Virtualizing Network File Systems

Junwen Lai

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

Advisor: Kai Li

January 2009

Abstract

Over the years, centralized file server appliances based on standard protocols such as NFS and CIFS have been adopted as the *de facto* standard for file service. The scalability limitations of centralized file servers have led large organizations to deploy numerous independent appliances to meet huge increases in their storage demand caused by phenomenal data growth and more stringent regulatory compliance requirements. Unfortunately, this stop-gap approach brings administrators into another management quagmire: forcing them to manage numerous independent storage islands and fight bottlenecks and load imbalances at a high cost. In addition, this approach often causes significant outage in file service as well as other business critical services that depend upon file service, resulting in loss of revenue as well as direct and significant customer impact.

This dissertation proposes a new *virtual file service* (*v*FS) layer to automatically manage file server infrastructure as a single resource pool with minimal or no attention from system administrators. *v*FS provides non-disruptive file service to clients even during server capacity expansion, server capacity shrinkage, server load balancing and other server maintenance events through protocol virtualization and transparent data migration. It consists of a dynamically growable set of light-weight *v*FS nodes placed between the clients and the file servers, and in achieving this, it requires no changes

to the clients, the servers and the protocols they use for communication. *v*FS nodes serving the same virtual volume also form a federation amongst themselves such that they can be used interchangeably for load balancing and failover purposes.

A proof-of-concept *v*FS prototype has been implemented on Linux with easily manageable software complexity. This prototype was used to measure the virtualization overhead for different NFS operations and the results indicate that most of the delay was caused by dual network traversals. A single *v*FS node without optimization is able to manage about 7 file servers in the experimental setup and adding more *v*FS nodes to a federation can aggregate the throughput of even more servers effectively. *v*FS nodes in the same federation are able to take over the responsibility of failed nodes quickly and transparently without clients noticing even for very strict client retry configurations. Transparent online data migration also has proved to be an effective mechanism in correcting load imbalance and access hotspots on file servers.

Acknowledgements

My work was funded by NSF grants CCR-9984790 and CNS-0313089.

First and foremost I would like to thank my co-advisors Kai Li and Randy Wang. For all these years, they have been extremely patient with my thesis progress and always provided support, advice and encouragement. It was my privilege to be able to receive guidance from two brilliant minds with extraordinary insight and vision in both academic and industrial researches. The inspiration from Kai and Randy also vastly deepened my understanding of the meaning of life.

I am grateful to Larry Peterson and David August for serving as readers on my thesis committee and for their valuable comments and suggestions on the thesis drafts. I would also like to thank other faculty members in the department who keep their doors open to graduate students. Graduate coordinator Melissa Lawson and other highly responsive CS staff members made my experience at Princeton a more enjoyable one. I am especially grateful to Melissa for her going above and beyond to help me and other Chinese students with our spoken English in my first year of study.

I am indebted to Ram Swaminathan and Mustafa Uysal for their mentorship at HP Labs. The quality of my PhD research was enhanced considerably by my interaction and collaboration with Ram and Mustafa.

No words are enough to describe my gratitude to my parents, Jingfeng and Jinxiu,

for their love, nurturing and support. Finally I would like to thank my wife Daijue without whose love, continued patience, understanding and encouragement, I could not have imagined finishing this thesis a few years after leaving school.

To My Parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

File systems are a core component of any modern operating system. They store and organize data on storage devices such as hard disks, CD-ROMs and CompactFlash cards; and translate the low level physical views of data to high-level logical views such as files and directories/folders. File systems make the task of locating, navigating, accessing and manipulating data significantly simpler and faster; as a result, they have long been relied upon by organizations and personal users alike to manage data. The insatiable need of these users for higher throughput and lower latency has led researchers to invent various creative file system designs [43, 54] by taking into account the mechanical nature of storage devices [56, 73, 33].

Enabled by digital communication channels, remote file systems naturally emerged to satisfy another equally important need of file system users to share information across different machines in an easily accessible manner. With a remote file system, one machine can access files and directories which are not stored and managed locally as if they were. As the bandwidth of LAN such as Ethernet continues to improve at a phenomenal rate, remote file systems keep gaining more and more popularity.

1.1 Remote File Systems

The simplest form of remote file systems is single-server file systems. In a single-server file system, one centralized server exposes a portion of its local file system, usually called a *volume*, to clients. Each client mounts the exported volume under its own local file system and can then access the volume similar to if it were mounted from a local storage device. Single-server file systems employ the typical "fat-server, thin-client" architecture in which the server assumes all the responsibility of storing and managing data, making administration relatively easy. Clients usually do not communicate with each other. Figure 1.1 shows a single-server file system hosted on server S accessed by four clients: C_1, C_2, C_3 and C_4. S exports volume `/share`. C_1 mounts it as `/server` and C_4 mounts it as `/public` in their own local file systems, respectively.

One of the earliest and most well known single-server file systems is the Sun Network File System (NFS) [60]. NFS's simple, open and portable design made it easy to implement NFS servers and clients on any operating system and machine architecture, thus enabling NFS to quickly gain massive success. Today, implementations of NFS is probably available on almost every computer and every operating system in the world, from mainframes to PCs, from Linux to Microsoft Windows. NFS has gone through a few rounds of Internet Engineering Task Force (IETF) standardization processes, and the currently dominant versions of NFS standards are NFSv2 [45] and NFSv3 [46]. Another highly popular single-server file system is Microsoft Common Internet File System (CIFS) [26, 14]. All versions of Microsoft Windows include CIFS client implementation and Microsoft Windows Server includes CIFS server implementation. Most Linux distributions also come with Samba [58], a suite of native

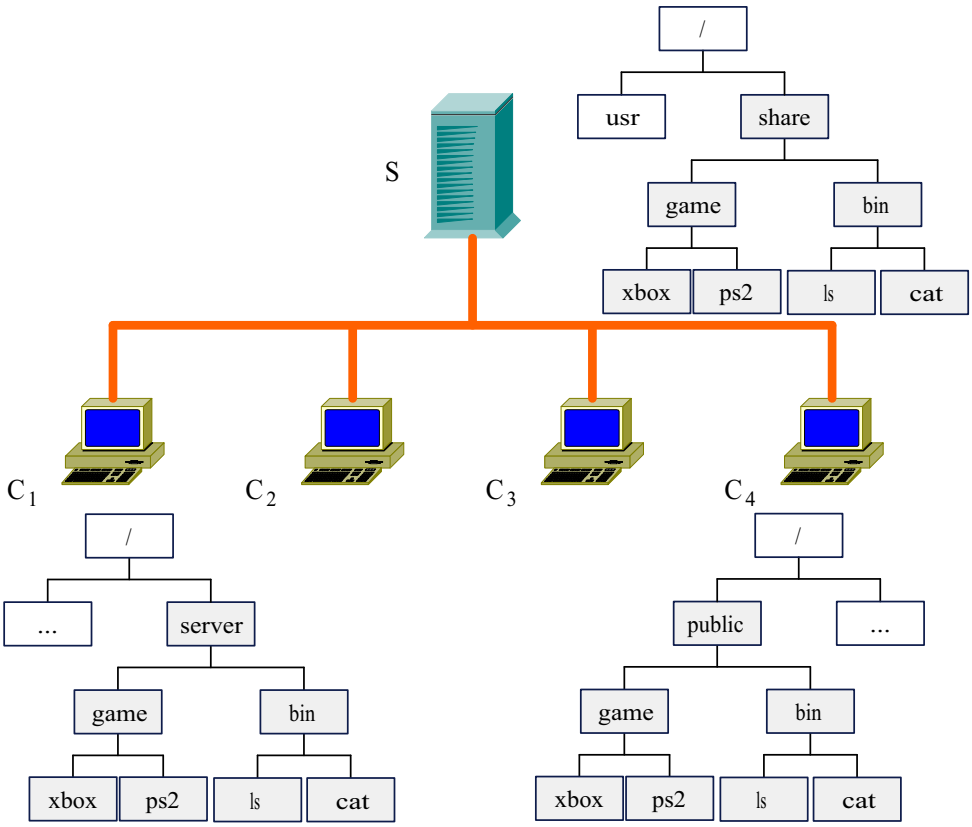

Figure 1.1: An example single server remote file system hosted on server S accessed by four clients: C_1, C_2, C_3 and C_4. S exports volume `/share`. C_1 mounts it as `/server` and C_4 mounts it as `/public`. All the files and directories under `/share` on S can be accessed by C_1 and C_2 as if there they are stored on a local device. For example, `/share/bin/ls` on S can be accessed as `/server/bin/ls` by C_1 and `/public/bin/ls` by C_4 respectively.

implementation of CIFS. The underlying messaging protocol Server Message Block (SMB) [24] of CIFS has already been an Open Group (formerly X/Open) standard for PC and Unix inter-operability since 1992.

The relatively simple single-server architecture makes NFS and CIFS easy to implement correctly and reliably, and the standardization makes different implementations inter-operable. As a result, the overwhelming majority of today's deployed file server infrastructure is based on either NFS or CIFS. As storage demands increase,

however, this architecture inevitably creates scalability limitations which eventually lead to serious *storage island* and *server sprawl* problems.

1.2 Storage Islands and Server Sprawl

The annual growth rate of total disk storage systems capacity shipped is about 50% per year, according to latest reports from IDC worldwide disk storage systems quarterly tracker [29]. This is also consistent with the annual growth rate of the amount of stored information, estimated to be somewhere between 50% and 125% according to most analysts. During the Internet bubble, taking into account only online data (excluding tapes, optical disks, and other tertiary storage media), storage system capacity was even estimated to have doubled every six to twelve months [50].

