CFS to return zeros when blocks that have not previously been written are read
- **Datainlog**. Normally, the CFS intent log records metadata updates. The **datainlog** mount option causes the data from small (less than 8 kilobyte) writes be written in the log as well. This option can reduce disk seeking in high-frequency random access update scenarios, especially when the same file system blocks are updated repeatedly

Finally, the **log**, **delaylog**, and **tmplog** mount options described in Table 13-3 affect performance by altering the time at which CFS writes intent log entries.

Table 13-3 Effect of mount options on CFS intent logging

Mount option ↓	Intent log update time
log	CFS writes intent log entries that pertain to an application I/O request persistently before signaling the application that its request is complete.
delaylog	CFS delays most intent log writes for about 3 seconds after signaling completion to the application, and coalesces multiple entries into a single write if possible. File deletion records are guaranteed to be persistent. delaylog is the default logging mode.
tmplog	CFS delays all intent log writing for an indeterminate period, and coalesces multiple entries into a single write if possible.

The **log**, **delaylog**, and **tmplog** mount options are cluster node-specific. To enforce consistent log behavior throughout a cluster, the mount option must be specified for all nodes.

Tuning CFS during daily operation: the vxtunefs command

Administrators can use the **vxtunefs** command to make immediate adjustments to tunable I/O parameters for mounted file systems. The **vxtunefs** command can affect:

- **Treatment of I/O requests**. Parameters that specify how CFS buffers, throttles, and schedules application I/O requests
- **Extent allocation**. Parameters that control file system extent allocation policy
- **File change log**. Parameters that affect the behavior of file change logs

The vxtunefs command operates either on a list of mount points specified in the command line, or on all mounted file systems listed in the **/etc/vx/tunefstab** file. File system parameters that are altered through the **vxtunefs** command take effect immediately

Administrative hint 36

An administrator can specify an alternate location for the **tunefstab** file by setting the value of the **VXTUNEFSTAB** environment variable.

when the command is issued, and are propagated to all nodes in a cluster.

Buffered and direct I/O

An administrator can mount a CFS file system with the option to transfer data for large I/O requests directly from application buffers, even for applications that do not specify the **VX_DIRECT** or **VX_UNBUFFERED** cache advisories. Table 13-4 lists the **vxtunefs** tunable parameters that affect application I/O requests buffering.

Table 13-4 Parameters affecting direct and discovered direct I/O

vxtunefs parameter ↓	Effect/comments
discovered_direct_iosz (default: 256 kilobytes)	I/O request size above which CFS transfers data directly to and from application buffers, without copying to page cache.
max_direct_iosz	Maximum size for non-buffered I/O request that CFS issues to a volume. CFS breaks larger application I/O requests into multiple requests of max_direct_iosz or fewer bytes.
In addition: vol_maxio (default: 2,048 sectors)	Maximum I/O request size that CVM issues to a disk. CVM breaks larger requests into requests for vol_maxio or fewer sectors, and issues them synchronously in sequence (Not set with vxtunefs)

CFS treats buffered read and write requests for more than **discovered_direct_iosz** bytes as though the **VX_UNBUFFERED** cache advisory were in effect. This is particularly significant with large I/O requests because it conserves system memory and processing by eliminating copying of data between application buffers and page cache.

The **max_direct_iosz** tunable specifies the largest non-buffered (subject to the **VX_DIRECT** or **VX_UNBUFFERED** cache advisory, or specifying more than **discovered_direct_iosz** bytes) I/O request CFS issues to a CVM volume. CFS breaks larger non-buffered requests into requests of no more than **max_direct_iosz** bytes, and issues them in sequence.

In addition to these, the CVM **vol_maxio** CVM parameter limits the size of I/O requests that CVM issues to volumes' member disks. If a CFS I/O request to a volume would require CVM to issue a disk request of more than **vol_maxio** bytes, CVM breaks it into smaller disk requests of **vol_maxio**-or fewer bytes.

Write throttling

By default, CFS flushes file data from operating system page cache at regular intervals. Administrators can limit the amount of operating system page cache that CFS will allow a single file's data to occupy in two ways. Table 13-5 describes how the **max_diskq** and **write_throttle** tunables affect CFS's periodic flushing of page cache.

Table 13-5 vxtunefs parameters affecting buffered write throttling

vxtunefs parameter ↓	Effect/comments
max_diskq (default: 1 megabyte)	Maximum number of bytes of data that CFS will hold in page cache for a single file. CFS delays execution of I/O requests to the file until its cached data drops below max_diskq bytes
write_throttle (default: 0) *(implying no limit)*	Maximum number of write-cached pages per file that CFS accumulates before flushing, independent of its cache flush timer

If the number of bytes in page cache waiting to be written to a single file reaches **max_diskq**, CFS delays execution of further I/O requests for the file until the amount of cached data drops below **max_diskq**.

If the number of pages cached for a single file exceeds **write_throttle**, CFS schedules pages to be written until the number of pages cached for the file drops below **write_throttle,** even if it has not reached its cache flush interval.

Sequential read-ahead and write-behind

Administrators can tune CFS to discover sequential buffered read and write patterns and pre-read or post-write data in anticipation of application I/O requests. Table 13-6 describes how the **read_ahead**, **read_nstream**, **read_pref_io**, **write_nstream**, and **write_pref_io** tunables control CFS's read-ahead and write-behind behavior.

Table 13-6 vxtunefs parameters affecting read-ahead and write-behind caching I/O

vxtunefs parameter ↓	Effect/comments
read_ahead (default: 1—detect sequential read-ahead)	Disables read-ahead, or enables either single-stream or multi-threaded sequential read detection

Table 13-6 vxtunefs parameters affecting read-ahead and write-behind caching I/O (Continued) (Continued)

vxtunefs parameter ↓	Effect/comments
read_nstream (default: 1) and **read_pref_io (default: 64 kilobytes)**	read_nstream is the maximum number of read-ahead requests of size read_pref_io that CFS will allow to be outstanding simultaneously
write_nstream (default: 1) and **write_pref_io (default: 64 kilobytes)**	write_nstream is the maximum number of coalesced write requests of size write_pref_io that CFS will allow to be outstanding simultaneously

An administrator can set the **read_ahead** tunable either to disable CFS read-ahead entirely, to detect a single stream of sequential reads, or to detect sequential reads from multiple sources.