The combination of rapid data growth and the single-server architecture forces large organizations to deploy numerous file server appliances [27]. For example, even a wine and spirits distributor needs to "add two or three servers each month just to keep up with storage demands" [3]. In such a multi-server environment, each file system client mounts all the volumes exported by all the file servers based on a mount table. Figure 1.2 depicts the file system configuration of a typical university department. All faculty members' home directories are hosted on server S_1's `/fac` volume, all students' home directories are hosted on S_2's `/stud` volume and all staff members' home directories are hosted on S_3's `/staff` volume. Such that different clients (such as C_1 and client C_2 in Figure 1.2) see the same name space, they share the same mount table: volume S_1:`/fac` mounted under `/home/fac`, volume S_1:`/stud` mounted under `/home/stud` and volume S_1:`/staff` mounted under `/home/staff`. The "state-of-the-art" practice in the field to manage NFS mount tables is to use `automounter` [13]

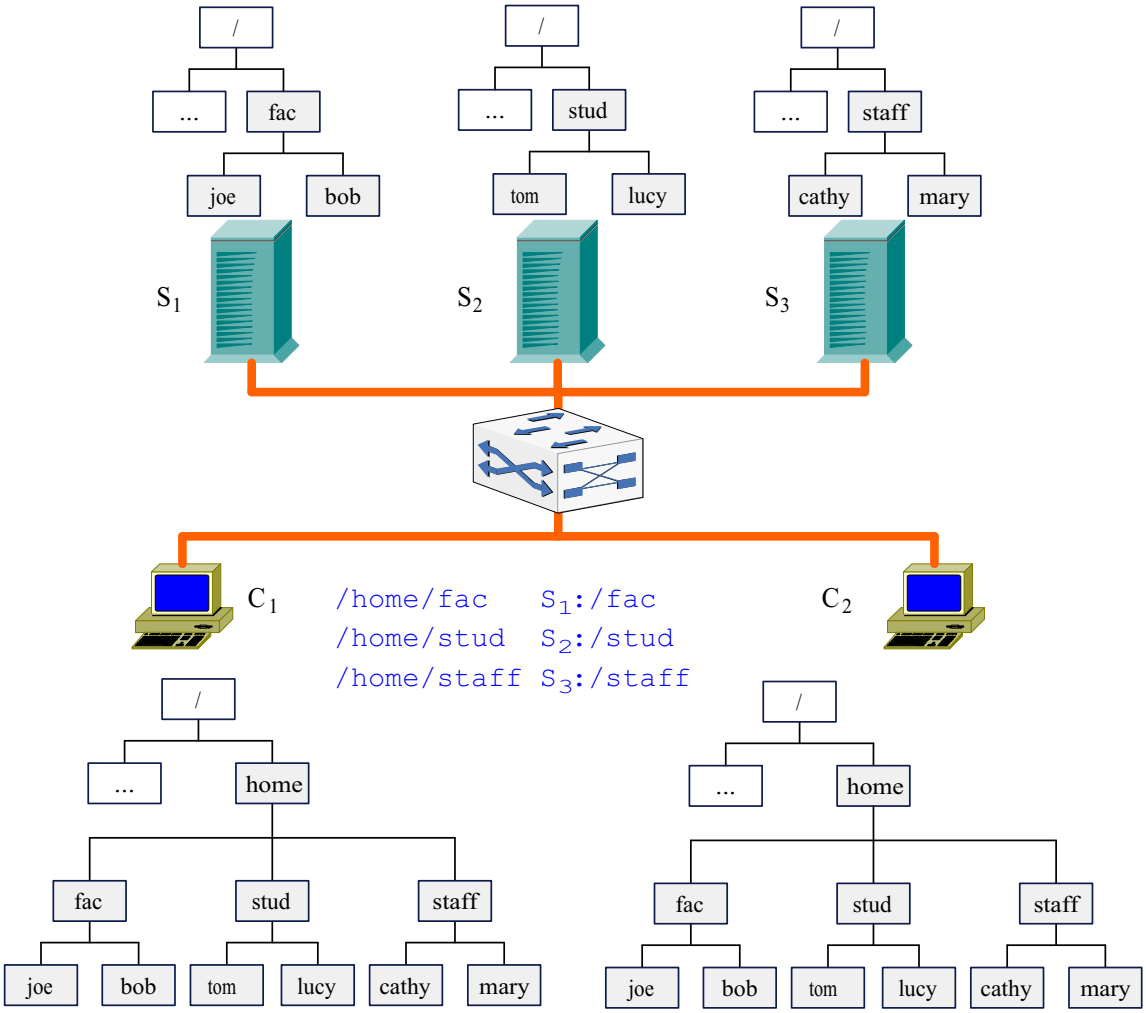

Figure 1.2: A typical file system configuration of a university department. Volume S_1:`/fac` is mounted under `/home/fac`, volume S_2:`/stud` is mounted under `/home/stud` and volume S_3:`/staff` is mounted under `/home/staff`.

or `autofs` [12] optionally combined with some light weight directory service such as LDAP [38]. CIFS also allows multiple CIFS servers to be mounted under one directory tree based on Distributed File System [18] and Active Directory [2] technologies. This server management practice has two important characteristics: (1) the partitioning of data among different servers is done manually and (2) once the partition is determined, it is kept unchanged for an extended period of time. This *manual* and *static* approach to managing file servers treats them as independent boxes which essentially form *storage islands*. Since each storage island is managed independently, the operation and management of the entire infrastructure is overly complex and costly. In addition, the practice often leads to very significant file service outage, and introduces significant server under-utilization, a phenomenon often referred to as *Server Sprawl*.

1.2.1 Storage Island Management Problems

There are numerous management tasks storage administrators need to carry out on a regular basis: (1) when file service needs outgrow existing resources, either new servers need to be added or existing servers need to be upgraded; (2) when there is significant load or capacity imbalance among servers, some data need to be shifted from one server to another; (3) when vendors release functionality/vulnerability patches for software or firmware, they need to be applied; (4) when some file server components such as disks fail, they need to be replaced; (5) old servers often also need to be retired. While clients are accessing the storage servers, none of these system administrative tasks can be done correctly and reliably. In order to avoid data loss, administrators often follow the lengthy recipe described below.

1. Send out "system downtime" email to all users affected and coordinate the

. . . sysadmin needs to take several NFS servers offline . . . These servers will be patched to the most current Sun OS patches and device drivers, sysadmin will also be upgrading the SCSI cards. . . . any NFS client which attempts to make use of `/u/share` or any other filesystem on the affected servers will hang. sysadmin strongly recommends you log out prior to . . .. If you believe that this downtime will cause a severe impact to your schedule, please notify me immediately. . .

Table 1.1: An email sample from the administrator of a prestigious research institute notifying users of file service disruption and coordinating the maintenance schedule.

maintenance schedule. Table 1.1 is an email sample that file service users are not unfamiliar with;

2. Stop all services such as web servers and email servers which depend on file service, causing application service disruption;

3. Detach affected file servers from the network;

4. Run the real maintenance tasks such as shifting a set of files off the overloaded server(s) to other servers;

5. Reconfigure affected servers;

6. Update the mount tables on each client or the central database and request all the clients to remount the file systems affected and

7. Send "back-to-normal" email and restart the application services.

As can be easily observed from this recipe, managing storage islands is very time consuming as well as error prone. Furthermore, both the complexity and frequency of maintenance events grow along with the number of servers. As a result, storage administrators are tied to the eternal fire-fight treadmill to satisfy the ever changing storage needs; and enterprises have to pay very high cost for each byte of storage. In fact, it is widely believed that the cost of maintaining a server is *five to seven times*

the purchase price. What is even worse, the current approach to managing file servers causes significant outage in file service as well as other business critical application services that depend on file service; for example, web services, email, content and metadata indexing, and archiving all depend on reliable file service. Frequent service disruption often leads to a loss of revenue as well as direct and significant customer impact.

1.2.2 Server Sprawl

Due to the high cost and service disruption associated with upgrading existing servers, adding new servers and retiring old servers, file servers are often over-provisioned to accommodate peak loads, resulting in significant resource under-utilization. Another common source of decision for over-provisioning comes from the practice of dedicating servers to individual applications. For example, it is common practice for web servers and email servers to have their own file server backends. These servers take up more space, consume more hardware, software and management resources than can be justified by their workloads. Anecdotal evidence shows that servers typically run at only about 15-20% of their capacity, further bloating enterprises' spending in information technology.

1.3 Thesis Focus

This dissertation proposes a new *virtual file service* (*v*FS) layer to automatically manage file server infrastructure as a single resource pool with minimal or no attention from system administrators. *v*FS provides non-disruptive file service to clients even during server capacity expansion, server capacity shrinkage, server load balancing and

other server maintenance events. It consists of a transparently and dynamically growable set of light-weight *v*FS nodes placed between the clients and the file servers, and in achieving this, it requires no changes to the clients, the servers and the protocols they use for communication.

By bringing disparate file servers under one umbrella of control, *v*FS provides the following benefits:

Location independence: *v*FS provides one global name space for each *virtual volume,* similar to physical volumes exported by an NFS server. Clients are completely oblivious of the distributed nature of virtual volumes. Each virtual volume can be mounted by multiple clients through different *v*FS nodes.

Resource sharing: *v*FS enables any virtual volume to dynamically consume resources from any server. It does this by breaking client-to-server bindings through *protocol virtualization* and breaking data-to-server bindings through *transparent online migration.*

Easy maintenance and scaling: Server maintenance/upgrades, capacity expansion and the retirement of obsolete hardware can be done online transparently with *v*FS, thus avoiding any service disruption. Rather than following the complex and error-prone 7 step recipe described above, system administrators now only need to identify data that needs to be shifted and simply issue migration instructions to *v*FS. Load and capacity balancing can even be fully automated by having an agent monitoring traffic pattern and storage capacity changes.

Transparent services: *v*FS can act as a file system gateway across isolated networks enabling the secure sharing of data or enforcing differentiated quality of service

levels. *v*FS also enables other capabilities such as unified user identities and new authentication schemes.

One of the major challenges in designing *v*FS is to make directories and files managed by different file servers appear to come from one single server although these servers were designed to only work independently by themselves and clients may mount the same volume through different *v*FS nodes. This challenge is tackled in *v*FS through complete protocol virtualization. The second major challenge is to enable dynamic resource sharing and data shifting while files and directories are being actively accessed by clients, potentially at a very high request rate. This concept of a single resource pool is achieved through transparent online data migration. Transparent migration is also designed to interfere with normal client traffic as little as possible. Yet another major challenge for *v*FS is to not introduce a single point of failure or bottleneck. In *v*FS, clients can access the same virtual volume through different *v*FS nodes, all of which present the same consistent image. These *v*FS nodes form a *federation* among themselves. In such a federation, the load on *v*FS nodes is balanced; and one *v*FS node can monitor another *v*FS node's health and transparently take over its responsibility without clients noticing.

A prototype for *v*FS has been implemented on Linux to demonstrate the manageable complexity and effectiveness of the *v*FS concept. The virtualization overhead experiments using this prototype show that most of the delay was caused by dual network traversals. Under the industry standard SPECsfs workload, a single *v*FS node without optimization has demonstrated the capability of managing about 7 file servers in the experimental setup. *v*FS shows linear scaling of throughput as the number of file servers and *v*FS nodes are increased. Standard clients accessing *v*FS nodes in the prototype have also proved to be able to fail over to another *v*FS node

well before the UDP-based RPC messages time out. Finally, *v*FS federations can dynamically and transparently migrate data to increase throughput and to balance the load among the file servers. In one experiment, *v*FS migrated over 120,000 files and directories containing more than 3.6 GB of data in around an hour and a half; reorganizing file system data among eight file servers and improving the throughput by about 65%.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 highlights a few principles used to guide the design of *v*FS, presents its high level architecture, introduces its various components, and compares and contrasts it with the literature. Chapter 3 briefly reviews the NFS protocol, studies the challenges virtualization faces in more detail, then describes how *v*FS virtualizes various aspects of NFS protocol: ID space, name space and locks. It also presents the details of a shared replicated persistent directory service component. The details for transparent migration can be found in Chapter 4. This chapter not only explains the various *v*FS migration primitives and the actual migration algorithms, but also studies how transparent migration can be used to support file server load balancing, capacity balancing and tiered storage management. Chapter 5 focuses on the load balancing and failover mechanism in a *v*FS federation. The prototype is described in Chapter 6 along with the setup and results of various experiments designed to evaluate *v*FS's overhead, scalability and the effectiveness of migration. Chapter 7 summarizes this thesis and presents a few ideas that are worth exploring in the context of *v*FS in the future.

Chapter 2

Overview of *v*FS

This chapter presents the high-level design overview of *v*FS. It starts by describing the requirements for a good *v*FS and the four principles that were used to guide the design process so that *v*FS is easy to deploy, easy to manage and scalable. Section 2.2 then presents the architecture and different functional components of *v*FS: the file service delivery layer, the replicated persistent directory service, the transparent migration coordinator, the migration policy manager, the mount time load balancer, the online load balancer and the management console. The last section examines the *v*FS work in the context of related literature.

2.1 Design Principles

As a component introduced to improve file server management, *v*FS itself should be *self-managing* so as not to introduce new management burden; as part of the shared storage infrastructure, *v*FS needs to be highly *reliable*; as an extra layer introduced into the client-server communication path, *v*FS needs to be very *light-weight* so as to minimize its impact on throughput and latency. With these as key requirements, the

architecture of *v*FS was guided by the following four design principles.

Simplicity: Since *v*FS nodes are placed on the path of file system requests and replies, only the minimal and most primitive operations are implemented in *v*FS nodes. More complex, but less frequently used functionality, such as transparent online migration, is implemented under the direction of *coordinators* outside the *v*FS nodes. This separation of responsibility not only reduces the overhead incurred by the virtualization layer, but also makes the whole system more robust and easier to diagnose.

Soft state: Any information that the *v*FS layer maintains should be soft state that is completely rebuildable from the underlying file servers being managed. Thus, *v*FS nodes do not cache any file system data, metadata, or any information critical to data integrity, the loss of which may render file servers inaccessible. In the worse case where the whole *v*FS layer crashes, file service may be temporarily disrupted but no data should be lost.

Leveraging file servers: Most commercial file server products are equipped with specialized hardware (e.g., NVRAM) and are highly optimized for file service. The *v*FS design seeks to exploit such optimizations as much as possible and avoid implementing duplicate functionality in the *v*FS.

Fast common case: Handling common tasks in a fast path is an effective technique for achieving better performance and higher throughput. *v*FS makes judicious use of this principle to minimize the overall performance impact of virtualization.

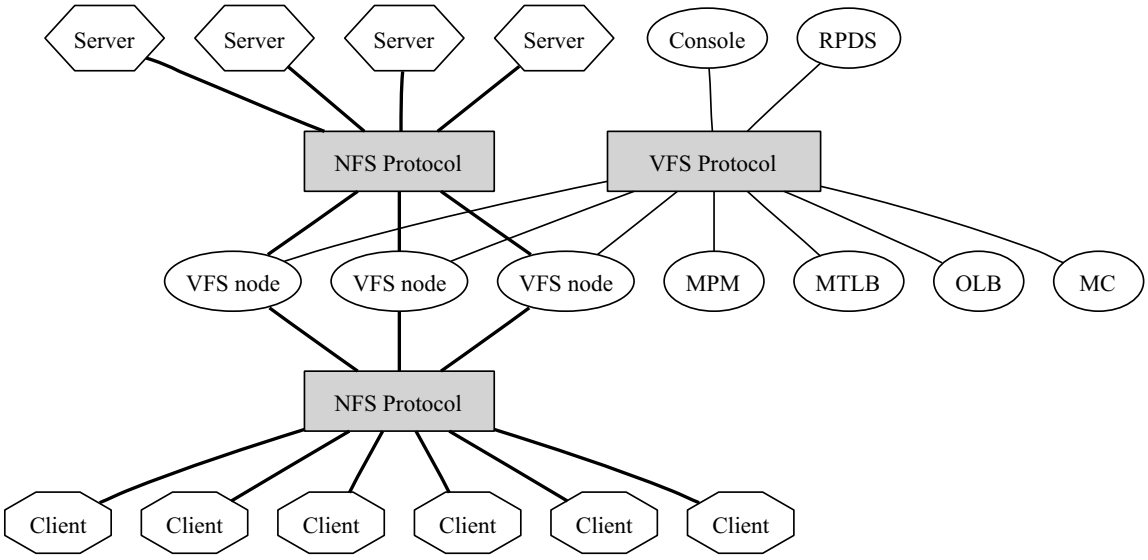

Figure 2.1: An example configuration showing the *v*FS architecture with four servers, three *v*FS nodes and six clients. All clients see the same coherent name space, but they mount the name space from different *v*FS nodes for load balancing purposes. *v*FS nodes communicate with servers and clients via standard NFS protocol. Besides the *v*FS nodes that directly deliver file service to clients, *v*FS also includes a Replicated Persistent Directory Service (RPDS) that provides global virtualization information shared by all *v*FS nodes, a Migration Coordinator (MC) that orchestrates the transparent migration process, a Migration Policy Manager (MPM) that automatically initiates migration based on predefined load/capacity/storage management rules, and a Mount-Time Lad Balancer (MTLB) and an Online Load Balancer that help balance out the load on the *v*FS nodes, and finally a management console that can be used to monitor the whole *v*FS system or manually initiate migration. All these various *v*FS components communicate with each other through a separate *v*FS protocol.

2.2 Architecture

The major component of *v*FS is the file service delivery layer that consists of lightweight *v*FS nodes placed in the data path between the clients and the file servers. These intermediate *v*FS nodes are transparent to both the clients and the servers: they behave like a file server to the existing, unmodified clients and act like a client to the existing, unmodified file servers. This is an important characteristic that

does not require the wholesale replacement of existing file servers and the client infrastructure. Figure 2.1 shows a *v*FS configuration with 3 *v*FS nodes, 4 servers and 6 clients. Typically, clients and servers are placed on isolated networks which can be either separate physical networks or separate VLANs. All communication between the clients and file servers is intercepted by the *v*FS nodes and request/reply packets are rewritten. This is transparent to the clients and servers, which continue to employ a standard file system protocol such as NFS or CIFS.

Besides these *v*FS nodes that directly deliver file service to clients, *v*FS also includes other components that assist virtualization, transparent migration, *v*FS node load balancing and management. All these various components communicate with each other through a separate *v*FS protocol that is explained throughout the rest of this thesis. The simplest component is the management console. The management console constantly collects load and health statistics from both the *v*FS nodes and the servers and present the aggregate results to administrators. From the management console administrators may also initiate data migration from one server to another or selectively shut down a few *v*FS nodes non-disruptively. *v*FS nodes do not persistently store any data, they rely on Replicated Persistent Directory Service (RPDS) for shared virtualization related data structures. RPDS is built upon Paxos [37] protocol for efficiency and reliability.

Each virtual volume in *v*FS is composed of a dynamic set of *subtrees* distributed among the file servers; though, such distribution is hidden from the clients. The Migration Coordinator (MC) is a component that orchestrates the transparent subtree migration process. Since the light-weight *v*FS nodes are on the direct data path between the clients and the servers, for robustness and simplicity concerns, they only implement very primitive operations critical to virtual volume consistency. It is

the MC that directs *v*FS nodes to cooperate with each other to accomplish the relatively complicated migration process. Although administrators can initiate migration through the management console for "one-shot" management tasks such as adding some *v*FS nodes or bringing some *v*FS nodes offline, migration is initiated more often automatically by the Migration Policy Manager (MPM) through pre-defined rules. MPM monitors the access patterns of data and migrates subtrees among the servers to make sure data is stored and served by servers best optimized for it. For example, hot data accessed by many clients frequently should be migrated to a high performance server and data very infrequently accessed should be migrated to low cost servers to lower the overall storage cost per byte. MPM also monitors the load and free capacity on file servers and migrate data as necessary to ensure both the load and data on file servers are properly balanced.

MC and MPM strive to ensure the load and capacity on file servers are balanced, the responsibility of the Mount-Time Load Balancer (MTLB) and the Online Load Balancer (OLB) is however to ensure the load on *v*FS nodes is balanced. A client may mount a virtual volume from any *v*FS node, similar to mounting a file system from a standard server, but MTLB directs different clients to different *v*FS nodes based on nodes's real time load information to ensure a good initial load distribution. Once a virtual volume is mounted from a specific *v*FS node, the client sends all file system requests to the chosen *v*FS node per the NFS protocol. However, *v*FS can still dynamically and transparently change the binding between the client and the *v*FS node through OLB which can dynamically change the identifers (IP addresses) *v*FS nodes bind to.

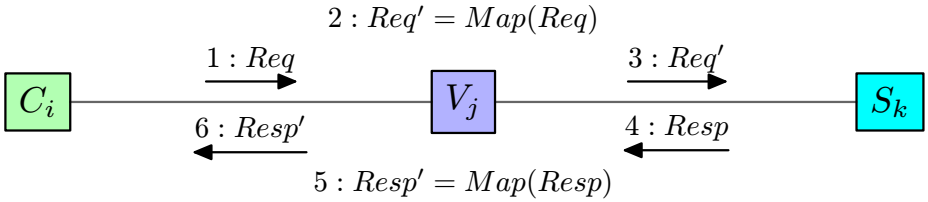

Figure 2.2: The six steps showing how a typical file system request is processed. In this configuration, client C_i mounted a virtual volume from vFS node V_j. C_i sends all requests to V_j. Most requests can be satisfied by contacting only one server say S_k. A very small fraction of the requests however may involve RPDS (not shown above).