When CFS detects that a file is being read sequentially, it allocates cache pages and issues **read_nstream** sequential read requests, each for the next **read_pref_io** bytes in the file, in anticipation of application read requests. Similarly, when it detects that a file is being written sequentially, it coalesces up to **write_pref_io** bytes of data in cache before issuing a write request. It allows up to **write_nstream** sequential write requests to be in progress concurrently. CFS sets default values for all four of these tunables by querying CVM when a file system is mounted to determine the volume's geometry (in particular, number of columns and stripe unit size), so administrators typically need not be concerned with them.

Controlling storage allocation and deallocation

Table 13-7 lists the **vxtunefs** tunable parameters that affect how CFS allocates storage space when applications create or append data to files. The **initial_extent_size** and **max_seqio_extent_size** tunables control the amount of storage CFS allocates for files as they are written. By default, when an application first writes data to a file, CFS allocates the larger of:

- **Calculated minimum**. the smallest number of file system blocks that is larger than the amount of data written by the application
- **Specified minimum**. 8 kilobytes

An administrator can raise the default value by setting the **initial_extent_size** tunable. Larger **initial_extent_size** is useful for file systems that predominantly contain large files, because it tends to reduce the number of extents across which file data is distributed.

Each time CFS allocates additional storage for an extending sequential write to a

file, it doubles the amount of its preceding allocation, until **max_seqio_extent_size** is reached, at which point it continues to allocate **max_seqio_extent_size**-size extents when additional space is required. This again tends to minimize the number of extents across which a large file's data is distributed.

Table 13-7 vxtunefs parameters affecting storage allocation

vxtunefs parameter ↓	Effect/comments
initial_extent_size	Minimum size of the first extent that CFS allocates to files whose storage space is not preallocated
inode_aging_count (default: 2,048)	Maximum number of inodes to retain in an aging list after their files are deleted (data extents linked to aged inodes are also aged). Aged inodes and extents accelerate restoration of deleted files from Storage Checkpoints
inode_aging_size	Minimum size of a deleted file to qualify its inode for aging rather than immediate deallocation when its file is deleted
max_seqio_extent_size	Maximum extent size that CFS will allocate to sequentially written files

The **inode_aging_size** and **inode_aging_count** tunables control CFS treatment of deleted files' inodes and data. When files larger than **inode_aging_size** are deleted, CFS saves their inodes in an age-ordered list of up to **inode_aging_count** inodes, and does not immediately delete their data. As applications delete additional qualifying files, CFS removes the oldest entries from the list. If Storage Checkpoints are active, files deleted from the primary fileset whose inodes are still on the aging list can be recovered (effectively "undeleted") quickly by copying them back to the active fileset, along with their data.

Tuning the File Change Log

The CFS File Change Log is useful for applications that depend on knowledge of which files in a file system have changed, but for certain types of activity, the overhead it imposes can be significant. Table 13-8 describes **vxtunefs** parameters that can be adjusted to reduce FCL activity.

The **fcl_keeptime** and **fcl_maxalloc** tunables control the retention of FCL records. To limit the amount of space used by the FCL, CFS discards records that are older than **fcl_keeptime** and frees the space they occupy. If the size of an FCL reaches **fcl_maxalloc** before any records have aged to **fcl_keeptime**, CFS "punches a hole" in the FCL by discarding the oldest records. Thus, for file

systems subject to heavy update loads, it is advisable to increase **fcl_maxalloc**, particularly if applications use FCL entries for auditing or other purposes.

Table 13-8 vxtunefs parameters that affect the file change log

vxtunefs parameter ↓	Effect/comments
fcl_keeptime	Number of seconds, that the File Change Log (FCL) retains records. CFS purges FCL records that are older than fcl_keeptime and frees the extents they occupy
fcl_maxalloc	Maximum amount of space that CFS can allocate to the FCL. When space allocated to the FCL file reaches fcl_maxalloc, CFS purges the oldest FCL records and frees the extents they occupy
fcl_ointerval (default: 600 seconds)	Minimum interval between open-related FCL records for a single file. CFS suppresses FCL records that result from opening a file within fcl_ointerval seconds of the preceding open.
fcl_winterval (default: 3,600 seconds)	Minimum interval between write, extend, and truncate-related FCL records for a single file. CFS suppresses FCL records of these types that occur within fcl_winterval seconds of the preceding operation of one of these types.

The **fcl_ointerval** and **fcl_winterval** tunables limit the number of FCL entries that CFS writes for files that are subject to repetitive activity. If the same file is opened repeatedly, for example, by an NFS server in a CNFS configuration, CFS suppresses the writing of all open-related FCL records within **fcl_ointerval** seconds of the last such record. Similarly, if a file is written repeatedly, as for example an application log might be, CFS suppresses write-related records within **fcl_winterval** seconds of the last FCL record.

Tuning CFS during daily operation: the fsadm utility

Administrators can use the **fsadm** command to tune certain properties of file systems while they are mounted, or if they have been cleanly unmounted. The principal uses of **fsadm** are:

- **Defragmentation**. Reorganizing file extents and directories to reduce fragmentation, and thereby improve performance
- **Resizing**. Increasing or reducing the storage space allocated to a file system, and the size of its intent log
- **Space reclamation**. Reclaiming storage space in disk arrays that support thin provisioning

In an empty file system, CFS allocates space to files in an order that tends to optimize I/O performance. Over time, as files are created, extended, and deleted, free space tends to become *fragmented* into a large number of small extents. Fragmentation tends to degrade file system performance, both because CFS must search harder to find free space to allocate to files, and because files that occupy many non-contiguous extents cannot be accessed with large I/O requests that use hardware resources efficiently.

CFS defragmentation

The **fsadm** utility *defragments* a mounted file system in three ways:

- **File reorganization**. Wherever possible, **fsadm** allocates contiguous extents of file system blocks and moves files to them to reduce the number of extents they occupy. CFS reorganizes files while a file system is in use; **fsadm** locks each file while it is reorganizing it
- **Free space consolidation**. In the course of reorganizing files, **fsadm** consolidates free space into as few extents as possible. Free space consolidation simplifies future space allocation because it enables CFS to allocate larger extents
- **Directory compaction**. When files are deleted, their directory entries remain in place, flagged so that they are invisible to users. **fsadm** compacts directory files by removing entries for deleted files and freeing any unused space that results from compaction

Fragmentation generally occurs sooner in file systems with high "churn"—rate of file creation, resizing, and deletion. Administrators should therefore schedule **fsadm** reorganizations regularly, for example, weekly for active file systems, and monthly for less active ones.

Administrative hint 37

Administrators can experiment with defragmentation frequency, increasing the interval if it does not result in much change in free space distribution, and decreasing it if the opposite is true.

Because defragmentation is I/O intensive, administrators should ideally schedule it to correspond with periods of low or non-critical I/O activity. To help determine the need for defragmentation, **fsadm** can produce a report that summarizes the level of free space fragmentation. The percentage of file system free space that is in very small and very large extents can be a guide to whether a file system should be defragmented. For example, if less than 1% of a file system's free space is in extents of 8 file system blocks or fewer, the file system is probably not excessively fragmented. If the number grows to 5% or more, then defragmentation is probably warranted.

Resizing CFS file systems and their volumes

One possible cause of fragmentation, and of lengthy **fsadm** defragmentation runs is lack of free space. If 90% or more of a file system's storage space is allocated to files, defragmentation may have little effect, primarily because there is little free space to reorganize. If this is the case, file system resizing may be warranted.

Administrators can increase or decrease a file system's size while it is in use, provided that the volumes it occupies have sufficient unallocated space to support an increase. CFS defragments a file system if necessary before decreasing its size. Decreasing a file system's size does not change the size of the volume it occupies.

Administrative hint 38

Administrators use a combination of the **vxassist** and **fsadm** commands to change the size of a CFS file system along with that of any of its underlying volumes. Alternatively, the CVM **vxresize** command can be used to resize both a volume and the file system that occupies space on it in the same operation.

An administrator can increase or decrease a file system's intent log size from the default of 16 megabytes when creating the file system. Larger intent log sizes may be desirable, for example, in file systems that provide NFS service, or those subject to write-intensive workloads. While they can improve performance by allowing more file system transactions to be buffered, larger intent logs use more memory, and may increase recovery time after system crashes, because there may be more logged transactions to replay.

Large files

By default, a CFS file system can host files larger than 2 gigabytes. An administrator can disable *largefiles* support when creating a file system, or can use the **fsadm** utility to disable or re-enable support while the file system is mounted. If a file system actually contains one or more files larger than 2 gigabytes, largefiles support cannot be disabled.

Reclaiming unused space in thin-provisioned disk arrays

For disk arrays that support thin provisioning, an administrator can use the **fsadm** *thin reclamation* feature to release physical storage occupied by unused file system blocks. The thin reclamation feature uses special APIs to communicate unused disk (LUN) block ranges to supported disk arrays. The disk arrays free the physical storage that backs the unused blocks. An administrator can optionally specify *aggressive reclamation,* which causes CFS to compact files prior to instructing disk arrays to reclaim space. Aggressive reclamation generally reclaims more space, but the compaction phase takes longer and

consumes more I/O resources. CFS supports APIs that enable applications and utility programs to initiate physical storage reclamation.

Application development tuning considerations

Using a library supplied as part of CFS, application programs can issue I/O control calls (**ioctl**s) to specify *advisories* that control how CFS uses cache on a file-by-file basis. Applications can specify:

- **Buffering**. The **VX_DIRECT** and **VX_UNBUFFERED** advisories direct CFS to read and write data directly into and from application buffers, rather than copying it to page cache before writing, or reading it into page cache before copying to application buffers. Direct I/O improves performance by eliminating the CPU time and memory consumed by copying, but applications must ensure that buffer contents remain intact until I/O operations complete. The **VX_DIRECT** advisory delays I/O completion until any metadata updates implied by writes have been written to disk; specifying **VX_UNBUFFERED** does not. Both advisories require page aligned application buffers; if they are not, CFS buffers the I/O
- **Metadata persistence**. By default, CFS signals I/O request completion before file metadata updates are persistent. This improves performance from the application's point of view. Applications that require absolute disk image consistency, can use the POSIX **O_SYNC** advisory to force CFS to delay signaling completion of I/O requests until metadata changes have been written to disk. Alternatively, specifying the POSIX **O_DSYNC** advisory (or its **VX_DSYNC** equivalent) delays request completion signals until data (but not necessarily metadata) has been persistently stored
- **Read-ahead behavior**. To accelerate sequential read performance, CFS detects sequential buffered reads from applications, and reads file data ahead in anticipation of application requests. Single-stream applications can specify the **VX_SEQ** advisory to instruct CFS to read ahead in the file by the maximum allowable amount. Multi-threaded applications that read several sequential streams in a file simultaneously can specify the **VX_ERA** advisory to cause CFS to maintain multiple read-ahead streams. Applications that read file data randomly can suppress read-ahead by specifying the **VX_RANDOM** advisory
- **Concurrent I/O**. Applications that manage their threads' file I/O requests so that concurrent requests do not corrupt data or cause incorrect behavior can specify the **VX_CONCURRENT** advisory to cause CFS to both bypass file write locking and to read and write directly into and from the application's buffers. The **VX_CONCURRENT** advisory provides the same accelerations for individual files that the **cio** mount option (page 189) provides for entire file systems

For these advisories and others, applications can determine the file system-wide

setting by issuing the **VX_GETFSOPT** ioctl.