2.2.1 Protocol Virtualization

Since a volume may consist of multiple subtrees from different file servers, vFS employs protocol virtualization to make these subtrees appear as one single file system tree to clients. Figure 2.2 shows how a typical file system request is processed. In the configuration, client C_i mounts a virtual volume from server V_j, therefore it sends all file system requests to V_j. Each such request Req only contains fields that V_j understands since to C_i, V_j is *the* server. Upon receiving Req, V_j parses Req and then consults a global light-weight virtualization database managed by RPDS or the node's own local copy to determine which servers need to be involved to process Req and how to translate Req into requests that file servers understand. For most requests, only one server, say S_k needs to be involved. V_j then rewrites Req and transforms it into Req' that server S_k can understand. In some cases, Req' may be very different from Req and therefore a full packet reassembly may be needed instead of simple packet rewriting.

Next, V_j sends Req' to server S_k. S_k then processes the request, just as it would handle a normal client request and sends the response $Resp$ back to V_j. V_j then parses $Resp$, translates it to $Resp'$ by consulting the same global virtualization database

again and then sends $Resp'$ back to client C_i after optionally asynchronously updating the database. Note that due to the "write-once read-many" nature of the majority of the virtualization database and *v*FS nodes' aggressive caching, *v*FS nodes very rarely need to consult RPDS for any virtualization needs.

Unlike previous designs, the *v*FS architecture aims to fully virtualize the file system protocol, aggregating the storage, computation and networking capacity of multiple file servers through the *v*FS nodes.

2.2.2 Transparent Migration

The full protocol virtualization in *v*FS enables transparent migration, that is, dynamically changing the location of files and directories by moving them among the file servers. This change is transparent to the clients, who can continue to access data both during and after the migration. This allows the virtualization layer to decide how much resources to consume from each server, simply by moving objects among them. Transparent migration is the fundamental mechanism for supporting server load/capacity balancing, online maintenance and retiring obsolete hardware without downtime or client reconfiguration.

The basic migration or distribution unit is a subtree which is stored in its entirety on one server. Figure 2.3 shows an example virtual volume composed of five subtrees distributed among three servers, each object annotated with a different shape (circle, square and diamond) to represent the server on which it is stored. Subtree$\{a, b, c, g\}$ is stored on the "circle" server; subtree$\{e, j, k\}$ and subtree$\{d, i\}$ are stored on the "square" server; subtree$\{f, l, m\}$ and subtree$\{h, n, o\}$ are stored on the "diamond" server[1]. The existence of these subtrees is transparent to clients who can perform any

[1]For simplicity, the rest of this thesis may use the root of a subtree to refer to the entire subtree.

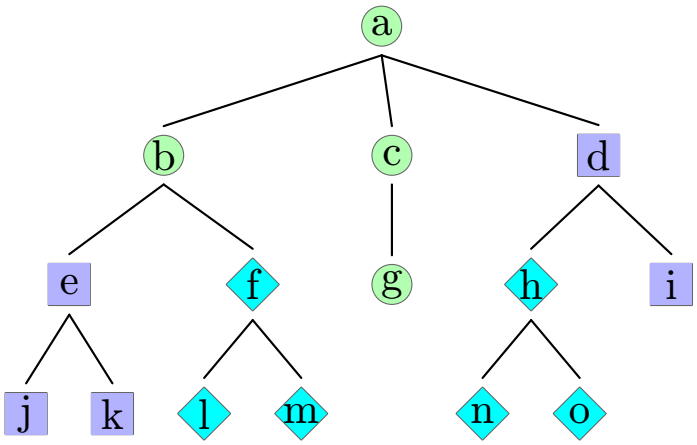

Figure 2.3: An example virtual volume consisting of five subtrees distributed among three servers. The shapes of the objects represent the servers on which the objects are stored.

legitimate operations on this virtual volume, oblivious to the fact that the operation may involve multiple objects stored on different servers. Each subtree can also have one or more asynchronous replicas which can be brought online for read-only access when the server that stores the primary copy is down.

The set of subtrees that a virtual volume consists of is dynamic. A new subtree of varying sizes can be created on the fly from anywhere within any existing subtree and migrated to another server. The boundary directories that "glue" subtrees stored on multiple separate servers together are called *junction directories* (jdirectory). The directories a, b, and d are examples of jdirectories in Figure 2.3. *v*FS nodes maintain additional soft state to facilitate processing of requests involving junction directories. Note that this construct does not exists in either NFS or CIFS, and is used only by the virtualization layer to aggregate multiple servers.

Not only can new subtrees be created as necessary, existing subtrees can also be merged with the parent subtrees from which they were created. For example, subtree$\{h, n, o\}$ is created from subtree$\{d, h, n, o, i\}$ which becomes subtree$\{d, i\}$.

Subtree$\{d, i\}$ is considered the parent of subtree$\{h, n, o\}$. If necessary, subtree$\{h, n, o\}$ can be merged with parent subtree$\{d, i\}$, reconstructing the original subtree$\{d, h, n, o, i\}$. The newly merged subtree, by definition, is also stored in its entirety on one server, Subtree merger comes in handy during subtree restructuring and server consolidation.

2.3 Related Work

This section examines the *v*FS work in the related literature classified into five categories: file system protocol virtualization, cluster file systems, storage virtualization, migration and utility computing.

2.3.1 File System Protocol Virtualization

Early file server virtualization schemes placed a single node between clients and servers to aggregate multiple existing homogeneous NFS or CIFS volumes [9, 31]. The Jade File System [53] takes a step further, and enables the naming and accessing of multiple existing heterogeneous file volumes exported through either NFS, FTP or Web in an internet environment by presenting a per-user logical name space with relaxed file system semantics. Anypoint [72] enables the automatic distribution of new directories or files to different NFS servers. Cuckoo [35, 59] improves aggregated throughput through read-only data replication.

Like all these schemes, *v*FS is also built on top of the proxy concept [62], using inter-positioning to add clustering to an existing client-server protocol. However in these existing schemes, data cannot move around once it is created and the middle node often becomes a single point of failure and performance bottleneck. More recently, commercial products such as Acopia [1] add the support for data location

independence, to the best of our knowledge, by fully storing and managing name spaces and metadata at the middle node, effectively treating the file servers as object stores only. This approach loses many optimizations at the file servers and renders the middle node even more prone to be a bottleneck. *v*FS differs from this work in three ways. Firstly, *v*FS is a scalable architecture in which file servers are managed by *multiple* *v*FS nodes that maintain access consistency semantics cooperatively; this architecture does not introduce a single point of failure or bottleneck. Secondly, *v*FS enables transparent fine-grained data migration ranging from any single file to the whole file systems whereas in Acopia, only pre-defined large partitions can be migrated. Thirdly, *v*FS exploits the optimizations at the specialized file appliances as much as possible, making the *v*FS nodes light-weight and easy to manage.

Slice [7] is a request routing proxy that can implement a virtual, scalable NFS server using a combination of specialized file servers and storage nodes. Unlike *v*FS , Slice cannot use existing unmodified NFS servers.

2.3.2 Cluster File Systems

Over the last two decades, numerous cluster and distributed file systems have been designed and implemented both in academia such as Andrew File System [28], xFS [8] and Frangipani [66] and in industry such as the open source Lustre [40], the Google File System [22] and the IBM General Parallel File System [61]. A cluster or distributed file system consists of multiple cooperative servers, clients are aware of all the servers and may contact different servers for data access. More recently, there have also been lots of attempts in the industry to make clustered NFS servers, such as Polyserve [52], Panasas [49] and SpinServer [34].

Cluster file systems provide scalability without sacrificing manageability. Despite

the success of these file systems in niche applications, NFS and CIFS continue to be highly popular, widely deployed, and supported by virtually all OSes. Managing and scaling the performance of these independent file server appliances continues to be a pressing problem.

In fact, the industry has been reluctant to embrace cluster file systems due to the excessive cost associated with the loss of existing investment in server hardware/software and administrator training and the lack of new (and potentially much more complex) client software for every platform and every operating system within a large enterprise. *v*FS on the other hand, provides most of the benefits cluster file systems provide without requiring the wholesale replacement of existing hardware/software infrastructure and administrators, presenting very *low deployment hurdle*.

2.3.3 Storage Virtualization and Federation

Storage virtualization and federation such as Federated Array of Bricks (fab) [57] and System Area Network (SAN) [63] storage have long focused on "block virtualization". Storage virtualization aggregates multiple storage devices and presents the illusion of one single disk with higher capacity, throughput and reliability. It is already a fairly well understood concept and technology, all major storage vendors now provide storage virtualization products such as Veritas Volume Manager and more recently Veritas Storage Foundation from Symantec [67], and PowerPath from EMC [20]. The *v*FS work is complementary to block virtualization, and targets file-based interfaces which still have to be provided by independent servers and the virtualization and federation of which are considerably more difficult due to their richer API and more complex semantics.

2.3.4 Migration

Databases, logical volume managers, disk arrays, and file systems have long provided support for transparent data migration. *v*FS introduces this capability to existing independently managed NFS servers. Almost all of these systems, including *v*FS, could throttle I/O performance during data migrations and can benefit from using systems like Aqueduct [39] to reduce such performance impact.

2.3.5 Utility Computing

Finally, the evolutionary utility computing (also known as on demand computing) vision advocates achieving adaptive infrastructures [30, 68, 10, 70] by aggregating existing server, network, and storage systems into a single, centrally managed pool of resources while maintaining full functional isolation. Our work on *v*FS bolsters this vision by enabling file service utilities.

Chapter 3

Protocol Virtualization

Protocol virtualization breaks the bindings between clients and file servers, aggregates all the resources of file servers and presents the illusion of consistent virtual volumes, similar to physical volumes exported by a single file server. This chapter presents the challenges NFS protocol virtualization faces and how *v*FS tackles them. It starts with a brief overview of the NFS protocol in Section 3.1 and introduces file handles, a very important concept in protocol virtualization. Section 3.2 highlights a few requirements that need to be satisfied for protocol virtualization to be fully transparent to clients. The details of virtualization are presented in Section 3.3 and Section 3.4, with focus on ID space and name space virtualization respectively. Section 3.5 tries to answer the question of how *v*FS nodes exporting the same virtual volume manage to present the same coherent image to clients. Section 3.6, 3.7 and 3.8 explain the details of virtualizing the special NFS procedures, the authentication schemes and the lock service. Section 3.9 presents the lightweight Replicated Persistent Directory Service built on top of the Paxos protocol that manages global information shared by all *v*FS nodes. The last section summarizes.

3.1 NFS Overview

The NFS protocol provides transparent remote access to shared files stored on a single server to a set of clients [60]. It was designed to be machine, operating system, network architecture and transport protocol independent. In order to achieve such independence, client-server communication is based on Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR [71]). The XDR specification provides a standard way of representing a set of basic data types such as string, integer, union, boolean and array in heterogeneous environments. More complex data structures can be built from these basic types. XDR therefore solves the problem of different sizes, different byte orders, different alignment conventions and different data type representations on different machines. There are currently two versions of NFS protocols in wide use: NFS version 2 [45] and NFS version 3 [46]. Our virtualization work is based on NFS version 3, but the design applies to both versions. Hereafter we will use the term "NFS" or "NFS protocol" to refer to NFS version 3 unless otherwise explicitly stated.

The NFS protocol is stateless. The arguments to each procedure call request contain sufficient information for the server to complete the call and the server does not keep track of past procedure calls. When a server crashes and then restarts, it does no crash recovery at all other than checking the underlying local file systems for errors. Clients merely resend procedure calls, potentially with exponential backoff, until a response is received, without even being able to differentiate between server crashes and servers simply being slow. From our experience, the stateless design of NFS protocol also makes the virtualization complexity more manageable since *v*FS nodes may virtualize each procedure call independently, avoiding the need to keep

ID	Name	Brief Description
0	NULL	Test service availability and service latency
1	GETATTR	Retrieve the metadata of an object
2	SETATTR	Modify the metadata of an object
3	LOOKUP	Resolve an object name to its FH
4	ACCESS	Check if given user has given access to given object
5	READLINK	Read the content of a symbolic link
6	READ	Read a chunk of data from a file
7	WRITE	Write a chunk of data to a file
8	CREATE	Create a file under a given directory
9	MKDIR	Create a directory
10	SYMLINK	Create a symbolic link for a given object
11	MKNOD	Create a special device node
12	REMOVE	Delete a file
13	RMDIR	Delete an empty directory
14	RENAME	Rename a given object
15	LINK	Create a hard link
16	READDIR	Read a portion of a directorys content
17	READDIRPLUS	Read a portion of a directorys content and FH
18	FSSTAT	Retrieve the dynamic file system information
19	FSINFO	Retrieve the static file system information
20	PATHCONF	Retrieve the POSIX information for a given object
21	COMMIT	Flush cached data on the server to stable storage

Table 3.1: Brief summary of NFS procedure calls.

track of procedure calls clients made in the past.

The NFS protocol provides 22 procedure calls to clients to request access or make changes to the file systems on a NFS server. These procedures are very briefly summarized in Table 3.1. Some of the procedure calls allow clients to query system wide information such as file system capacity and file system free space; some of them allow clients to read and write files; some of them allow clients to create or delete existing files or directories; the rest of them enable clients to traverse the file system namespace hierarchy.

3.1.1 File Handles

One of the most important concepts in the NFS protocol is *File Handle* (fhandle). Fhandles are created and issued by servers and are used to uniquely identify objects (files or directories) on servers. The exact format and content of fhandles are implementation specific, but fhandles should be unique among all objects ever created on a server, including those created and then deleted. Clients treat fhandles as sequences of opaque bytes, and can only pass them as arguments to procedure calls to refer to different objects.

A NFS client obtains from a server the first fhandle, the root fhandle of an exported volume $FH_{/}$, at mount time through a separate MOUNT protocol [46]. The fhandles of other existing objects are usually resolved through LOOKUP calls component by component. Given the fhandle of a directory and the name of an object under that directory, LOOKUP returns the fhandle for the object. For example, in order to get the fhandle of file `/usr/bin/bash` $FH_{/usr/bin/bash}$, a client needs to get the fhandle of `/usr` $FH_{/\texttt{usr}}$ through LOOKUP($FH_{/}$, "`usr`") first, then get the fhandle of `/usr/bin` $FH_{/usr/bin}$ through LOOKUP($FH_{/usr}$, "`bin`"), and finally get the desired $FH_{/usr/bin/bash}$ through LOOKUP($FH_{/usr/bin}$, "`bash`"). LOOKUP resolves the name of one object at a time, READDIRPLUS on the other hand returns the fhandles along with the names and other attributes of all the children under a given directory. If the directory is too large, multiple READDIRPLUS requests may be required.

When a new object is created, the fhandle of the newly created object is returned as part of the procedure call response. Related procedure calls include CREATE for creating a new file, MKDIR for creating a new directory, LINK for creating a hard link, SYMLINK for creating a symbolic link and MKNOD for creating a new special device file.

3.1.2 Compromises to the Stateless Design

The stateless design of NFS allows a server to respond to different procedure calls independently. Should the need to associate multiple procedure calls arise, a NFS server returns some temporary identifiers as part of the reply; clients then use them as arguments to subsequent procedure calls. A NFS server relies on such identifiers to track the progress of a series of calls or detect whether the server state has changed since previous calls. For example, clients may want to read the content of a huge directory that does not fit in one RPC message through multiple READDIR or READDIRPLUS requests[1]. In response to each such request, a server fills in a *directory cookie* representing an offset within the directory of interest. Clients then send the directory cookies in subsequent READDIR or READDIRPLUS requests to read the rest of directory entries. While the large directory is being read, another client may step in and delete existing objects or create new objects in the directory and hence invalidate the previously issued directory cookies. To handle such cases, a server also issues a *directory cookie verifier* as part of the response along with a directory cookie to identify different versions of the same directory.

For performance considerations, NFS supports asynchronous writes. In an asynchronous write, a client sends data to a server through a WRITE call with an `unstable` flag followed by a COMMIT call some time later. The server does not need to flush data or metadata to stable storage until it receives the COMMIT call. Asynchronous writes not only allow a server to aggregate more data to write at once and therefore more effectively utilize disk bandwidth, but also give clients much better response time. However, if the server crashes before flushing the data to stable storage and

[1]The difference between READDIR and READDIRPLUS is that the latter returns the fhandles of directory entries and the former does not.

then receives COMMIT after a reboot, it will respond to COMMIT with a success as it is not aware of the data loss. In order to give clients enough information to safely determine whether the server could have lost data and whether the data needs to be retransmitted, both WRITE and COMMIT return a server instance identifier called *write verifier* that changes after each reboot.

The majority of NFS procedure calls are idempotent — executing the call once or multiple times leads to the same result — hence they are safe in the presence of retransmissions. However there are exceptions and CREATE is one such example. In order to support exclusive creation semantics, NFS allows a client to specify a unique identifier called *create verifier* in a CREATE request. The server stores the create verifier in stable storage, associates it with the newly created file, and returns success rather than failure if it sees a duplicate request with the same create verifier.

3.2 NFS Protocol Virtualization Requirements

The goal of protocol virtualization was to break the bindings between clients and file servers, and to aggregate the storage, computation, and networking capacity of all servers together. We take the extreme approach of transparent virtualization where unmodified clients and servers run unmodified file system protocol and a virtualization layer transparently interposes on the traffic between the client and the servers. This approach is also known as *in-band* virtualization.

There are three main requirements and challenges protocol virtualization faces. The first requirement is to aggregate multiple ID and name spaces independently managed by different servers, form one consistent name space and one consistent ID space for each virtual volume, and present them to clients. When this requirement

is met, clients should not be able to observe ID or name clashes. Neither should clients be able to observe the fact that objects may be spread over multiple servers and multiple storage devices. The second requirement is to efficiently locate and access pertinent file servers and the objects with as little processing as possible in the *v*FS layer, given a NFS request. The last but not least requirement is to guarantee the file system semantics with the same level of consistency as NFS to all clients accessing the same virtual volume, potentially through multiple different *v*FS nodes; and to ensure that existing ad hoc NFS caching algorithms on clients work flawlessly. On top of these requirements and challenges, we need to also keep in mind one additional implicit requirement: all virtualization effort should use only the standard NFS operations from NFS servers.

Clients interact with NFS servers only through various identifiers and the virtual volume name space, if *v*FS manages to present a coherent ID space and a coherent virtual name space just as a single NFS server, clients will not be able to tell the difference. The next three sections will explain in detail how the ID spaces and name spaces of multiple servers are aggregated and how the virtual volume consistency semantics is achieved.

3.3 ID Space Virtualization

The NFS ID space contains various long-lived permanent identifiers such as fhandles, file system IDs, file IDs and device IDs which uniquely identify different file systems, files[2] and devices respectively *within the same server*. The NFS ID space also contains various temporary identifiers such as directory cookies, directory cookie verifiers, write

[2]File IDs are only unique among live objects, similar to inode numbers on Unix. Fhandles however are guaranteed to be unique for all objects ever created.

verifiers and create verifiers. Since all these IDs opaque to clients are independently managed by different servers, they pose two problems: a client dealing with multiple servers may experience ID conflicts; and the same object, after being migrating to a different server, is almost guaranteed to have a different ID. Both problems confuse client software and may trick them into treating different files as the same or treating the same file as having changed. Existing ad hoc caching algorithms also almost always break.

*v*FS's solution is for the *v*FS nodes to issue virtual IDs, completely hide physical IDs from clients and perform the translation between virtual and physical IDs. *v*FS nodes communicate with clients using only virtual IDs and communicate with NFS servers using only physical IDs. Furthermore, *v*FS nodes guarantee that virtual IDs are unique and survive object migrations. The next two sub-sections provide more details on the different approaches *v*FS employs in virtualizing permanent IDs and temporary IDs.

3.3.1 Permanent ID virtualization

*v*FS nodes issue *virtual file handles* (vhandle) to clients for the objects stored in virtual volumes; akin to *physical file handles* (phandle) issued directly by NFS servers. *v*FS also explicitly maintains a mapping table called v-table, each entry of which associates the vhandle of an object with the IP address of the current server storing the object and the phandle it issues for the object. The server address and the phandle together are hereafter called a <server, phandle> pair. Before settling down on the explicit v-table, two other alternatives were considered and neither of them fully satisfies our requirements. The first alternative embeds <server, phandle> in its corresponding vhandle, an approach employed by all existing NFS virtualization effort [9, 72] to

the best of our knowledge. This alternative suffers from two problems: the changing vhandle problem since the <server, phandle> pair for an object changes once it is migrated; and the fhandle length limit problem which prevents a vhandle from being constructed if the length of <server, phandle> exceeds the fhandle length limit. The second alternative computes the hash of <server, phandle> as the corresponding vhandle. Although very elegant, this approach also suffers from the changing vhandle problem, and in addition one still can not translate a vhandle back to the <server, phandle> pair. Since clients may access the same virtual volume through different *v*FS nodes, to guarantee the consistency of vhandle space, the v-table is managed by a shared light weight Replicated Persistent Directory Service (RPDS, described in section 3.9).