In addition to these, other **ioctl** functions allow applications to control space allocation on a file-by-file basis. Applications can reserve space, trim reserved but unused space, specify a file's extent size, and specify alignment for newly allocated extents. Of these, the ones that affect I/O performance most directly are reservations that can allocate contiguous space for a file, and alignment, that can meet the constraints of the **VX_DIRECT** and **VX_UNBUFFERED** advisories. The **VX_TRIM** advisory, which deallocates unused space allocated to a file, promotes efficient space utilization.

Tuning CFS for space efficiency

The smallest unit of space that CFS can independently allocate is a file system block. As applications append data to a file, CFS allocates sufficient file system blocks to hold it. If the amount of data in a file is not a multiple of the file system block size, storage space is allocated but not used.

For example, a file that contains only one byte of data consumes a file system block whose size is between one and eight kilobytes. Table 13-9 illustrates the space efficiency (ratio of space occupied by data to space allocated) for a file containing 2,500 bytes of data (roughly comparable to a page of text in this document) and one ten times as large.

Table 13-9 Small and large file space efficiency for different file system block sizes

File system block size	Space allocated to a 2,500 byte file	Data:space ratio (%)	Space allocated to a 25,000 byte file	Data:space ratio (%)
1,024 bytes	3,072 bytes (3 file system blocks)	81%	25,600 (25 file system blocks)	98%
2,048 bytes	4,096 bytes (2 file system blocks)	61%	26,624 bytes (13 file system blocks)	94%
4,096 bytes	4,096 bytes (1 file system block)	61%	28,672 bytes (7 file system blocks)	87%
8,192 bytes	8,192 bytes (1 file system block)	30%	32,768 bytes (4 file system blocks)	76%

Table 13-9 represents a “best case” scenario, in the sense that no more file system blocks are allocated to the file than are required to hold its data. Thus, the maximum “wasted” (allocated but not used to store file data) space is one file system block. The table underscores the point that for small files, this can be a significant percentage of file size.

As the second and third columns of Table 13-9 suggest, smaller file system block sizes result in greater space efficiency (greater percentage of storage space occupied by actual data) for file systems that hold mostly small files. This suggests that if the average size of files that a file system will contain is known to be small when the file system is created, a smaller file system block size should be specified to optimize storage utilization. For example, if file sizes are expected to cluster between zero and 1.5 kilobytes, the administrator creating the file system should choose one kilobyte as a file system block size. For file sizes between 1.5 and 2 kilobytes, a file system block size of 2,048 is likely to be optimal, and so forth. This decision should be tempered by an awareness that the maximum size of a CFS file system is determined by its file system block size, because of the way free space data structures are organized. For example, the maximum size of a file system with a 1 kilobyte file system block size is 32 terabytes.

For larger files, storage space efficiency is less of a consideration, as the fourth and fifth columns of Table 13-9 suggest. Maximum “wasted” space for a file is again one file system block, but this is a much smaller percentage of file size, even for the relatively modest sized 25 kilobyte file size in the example.

For larger file sizes, another consideration may be more important than space efficiency—allocation efficiency. When files are extended, CFS allocates contiguous space from within a single allocation unit if possible. But when a file system becomes fragmented, it may not be possible to allocate large blocks of contiguous space. CFS must create an extent descriptor for each non-contiguous range of file system blocks it allocates to a file. Larger file system block sizes result in fewer extent descriptors than smaller block sizes. Files can grow larger before indirect extent descriptors are required, leading to better I/O performance for two reasons:

- **Larger disk reads and writes**. Because each extent is a contiguous range of block addresses on a single disk, an internal CFS I/O request can read or write as much as the size of an entire extent
- **Fewer accesses to indirect extent descriptors**. In file systems with larger file system block sizes, a file’s inode can map more data than with smaller ones. Therefore, it becomes less likely that CFS will have to refer to an indirect extent descriptor to retrieve or store data at the end of a large file

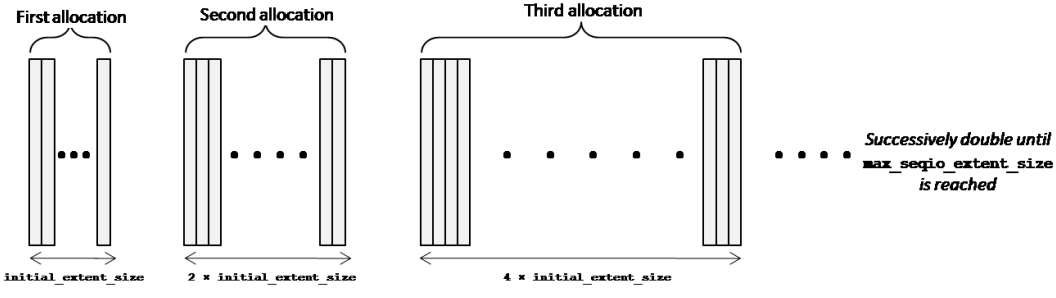
Tuning CFS for performance

Typically, CFS tuning delivers the greatest benefit for data streaming applications that access large files sequentially.

Tuning for sequential access

An administrator can further enhance storage utilization and I/O performance in file systems that create and access large files sequentially through the use of two other CFS tunables, **initial_extent_size** and **max_seqio_extent_size**. CFS allocates a single extent large enough to hold the data in an application's first write to a file. If CFS detects that the application is continuing to write data sequentially, it doubles the size of each subsequent allocation up to a default maximum of 2,048 file system blocks. If if no writes are issued to a file for a period of 60-90 seconds, CFS deallocates unused file system blocks.

Figure 13-2 CFS file system block allocation for sequential files



By raising the value of **initial_extent_size**, an administrator can cause large, sequentially written files to be more contiguous on disk. This reduces the number of times CFS must allocate storage for a large file, and at the same time, improves subsequent read performance because more data can be read with each disk request.