Query is the most common operation on v-table. Every NFS procedure call includes at least one fhandle in its arguments[3], and therefore the virtualization of every procedure call involves the translation of vhandles to <server, phandle> pairs. In order to minimize the overhead introduced by this translation, each *v*FS node aggressively caches a local *in-memory* copy of the v-table. The local copy and the RPDS copy are called local v-table and global v-table respectively. A *v*FS node may supply a vhandle and retrieve one v-table entry from RPDS or supply a vhandle prefix and retrieve the v-table entries for all the objects with the matching prefix. Given the need to translate a vhandle, a *v*FS node always checks its local v-table first and then performs the translation if the corresponding entry is found. Otherwise, *v*FS node consults RPDS for the entry. Besides this on-demand "page-in" of v-table entries, each *v*FS node also periodically pulls from RPDS new v-table entries to reduce v-table lookup misses which may directly increase the user perceived latency of NFS

[3]Except NULL which is not a "real" NFS procedure call.

operations. In addition to translating vhandles to <server, phandle> pairs in every procedure call request, *v*FS nodes also need to translate <server, phandle> pairs back to vhandles for every LOOKUP and READDIRPLUS response. The process is very similar to the translation in the other direction.

The v-table expands as part of the process of virtualizing CREATE and MKDIR procedure calls when new vhandles and new v-table entries are created. Upon receiving the response for a CREATE or MKDIR request, the *v*FS node does six things: (1) allocating a new vhandle; (2) creating the corresponding v-table entry, inserting it into its local v-table and marking it as non-evictable; (3) rewriting the response with the new vhandle; (4) sending the response back; (5) sending the new vhandle and v-table entry to RPDS for persistent storage; and (6) removing the non-evictable flag on the new v-table entry when the confirmation from RPDS comes back. (3)(4) and (5)(6) happen in parallel so as to avoid introducing unnecessary virtualization latency. Note also that the allocation of vhandles and the construction of new v-table entries are done independently by multiple *v*FS nodes to reduce latency and improve throughput. So as to avoid vhandle clashes, *v*FS nodes only allocate vhandles from its own local chunk of vhandle space, containing about a few thousand of vhandles. Before the local chunk of vhandle space runs out, *v*FS nodes request a new chunk from RPDS.

The v-table does not just expand as the result of MKDIR and CREATE. Some procedure calls such as RMDIR and DELETE delete objects stored on a server, rendering their phandles and therefore the corresponding vhandles and v-table entries useless. A mechanism to prune the v-table therefore is also required. Unfortunately, although *v*FS nodes know exactly which vhandles are created and when, they do not possess the information about when and which vhandles can be deleted. A DELETE or RMDIR

request only includes the name of the object being deleted but not its vhandle. One possible solution is to maintain a consistent mapping between the vhandle and the name of an object, but it leads to much higher storage requirement on *v*FS nodes and RPDS and requires strong coherence protocol among all the *v*FS nodes exporting the same virtual volume and thus is very expensive.

*v*FS's final solution exploits the uniqueness of fhandles in NFS protocol and uses a lazy garbage collection technique to prune the obsolete entries. In the background, RPDS periodically scans the global v-table and for each phandle sends a harmless PATHCONF request to the server currently hosting the object. If the object has been deleted or the phandle is obsoleted, the server returns `NFS3ERR_STALE` or `NFS3ERR_BADHANDLE` in which case RPDS may safely remove the pertinent v-table entry. Such a scan of the whole v-table is expensive, RPDS in fact maintains an extra last-accessed timestamp field with each v-table entry and only those entries that have not been accessed or scanned for a long time are scanned.

The virtualization of other permanent IDs can also follow the way fhandles are virtualized, but doing so certainly involves extra complexity and extra performance hit. We made three important observations: (1) the content of a vhandle can be any random bits as long as the uniqueness is guaranteed (2) other permanent IDs such as file system ID, file ID, and device ID are short and (3) they do not change throughout the lifetime of a vhandle. Therefore, different fields of vhandles can be used to store these IDs. The format of the initial version of vhandles includes only a one byte virtual volume ID and a 10 byte virtual file ID. Virtual volume IDs uniquely identify different virtual volumes and all the vhandles issued for the same virtual volume share the same virtual volume ID. Virtual volume IDs are also returned to clients as file system IDs and device IDs; a virtual volume thus appears to be hosted on one file

Field	**Name**	**Size** (bytes)
1	Virtual volume ID	1
2	Parent vhandle summary	3
3	Virtual file ID	10
4	Hash	0 - 50

Table 3.2: The format of vhandles. Virtual volume IDs uniquely identify virtual volumes, file systems and devices. Virtual file IDs uniquely identify different objects within the same virtual volume. Parent vhandle summary gives all vhandles under the same directory the same prefix and Hash protects *v*FS nodes from forged vhandles.

system and one storage device. Different vhandles in the same virtual volume have different virtual file IDs which are allocated by *v*FS nodes as described.

Later on two more fields were introduced into vhandles: a 3 byte parent vhandle summary field and an optional hash field. Table 3.2 briefly describes all four different fields. The parent vhandle summary is a short hash (such as the last 3 bytes) of the virtual file ID of the object's parent. Note that all the vhandles under the same directory have the same (virtual volume ID, parent vhandle summary) prefix. *v*FS nodes use exactly this prefix to retrieve all the v-table entries for the objects under the same directory[4]. The hash field is calculated by running a secure hash function on the virtual volume ID, virtual file ID, parent vhandle summary and a secret known only to *v*FS. This hash field is a measure to guard against clients using the *v*FS nodes as gateways to gain access to servers by forging vhandles. The secure hash function can be any existing secure hashing schemes such as SHA-1 or SHA-256 [44] as long as the hash result is shorter than 50 bytes. With this scheme, it is computationally expensive for clients to forge a vhandle without being detected and it is relatively easy for any *v*FS node to verify the validity of a vhandle presented to it.

[4]The probability that different vhandles have the same summary is low.

The hash field is not stored in either local v-tables or the global v-table. Typically, phandles are shorter than 15 bytes, so conservatively a v-table entry is shorter than 40 bytes including a 1 byte server ID (mapped through a small table to the server's IP address) and a 2 byte last-accessed timestamp. A *v*FS node with 4GB RAM can store 100 million entries for the most frequently accessed objects, and a RPDS service with 40GB storage can store a v-table for one billion objects.

3.3.2 Virtualization of other IDs

Write verifiers identify different incarnations of the NFS service on a particular server. They are used to facilitate the detection of server reboots by clients such that clients can resend data contained in previous `unstable` WRITE procedure calls that may not have been committed to stable storage before the server rebooted. As an object is migrated to a new server, the write verifiers issued by this new server may confuse clients, therefore *v*FS also issues virtual write verifiers and translates between virtual and physical ones. The initial design tries to pass virtual write verifiers between one *v*FS node and another in the event of *v*FS node failover, and naturally the solution gets RPDS involved since the behavior of a failing node can not be relied upon. The final design was simplified by one important observation: WRITE calls are idempotent and requesting clients to resend data adds overhead but does not violate NFS consistency semantics. Each *v*FS node independently allocates virtual write verifiers. Virtual write verifiers have a one byte *v*FS node ID field, a 4 byte current time field and a 3 byte monotonically increasing ID field to ensure that all the virtual write verifiers allocated by *v*FS nodes are unique. Like fhandle virtualization, each *v*FS node also maintains a translation table between virtual write verifiers and <server, physical write verifier> pairs, but this table is purely local to each node and RPDS is

completely unaware of it. Whenever objects are migrated from one server to another or when one *v*FS node takes over another *v*FS node's responsibility, *v*FS creates the illusion that the NFS server has been rebooted so that clients can resend data. Since both migration and failover events are very rare and clients do not accumulate much data before COMMITting them, *v*FS throughput is hardly affected.

Creation verifiers are a client side concept, no virtualization is necessary. The virtualization of directory cookies and directory cookie verifiers is described in the next section since they are used for name space operations.

3.4 Name Space Virtualization

Each virtual volume in *v*FS has its own name space constructed by virtualizing the name spaces of all the underlying file servers. For the clients, the name space is similar to that of a centralized NFS server even though the objects in a virtual volume may span multiple servers. For the most part, the *v*FS design seeks to leverage file servers to implement name space operations by storing each subtree in its entirety in a file server. This way, name space operations within a subtree are handled by the file servers with *v*FS nodes interposing on the requests and replies only to apply the ID space virtualization.

3.4.1 Subtree

The root directory of each NFS server managed by *v*FS has exactly N (typically 4) first level directories named 0, 1, 2, $\cdots$, $N-1$ respectively. Each first level directory also has N second level directories, also named 0, 1, 2, $\cdots$, $N-1$ respectively. Second level directories are the parents of all subtrees. The first and second $\log N$ bits of the

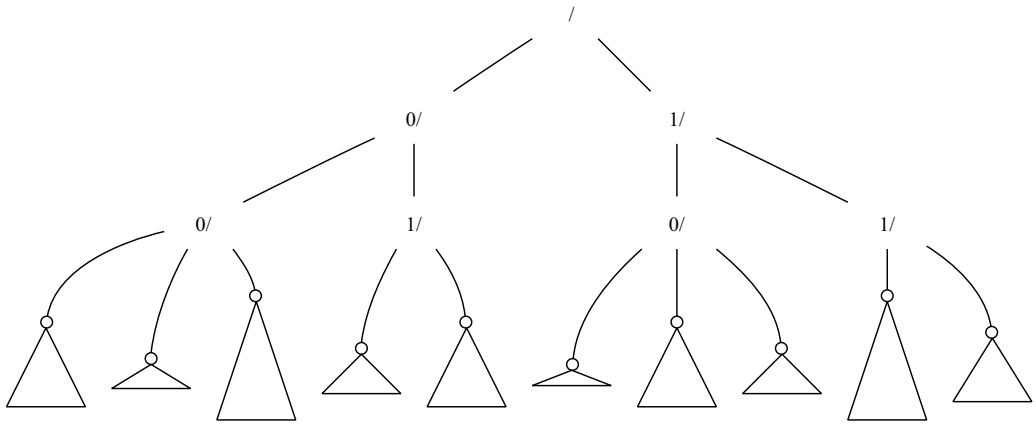

Figure 3.1: The typical layout of the file system of a *v*FS managed NFS server ($N = 2$). In the graph, each small triangle represents one subtree.

virtual file ID of a subtree root determine under which second level directory the subtree should be hosted. For example, for all the subtrees hosted under directory `/2/3/`, the first and second 2 ($N = 4$) bits of the virtual file ID of their roots are 2 and 3. Figure 3.1 shows the typical layout of a NFS server's file system hosting 10 subtrees, each represented by a small triangle. In total, each server is expected to have about a few tens of subtrees.

Individual subtrees spread across multiple NFS servers are not visible to clients, instead, they are "glued" together through *attachment* operations to construct one single tree which is the virtual volume clients observe. An attachment operation attaches the root of one subtree to a directory (called host directory) in a different subtree (called host subtree), usually stored on a different server. More than one subtree can be attached to the same directory, one at a time. Once an attachment operation finishes, the host directory appears to clients as one normal directory containing both the original children it had before the operation and all the children of the subtree root just attached, although the latter are physically stored on a different

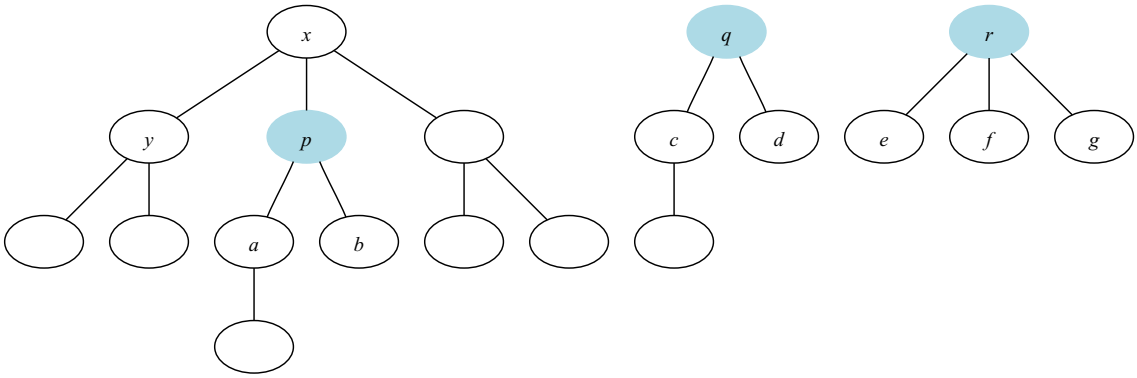

Figure 3.2: Three subtrees hosted on three different servers. Subtree q and subtree r are to be attached to directory p in subtree x one after another. Directory p has direct children a and b; directory q has direct children c and d; and directory r has direct children e, f and g.

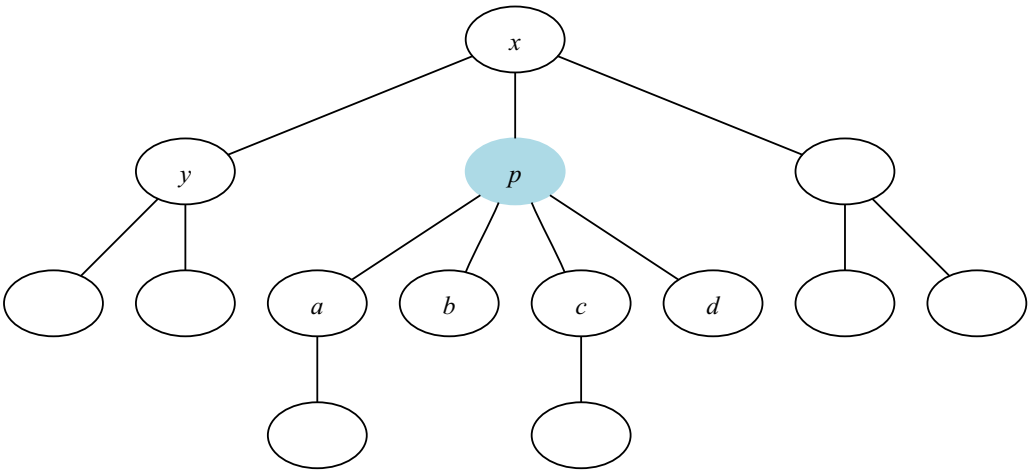

Figure 3.3: The client view of subtree x after q has been attached to directory p. Object a, b, c, and d all appear to be direct children of directory p to clients.

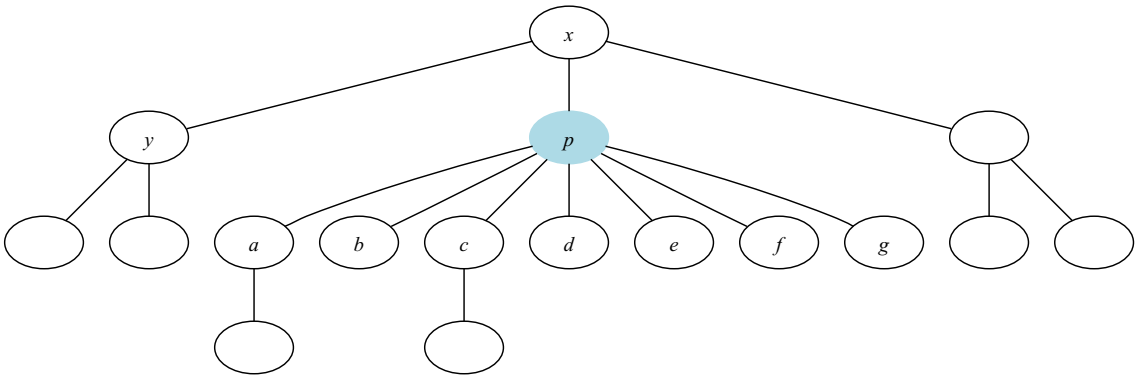

Figure 3.4: The client view of subtree x after r has also been attached to directory p. Object a, b, c, d, e, f and g all appear to be direct children of directory p to clients.

server. Figure 3.2 shows three subtrees x, q and r hosted on three different servers. Figure 3.3 shows the client view of subtree x after subtree q is attached to directory p in subtree x and Figure 3.4 shows the client view of x after subtree r is also attached to p. The client view of a directory after attachment operations, such as that of p in Figure 3.4, is called a *junction directory* (jdirectory) which is different from the underlying host directory.

3.4.2 Virtualization

Name space operations on a virtual volume can be divided into three categories: category I operations only involve objects within one subtree; category II operations involve objects from 2 subtrees; and category III operations involve objects at the boundaries between subtrees, i.e. jdirectories. Since each subtrees is stored in its entirety on a single server, category I operations are simply forwarded by *v*FS nodes to the server that stores the subtree involved. For example, in Figure 3.4, only ID space virtualization is required for a CREATE request trying to create a new file under directory a or a READDIR request trying to list the content of directory c. Since each

jdirectory has a physical counterpart: the host directory, requests that operate on the parents of jdirectories also fall in category I and hence can be directly handled by servers. For example, a READDIR request listing the content of x in Figure 3.4 will see jdirectory p, and a request to RENAME jdirectory p to y will fail since the server storing x can detect the name clash due to the existence of another directory y under x. By allowing file servers to serve most of the metadata operations which can be dominant in a lot of workloads [19], the *v*FS architecture fully exploits the optimizations at the servers and keeps the *v*FS nodes light-weight.

The only type of name space operations that falls into category II is RENAME requests renaming an object from one subtree to another subtree. The initial design of *v*FS supports the virtualization of category II operations, however as we gained more and more experience building the system, it became evident that the complexity introduced is not justified by the value added and hence the support for category II operations was dropped. For example, supporting category II operations requires very strong synchronization between RPDS and all the *v*FS nodes. RENAME operations are very uncommon and the majority of them only involve objects within the same directory. *v*FS's support for RENAME operations within the same subtree is sufficient for almost all applications.

Category III requests are those that operate *on* jdirectories, which appear to clients no different from any other normal directories. For example, a client listing the content of p through READDIR requests shall see object a, b, c, d, e, f and g. It shall be able to create a new object or deleting an existing object under p. Furthermore, it shall also be able to rename objects within p, such as renaming object a to e, which are physically managed by two separate servers. The illusion of jdirectories is provided through RPDS and all category III requests are forwarded to RPDS by

*v*FS nodes in order to preserve a cohesive view to clients. For each jdirectory, RPDS maintains an ordered list of the root vhandles of attached subtrees. In the case of p in Figure 3.4, the list is (vhandle-q, vhandle-r). For each child under a jdirectory, RPDS also maintains an entry that associates the vhandle of that child with the name of that child and the vhandle of the jdirectory. The table that stores all such entries is called *j-table*. For p in Figure 3.4, RPDS keeps the following entries in the j-table:

<vhandle-p, a> $\leftrightarrow$ vhandle-a

<vhandle-p, b> $\leftrightarrow$ vhandle-b

<vhandle-p, c> $\leftrightarrow$ vhandle-c

<vhandle-p, c> $\leftrightarrow$ vhandle-d

<vhandle-p, d> $\leftrightarrow$ vhandle-e

<vhandle-p, e> $\leftrightarrow$ vhandle-f

<vhandle-p, g> $\leftrightarrow$ vhandle-g

Note that RPDS does not store, cache or manage the metadata of any children of a jdirectory. The virtualization of different category III requests uses the j-table and the v-table in very different ways as described below:

LOOKUP: when RPDS sees a LOOKUP request that tries to resolve the name of an object to its file handle, it takes the vhandle of the jdirectory and the name of the object from the request, consults the j-table it maintains and sends the resolved vhandle directly back to clients.

CREATE/MKDIR: a CREATE or a MKDIR request requires that the new object created have a unique name in its parent directory; since none of the constituent subtrees, neither the host subtree nor the attached subtrees, have the complete view of the jdirectory, it is the responsibility of RPDS to check for name clashes

before forwarding the requests to servers for object creation. By default, all legitimate new object creation requests are forwarded to the server that manages the host subtree for processing. However, *v*FS also allows a creation policy to be associated with each jdirectory which supports load and capacity balancing by controlling where new objects are created. Currently three simple policies are supported: a static policy which directs new objects to one designated constituent subtree; a round robin policy which directs new objects to all the constituent subtrees in a round robin fashion; and a hash policy which directs new objects to all the constituent subtrees based on new object names. Once the response comes back, RPDS updates its j-table; and like any other CREATE or MKDIR request, RPDS also updates the v-table.