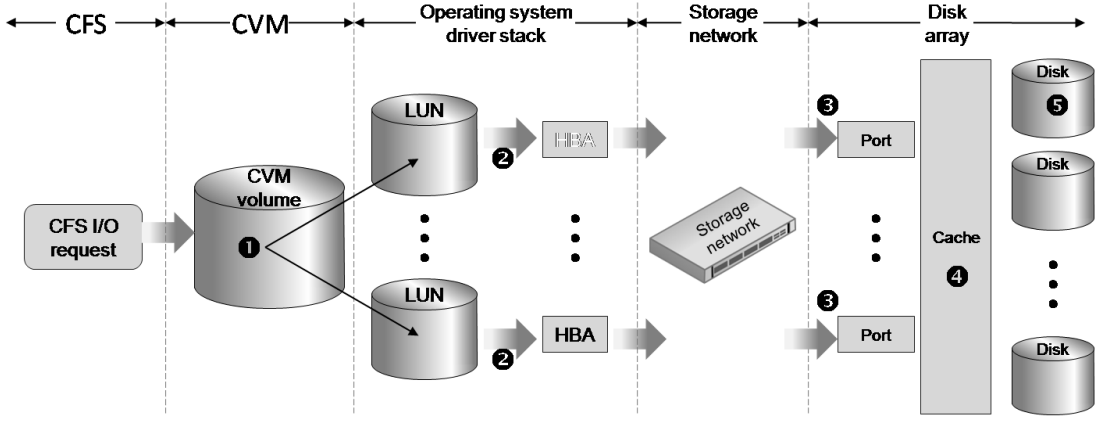
The value of **max_seqio_extent_size** limits the amount of storage that CFS will allocate to a sequentially written file at one time. Administrators can use this tunable to reduce allocation failures caused by files occupying excessively large contiguous ranges of file system blocks. CFS prevents the value of **max_seqio_extent_size** from falling below 2,048 file system blocks.

Tuning CFS for sequential I/O with disk arrays

Figure 13-3 illustrates the path taken by a sequential I/O request issued by CFS. The CFS I/O request is either the direct result of a request made by an application, or, if an application request specifies more than **max_direct_iosz**

bytes, one of the smaller requests for **max_direct_iosz** or fewer bytes into which CFS breaks it.

Figure 13-3 Stages in sequential I/O operations



If the CFS request specifies more than **vol_maxio** bytes, CVM breaks it into multiple requests. Figure 13-3 assumes that the request is not for more than **vol_maxio** bytes, and therefore is not broken down further.

CVM supports striped, mirrored, and striped mirrored volumes. If the volume has any of these configurations, CVM decomposes CFS I/O requests into stripe unit-size read and write commands to its member LUNs (❶).

Each LUN is associated with one or more host bus adapters (HBAs). If the LUNs that make up a volume are associated with different HBAs, they can process read and write commands and transfer data concurrently (❷). Likewise, if the LUNs are associated with different disk array ports, commands can be processed and data transferred on them concurrently (❸).

When CVM writes to a disk array, the data is typically absorbed by the array's non-volatile cache (❹), and written to disk media at some time after the array has signaled completion of the write. Client read commands may be satisfied from disk array cache, and indeed often are, because some disk arrays are able to detect sequential read patterns and "read ahead" in anticipation of client requests. For random reads, however, cache hit rates are typically low, especially if the application's working set of data is much larger than the disk array's cache.

Ultimately, the disk array transfers data to or from the disks that make up the LUNs to which commands are addressed (❺). In most cases, LUNs are striped, mirrored, or RAID-protected, so they present further opportunities for parallel execution.

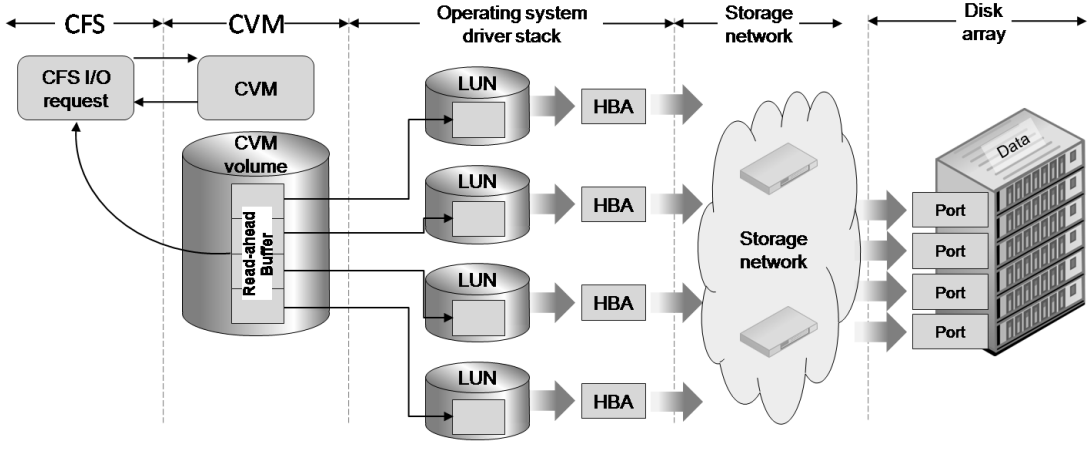
The makeup of CVM volumes can be exploited to maximize sequential I/O

performance. If the LUNs that make up a volume have separate paths to the disk array, then all can transfer data concurrently.

Thus, for example for a volume striped across four LUNs, the ideal I/O size is four times the stripe unit size. CVM splits such an I/O request into four smaller requests that execute concurrently (assuming the buffers are properly aligned). Figure 13-4 illustrates how this might work in the case of read-ahead.

If CFS detects that an application is reading from a file sequentially, and if the file's **read_ahead** (described in Table 13-6 on page 220) tunable parameter allows read-ahead detection, CFS allocates **read_nstream** buffers of **read_pref_io** bytes each, and issues read commands to CVM to read sequential file blocks into them.

Figure 13-4 Optimizing read-ahead performance



CFS aligns read-ahead buffers with volume stripe units, so that, as Figure 13-4 suggests, CVM is able to split the read-ahead request into four commands, each of which it issues to one of the volume's LUNs. As long as there is a separate path to each LUN (or alternatively, the paths to the LUNs have sufficient bandwidth to carry concurrent data transfers for all of them) the four commands execute concurrently, and data is ready for the anticipated application read request in a little more than a quarter of the time that would be required to read it in a single stream.

CVM Dynamic Multipathing (DMP) can simplify read-ahead optimization as well. For disk arrays that support concurrent multi-path LUN access, CVM can schedule I/O on different paths (For example, if the administrator selects shortest queue scheduling, CVM will usually issue a sequence of near-simultaneous requests such as that illustrated in Figure 13-4 to different paths.) Using DMP with supported disk arrays can provide the performance advantage of parallel I/O scheduling along with protection against I/O path failure.

Tradeoffs in designing CFS-based applications and systems

While for the most part, CFS can be considered to be “pre-tuned” for all except the most homogeneous workloads, designers should be aware of a few obvious factors as they develop applications and plan for deployment. The sections that follow present some guidelines for designing applications and configuring CFS-based clusters for high performance and optimal recoverability.

Performance consideration: sharing file systems, files, and directories

One of the primary advantages of CFS is that it enables applications running on different cluster nodes to simultaneously access shared data at all levels of granularity including concurrent write access to individual files. CFS uses its Global Lock Manager, described in Chapter 8, to maintain structural and content integrity of directories and files as applications create, manipulate, and delete files.

While the GLM design minimizes inter-node locking traffic, exchanging lock messages over a cluster’s private network inherently takes longer than locking access to resources within a single node. As designers determine the cluster nodes on which applications that share data will run, they must be aware of the tradeoff between the benefits of aggregating CPU power, cache, and network bandwidth by running applications on separate nodes, and the “cost” of increased I/O latency due to lock messaging as applications on different nodes access shared files concurrently.

Performance consideration: using directories to organize data

CFS implements the common UNIX hierarchical directory model by structuring each directory as a file. A CFS directory file is essentially a list of file (or subdirectory) names along with their corresponding inode numbers. As designers structure the data for their applications, they can choose between “flat” structures with few levels, and “deep” structures containing multiple levels of subdirectories.

In applications that involve a high frequency of directory operations, designers should be cognizant of the implications of different directory hierarchy designs. Because directories tend to grow in small increments, they often become fragmented in dynamic file systems that experience large numbers of file creations and deletions. Moreover, as files are deleted, directory blocks are not fully utilized (until directories are compacted administratively). Lookups in a large flat directory therefore tend to require a non-contiguous disk read for each extent, and may therefore be time-consuming, at least until frequently accessed

entries are cached. Deeper directory structures, with an average of fewer files per directory may have less tendency to fragment, but initial lookups in such a structure must traverse each level in the hierarchy.

Performance consideration: monitoring resources

CVM can be configured to monitor virtual volumes by accessing their disks periodically, and raising alerts if it encounters exception conditions. Application I/O to volumes also causes alerts to be raised when exceptional conditions are encountered, but volume monitoring can detect problems with idle volumes so that administrators can take remedial action before failures become critical, but in clusters with hundreds or thousands of disks, can result in considerable I/O activity that can have a discernible impact on application I/O. When CVM volumes provide the storage for CFS, however, there is sufficient background I/O activity that exceptions can usually be detected in the course of CFS I/O operations, so volume monitoring is unnecessary.

VCS monitors mount and volume resources for each file system and volume so it can take action (for example, failover) if it detects abnormalities. In clusters that host a large number of file systems, the number of resources that VCS must monitor is correspondingly large, and may consume noticeable processing and network resources. Designers should be cognizant of resource monitoring overhead as they specify virtual volume makeup and file system name spaces.

Recoverability consideration: file system sizing and Storage Checkpoints

CFS supports up to a billion files in a single file system. Data structures, caching, resource locking, and buffer management, are all designed to accommodate file systems on this scale. Moreover, CFS supports Storage Checkpoints within the context of a file system. Storage Checkpoints are extraordinarily useful for several purposes-for establishing application-consistent baselines for backup or data analysis, for testing applications against live data, for training developers, users, and administrators, and for enabling users to recover their own deleted or corrupted files. For the latter purpose especially, some administrators keep large numbers (dozens to hundreds) of active Storage Checkpoints of a file system. Again, CFS data structures and algorithms are designed to cope with the large amount of metadata that Storage Checkpoints necessarily entail.

When a UNIX system that crashes with mounted file systems recovers from the crash, it is generally necessary to verify file system structural integrity before remounting and making files accessible to applications. Verifying the integrity of a file system containing hundreds of millions of inodes could take days, and is clearly impractical if any reasonable service level agreement is to be met.

CFS overcomes this deficiency in almost all cases because it is a journaling file system that performs structural modifications transactionally and logs all

metadata updates in its intent logs before executing them. To recover from a system crash, CFS "replays" crashed nodes' intent logs, which contain a precise record of which file system metadata structures might be at risk. As a result, recovery time is related to the number of transactions in progress at the time of a failure rather than to the size of a file system or the number of files it contains.

On rare occasions, however, such as when disk media or memory failures corrupt file system structural data, it may become necessary for CFS to perform a full file system integrity check ("**full fsck**") during recovery. The most time consuming part of full file system checking consists of verifying that extent descriptors are consistent with the data structures that the file system uses to manage storage space. This time is proportional to both the number of files in the file system, the number of active Storage Checkpoints, and the degree of file fragmentation (average number of extents per file). While it is very rare that full file system checking is required, designers should be cognizant of the recovery time implication of file systems that contain very large numbers of files, have hundreds of active Storage Checkpoints, or are allowed to become very fragmented. Configuring more file systems with fewer files in each and limiting the number of active Storage Checkpoints can reduce recovery times in these rare instances, but at the expense of managing multiple name spaces.

Recoverability consideration: CFS primary and CVM master placement

While CFS and CVM are largely symmetric in the sense that any instance can perform nearly any function, each architecture incorporates the concept of a special instance that is the only one able to perform certain key operations. CVM's *master instance* manages all volume configuration changes, while each CFS file system's *primary instance* is responsible for allocation unit delegations and other administrative tasks. Both the CVM master instance and CFS file system primary instances do slightly more work than other instances, and perhaps more importantly, are critical to uninterrupted volume and file system operation respectively. The critical roles they play suggest that designers should carefully consider their placement in clusters of non-identical nodes or nodes that run at significantly different resource saturation levels. The CVM master instance should generally be configured to run on the most powerful or least loaded cluster node. CFS file system primary instances should generally be distributed across a cluster, with a with a slight bias toward more powerful or lightly loaded nodes, to equalize the file system processing load, but more importantly, to minimize the time to recover from a node crash, which is longer if one or more CFS primary instances must be recovered. For the same reason, CFS instances should be biased toward nodes whose application loads are the most stable (least likely to crash the nodes on which they are executing).

Checkpoints and snapshots

With respect to frozen image technology, CFS enjoys the proverbial "embarrassment of riches." File system Storage Checkpoints provide both read-only and read-write space-optimized point-in-time images of file system contents. In addition, CVM snapshots provide both full-size and space-optimized point-in-time images of administrator-selected groups of volumes that contain CFS file systems. Each of these frozen image techniques has its own advantages and limitations, and each is the optimal for certain scenarios. For example, Storage Checkpoints are simple to administer, because they occupy storage capacity within a file system's volume set. No separate storage administration tasks are required to create or remove a Storage Checkpoint.

Because they are space-optimized, Storage Checkpoints typically consume relatively little space compared to the file systems whose images they capture. This tends to motivate administrators to keep large numbers of them active. When this is the case, it becomes necessary to monitor file system space occupancy regularly, and to specify a strategy for handling out of space conditions should they arise, for example by making Storage Checkpoints automatically removable.

In addition to their other uses, Storage Checkpoints are an important mechanism for recovering from file loss or corruption; Deleted or corrupted files can be recovered by copying them from a Storage Checkpoint to the live file system. By themselves, however, they do not protect against data loss due to storage device failure, because they do not contain full file system images. Volumes mirrored by CVM or based on failure-tolerant disk array LUNs should be configured to protect against physical data destruction.

Administrators must explicitly designate the devices storage to be used by volume-level snapshots. They are useful for creating snapshots of multiple file systems at the same point in time. Full-size file system snapshots require storage space equivalent to that of the snapped file system, but can be used to take frozen file system images "off-host", for example to other cluster nodes, where they can be mounted privately and processed without interfering with live production I/O performance. As with Storage Checkpoints, space-optimized volume snapshots do not contain full images of their parent volumes contents, and so must be used by cluster nodes that are connected to the parent volumes.

CFS: meeting the technical challenge

For the past 4 years, it has been my privilege to lead Symantec's File System Solutions development team, the group responsible for creating and evolving CFS. When I commissioned this book, I did so in the hope of giving the enterprise IT user community an appreciation for what goes into the architecture, development, and effective deployment of the robust, scalable distributed file system that manages some of the most critical data on the planet and takes on some of the most demanding workloads in enterprise computing.

Reviewing the manuscript has led me to reflect on the team that has built this software to run on several operating systems and hardware platforms, and that continues to develop it apace with increasing customer demands and changing hardware and software platforms. The single-host VxFS file system that is the foundation for CFS has been developed over the past two decades by a team of several dozen engineers who work from three major locations: Symantec's Mountain View, California headquarters, Pune, India, and Green Park (Reading), England.

The on-going challenge of CFS attracts the best and the brightest. An average CFS engineer has nearly ten years of file system development experience; some of the most senior have over a decade of experience on CFS alone. CFS engineers have "earned their spurs" at some of the industry's leading companies, including Microsoft, Hewlett-Packard, IBM Corporation, Oracle, Cisco, Amdahl, Novell, and others, before joining the CFS team. Over a dozen have earned the rank of Technical Director at Symantec during or after tenure on the CFS team. Several CFS developers have gone on to successful careers, including a handful who are CEOs of startup companies and several others who are architects in firms developing storage products.

CFS has been developed by some of the most talented individuals in the field, but the real secret to success has been how they function as a disciplined development team, and integrate with other Symantec product teams to produce high-quality releases in synchronization with VCS, CVM, and other

Symantec software with which there are mutual dependencies. Each release starts with negotiation of a prioritized list of product requirements driven by product management. Once requirements are agreed upon, a regular series of File System Steering Committee meetings gives all stakeholders an opportunity to discuss and review designs and evaluate dependencies. Development occurs in a number of "sprints," each with well-defined outcomes and measurement criteria that include customer feedback, so that managers have a real-time picture of progress, both for CFS itself, and for related Storage Foundation components, throughout the cycle.

Fairly early in the development process, Quality Assurance begins testing new capabilities against an expanding repertoire of regression tests, to ensure that new features do not "break" existing functionality on which our customers depend. Later in the cycle come targeted beta tests, often with the very customers who have helped motivate new features and capabilities. Finally, CFS and all the other products that make up the various Storage Foundation bundles come together in release form for delivery to customers. And for engineering, the cycle begins again. Enthusiastic customers continually press for more, and the CFS team continues to develop CFS to meet the requirements they articulate.

Symantec puts customers first. Responsiveness to customer needs is a primary driver for CFS, and indeed, for the entire Symantec file system development team. All development managers, and most engineers regularly accompany product managers on customer visits. The first-hand exposure to users' concerns gained during these visits becomes a strong motivator to excel. Developers work extra hard on a feature when they've sat across the table from someone who wants it. Because CFS manages so many enterprises' most critical data, any customer problem that escalates to engineering becomes a top priority. Engineers interleave development with problem resolution, working round the clock if necessary.

I describe all of this in the hope that the reader will come to appreciate the complexity of our task, and the dedication and enthusiasm we bring to it. The CFS team takes its role as custodian of some of the world's most important data very seriously, and expends every effort to deliver the highest quality file system products in the industry, release after release. I am truly proud to manage this team of skilled, talented, and motivated people who bring you the best there is in enterprise file systems.

Bala Kumaresan

Bala Kumaresan
Director, Symantec File System Solutions Team
Mountain View, California

December 2009

CFS cache organization

To understand CFS performance, it is helpful to understand the internal actions required to execute client requests. Like most UNIX file systems, CFS relies extensively on cache memory to maximize performance, particularly in highly concurrent environments. Both data and metadata are held in cache at times during and after I/O request execution. File system mount options and application program cache advisories give both the administrator and the developer a degree of control over when and how data is cached and when in the I/O operation life cycle it is made persistent by writing it to disk storage. Chapter 13 discusses mount options and cache advisories and their interactions. This appendix discusses the five types of cache that CFS uses to optimize performance.

CFS internal cache

CFS uses five separately managed cache memories to hold metadata and data at various times throughout the I/O operation life cycle:

- **inode cache (i-cache).** CFS instances keep file inodes and related data structures in this dedicated cache while the files they represent are in use, and indefinitely thereafter, until it must reclaim space for the inodes of other active files
- **Quota cache.** CFS instances keep records of changes to file systems' quota files in this dedicated cache for rapid reconciliation and accurate response to requests that are subject to quotas (for example, a user or group with an assigned quota appending data to a file)
- **DNLC.** A file system directory is itself a file containing a list of data structures, each of which includes a [**file name, inode number**] pair. Given a path name, for example, in a file open request, CFS looks up the corresponding inode number by traversing the hierarchy of directories in the name. Because traversal can be time-consuming, CFS stores lookup results in a *Directory Name Lookup Cache* (DNLC) for future references. CFS instances' DNLCs are independent of each other; CFS does not reconcile them, but does manage

them so that inconsistencies between instances do not arise. Enterprise UNIX versions of CFS allocate and manage DNLCs; Linux versions use the operating system's **dcache**, which serves the same purpose

The CFS i-cache, quota cache, and DNLC are each dedicated to a single type of frequently used metadata, which simplifies both content management and searching. For other, less homogeneous, types of metadata and application data, CFS manages two other cache pools:

- **Buffer cache.** CFS instances use their own buffer cache to hold metadata that is not kept in the i-cache, quota cache, or DNLC
- **Page cache.** CFS instances use host operating system page cache to hold file and snapshot data

A CFS instance allocates memory for inode cache, quota cache, DNLC, and buffer cache when mounting a file system. It calculates the amount of memory allocated for each based on the memory size of the system in which it is running. Thus, in a cluster whose nodes have different memory capacities, different CFS instances may allocate larger or smaller amounts of cache.

CFS uses operating system page cache memory "greedily," relying on the operating system to regulate the amount of memory it is consuming relative to other processes in its hosting system.

Administrative hint 12

The **vxfs_ninode** tunable parameter is available on all platforms that CFS supports, but is manipulated differently on each. In Solaris systems, for example, it appears in the **/etc/system** file, whereas on RedHat Linux platforms it is specified in a configuration file in the **/etc/modprobe.d** directory. Administrators should consult the *Veritas Storage Foundation Cluster File System Administrator's Guide* for the platform and file system version in use to determine the location of the **vxfs_ninode** parameter.

In some cases, an administrator can adjust individual cache sizes by manipulating CFS parameters ("tunables") directly. For example, changing the value of the **vxfs_ninode** tunable causes CFS to override its computed size of the i-cache with the supplied value.

As Chapter 8 discusses, CFS implements several mechanisms by which instances synchronize access to critical metadata structures that are held in cache in various cluster nodes.

Bibliography

Key works related to CFS and other UNIX file systems

File System Forensic Analysis, by Brian Carrier
Addison Wesley Professional, March, 2005, ISBN-13: 978-0321268174

NFS Illustrated, by Brent Callaghan
Addison Wesley Professional, January 7, 2000, ISBN-13: 978-0201325706

Shared Data Clusters: Scaleable, Manageable, and Highly Available Systems, by Dilip M. Ranade
John Wiley & Sons, July 2002, ISBN-13: 978-0471180708

Storage Virtualization: Technologies for Simplifying Data Storage and Management, by Tom Clark
Addison Wesley, March 2005, ISBN-13: 978-0321262516

UNIX Filesystems: Evolution, Design, and Implementation, by Steve D. Pate
John Wiley & Sons, ISBN-13: 978-0471164838

Using SANs and NAS, by W. Curtis Preston
O'Reilly Media, February 2002, ISBN-13: 978-0596001537

Related Symantec publications

(available at no charge from www.symantec.com/yellowbooks)

Using Dynamic Storage Tiering

Using Local Copy Services

Standardizing Storage Management

The Veritas Cluster File System: Technology and Usage

Distributed applications with dynamic computing and I/O demands benefit greatly when they can share data, particularly when performance scales linearly as servers are added. Symantec's Veritas Cluster File System (CFS) is the file management solution of choice as major applications evolve to take advantage of modern data center "scale out" architectures.

Part I of "The Veritas Cluster File System: Technology and Usage" positions CFS within the spectrum of shared file management solutions. It goes on to describe CFS's unique properties and application scenarios in which it is used, and introduces an exciting new application: the industry's highest-performing NFS file server. Part II takes a "deep dive" into the structure of CFS and its VCS cluster environment, with particular emphasis on transactions, lock management, dynamic storage tiering, and performance accelerators for database management systems and other applications. Part III is an overview of CFS installation procedures and tuning options, along with a description of the tradeoffs between performance, cost, and data integrity that system administrators and application managers must inevitably make.

The audience for "The Veritas Cluster File System: Technology and Usage" includes both executives who wish to understand the distributed file management options available to them, application designers and developers in search of optimal ways to use CFS, and administrators responsible for the performance and robustness of CFS clusters. The authors' goal has been to answer questions commonly asked about CFS by those evaluating it, by new users, and by veterans wishing to understand its inner workings.

About Symantec Yellow Books™

Symantec Yellow Books deliver skills and know-how to our partners and customers as well as to the technical community in general. They show how Symantec solutions handle real-world business and technical problems, provide product implementation and integration knowledge, and enhance the ability of IT staff and consultants to install and configure Symantec products efficiently.

$49.99
ISBN 978-0-615-36045-4
54999>
9 780615 360454

www.symantec.com