READDIR/READDIRPLUS**:** a READDIR or READDIRPLUS request lists the children of a directory, their attributes and file handles (in the case of READDIRPLUS). If a directory has too many children and the response does not fit into one message, the NFS server sets the `EOF` flag in the response message to false, and returns a directory cookie that essentially represents how many children's information has been returned and a directory cookie verifier that effectively represents the update time stamp/version of the directory. The client, upon seeing the the false `EOF` flag, sends subsequent READDIR or READDIRPLUS requests until the `EOF` flag is set to true in the response. RPDS exploits this support for large directories and returns the content of a jdirectory through multiple request/response exchanges. Again we use jdirectory p as one example, assuming host directory p (not jdirectory p) is very large and its content can be returned to clients through 2 messages; and directory q and r are fairly small and their content can fit into one message, a client C needs to go through the following 4 steps in order to

retrieve all the content of jdirectory p:

1. C sends READDIR(vhandle-p, 0)[5] to the vFS node V from which it mounted the virtual volume. V forwards the request to RPDS which upon seeing the cookie value 0, knows C intends to read the content of p from the beginning, and therefore sends request READDIR(phandle-p, 0) to S_p, the server hosting subtree x. S_p sends back to RPDS the first portion of the content of directory p: RESPONSE(content, cookie-1, false) where cookie-1 is a directory cookie and false is the `EOF` flag. RPDS creates a virtual directory cookie vcookie-1, creates an association between vcookie-1 and (cookie-1, vhandle-p) in its *vcookie table* and then sends RESPONSE(content, vookie-1, false) to V which in turn sends it back to C.

2. C sends READDIR(vhandle-p, vcookie-1) to RPDS through V. RPDS looks vcookie-1 up in its vcookie table, learns that vcookie-1 is issued by S_p for directory p, and sends READDIR(phandle-p, cookie-1) to S_p which returns the second portion of the directory content: RESPONSE(content, cookie-2, true) where true means S_p has no more content to return. RPDS creates a virtual directory cookie vcookie-2, creates an association between vcookie-2 and (0, vhandle-q), and sends RESPONSE(content, vcookie-2, false) back to C through V. Note that V rewrites the `EOF` flag to false.

3. C sends READDIR(vhandle-p, vcookie-2) to RPDS via V. Seeing vcookie-2, V learns about the directory reading progress and issues READDIR(phandle-q, 0) to S_q, the server hosting directory q. S_q replies with the full content of q in RESPONSE(content, cookie-3, true). Upon receiving the re-

[5] 0 is a special directory cookie value representing the beginning of a directory.

ply, V creates vcookie-3 and an association between vcookie-3 and (0, vhandle-r) since it knows that q has no more content to offer, and sends RESPONSE(content, vcookie-3, false) back to C via V.

4. C sends READDIR(vhandle-p, vcookie-3) to RPDS via V. V issues READDIR(phandle-r, 0) to S_r, the server hosting directory r. S_r replies with the full directory content in RESPONSE(content, cookie-4, true). Upon receiving the reply, V finally sends RESPONSE(content, 0, true) back to C via V, signaling that there is no more content to read for jdirectory p.

Since the virtual directory cookies are meaningful only during the process of reading directories, RPDS only keeps them in main memory for a very short while, such as 1 minute. Since clients are already aware of possible cookie expiration, clients will simply try to read the jdirectory from beginning afresh in case virtual directory cookies expire.

The above description did not mention virtual directory cookie verifiers. They are incremented by one each time the related jdirectory is modified or when virtual directory cookies for the jdirectory expire. They are also sent back in the response to READDIR or READDIRPLUS as a normal NFS server would do.

RENAME: a RENAME operation renames a `src` object to a `dst` object. If `dst` does not exist or `dst` is hosted on the same server as `src`, such as RENAME(c, d) and RENAME(e, f) under jdirectory p in Figure 3.4, RPDS does three things: (1) forwarding (with rewriting) the request to the server hosting `src`; (2) updates its j-table based on whether the operation succeeds or not; and (3) sends a (rewritten) reply back. If however `src` and `dst` are located on two different servers, RPDS essentially needs to use a combination of a DELETE request and

a RENAME request to emulate one single RENAME operation issued by clients, as outlined in the following steps:

1. RPDS sends RENAME(`dst`, `uniq`) request to the server manages `dst` where `uniq` is a unique name generated by RPDS on the fly.
2. If RENAME(`dst`, `uniq`) succeeds, RPDS removes `dst` from its j-table, and sends RENAME(`src`, `dst`) to the server hosting `src`.
3. If RENAME(`src`, `dst`) succeeds, RPDS updates its j-table, sends the reply back to the client and potentially asynchronously issues DELETE(`uniq`) to the server that manages `uniq` to delete it.
4. Otherwise RPDS issues RENAME(`uniq`, `dst`) to the server manages uniq to bring `dst` back to normal.

RPDS serializes such cross-subtree RENAME requests to ensure their atomicity. The tactic to RENAME rather than DELETE `dst` first avoids the partial-failure state where `dst` is gone but `src` is not renamed.

3.5 Virtual Volume Consistency

In the *v*FS design, multiple clients can access a virtual volume using any of the *v*FS nodes that exports it. It is critical to keep the file access semantics unchanged despite the underlying implementation being distributed. *v*FS design maintains the access semantics using a simple mechanism referred to as *coordination point preservation*. This means that requests that used to meet each other at a coordination point continue to do so when *v*FS is interposed between the servers and clients. In the traditional NFS, the coordination point is always the NFS server. In *v*FS, the

coordination point is the underlying file server storing the data for the most cases. Only requests on jdirectories require special care, and in these cases RPDS serves the role of the coordination point. *v*FS nodes do not cache data or metadata; there are no NFS operations performed by a *v*FS node that do not end up o n a file server.

*v*FS is able to preserve the access semantics by supporting the following property through the coordination point preservation mechanism: clients are able to observe changes made by other clients if and only if those changes are visible without the *v*FS layer. For example, if a file's content is changed by one client, other clients will be able to observe that change because the final data always comes from the physical file stored on a file server. Similarly, two objects with the same name can not be created under the same directory since the real creation requests are either carried out by a file server or checked by RPDS.

3.6 Special Procedure Virtualization

There are two operations that *v*FS does not support. The first one is LINK which creates a hard link to an existing object. There are three reasons why *v*FS elected not to support it. Firstly, in the context of *v*FS, the concept of cross-server hard links is very difficult to enforce and it also prevents us from using data migration as the fundamental mechanism for load and capacity balancing. Secondly, since the introduction of symbolic links, hard links became almost completely obsoleted due to its inflexibility. Lastly, hard links are not a feature required by NFS protocol and in fact, even quite a few modern file systems do not support hard links. *v*FS does not support MKNOD and the concept of device files either yet. This feature is very rarely used and can be added relatively easily if needed, unlike hard links.

There are three NFS operations that are not directly tied to specific file system objects although they do pass along a file handle to locate a specific file volume: FSSTAT, FSINFO and PATHCONF[6]. FSSTAT retrieves volatile volume state information, including maximal volume size, volume free space, maximal number of objects supported, number of additional objects that can be created. Since each volume can span multiple servers and the total capacity of a single server may be split among multiple physical volumes, *v*FS can not afford to retrieve all the information needed for FSSTAT in real time. *v*FS nodes periodically queries all the servers by issuing FSSTAT requests to them and calculates the summary. A loose per-user quota, can also be implemented in this manner.

Procedure FSINFO retrieves nonvolatile volume state information and general information about the NFS protocol implementation such as the preferred and maximal size of a READ request and a WRITE request, whether hard links are supported, whether symbolic links are supported, and whether the update time of an object can be changed through SETATTR requests. *v*FS, of course, makes it clear that hard links are not supported. In addition, *v*FS declares a slightly smaller maximal WRITE size and a slightly smaller preferred WRITE size than those declared by servers such that after rewriting and reassembly, a WRITE request from clients does not grow beyond servers' limits. Similarly, *v*FS declares a slightly larger maximal READ size and a slightly larger preferred READ size such that clients do not get caught by surprise receiving a bigger READ reply from *v*FS.

Procedure PATHCONF retrieves the pathconf information for a file or directory such as the maximal number of hard links to it, the maximal path component length and whether file names are case sensitive. *v*FS presents the most restrictive pathconf from

[6]Technically, NULL is not associated with any specific object either, but we'll ignore it for the purpose of this discussion.

all the servers, especially the limit on the maximal path name length. Fortunately, all existing commercial or open source NFS servers have similar pathconf information.

3.7 Authentication Virtualization

The RPC protocol beneath the NFS protocol includes a slot for authentication parameters in every call, the content of which is determined by the type of authentication scheme used by the server and the client. The most widely used authentication scheme is called AUTH_UNIX, which requires each call to include a series of UNIX-style credentials: a user ID, a group ID and groups the user belongs to. NFS servers check permissions after extracting the credentials from the RPC authentication fields in each remote request. Since existing manual storage management approaches today already allow one client to access different servers, these servers are most likely already sharing the same user and group ID space. But for servers that do have different views about which ID represents which user, their access right checking is completely broken. *v*FS's remedy is again to store a table on RPDS that maps global user/group IDs to server specific user/group IDs and this table is constantly propagated to all the *v*FS nodes to avoid having to go through RPDS for lookup operations.

3.8 Lock Virtualization

NFS uses the Network Lock Manager (NLM) protocol to support file locking and the Network Status Monitor (NSM) protocol to notify clients and servers about their lock states in the presence of server crashes and reboots. NLM follows the same simple one-server architecture as NFS, and NLM servers are often co-located with NFS servers. By their very nature, locking protocols are stateful: they maintain

the information about which clients currently own a lock on which file and which other clients are waiting for a lock. This poses a problem for the lock protocol virtualization as there does not exist a mechanism to query the internal states of lock managers. Therefore, it becomes almost impossible to migrate a file from one file server to another file server and reinstate the file's lock state there. Furthermore, NLM and NSM are used very rarely, there is no need to leverage the file servers for load balancing purposes. As a result, *v*FS virtualizes NFS lock service through a different approach than the virtualization of ID space and name space; it completely bypasses the server implementation of NLM and NSM and provides such functionality from scratch through RPDS.

3.9 Replicated Persistent Directory Service

Replicated Persistent Directory Service provides both a reliable store with limited capacity and processing that needs to be serialized. RPDS is replicated among an odd number of machines, typically 3 or 5, to achieve higher reliability and higher availability; it survives the non-Byzantine failure of $\lfloor n/2 \rfloor$ replicas where n is the total number of replicas (nodes).

3.9.1 Paxos Protocol

Like most other existing fault tolerant systems [57][11][41], at the core of RPDS is the well known Paxos protocol. The Paxos protocol was first proposed by Leslie Laport [36][37] as a more efficient alternative to voting [23][25] for achieving *asynchronous distributed consensus*. Distributed consensus refers to the agreement among

machines[7] within a particular distributed group, and asynchrony refers to the fact that these machines can progress at vastly different speeds and message delivery latency is unbounded. Another protocol equivalent to the Paxos protocol was independently invented by Oki and Liskov under the name of viewstamped replication [48]. The Paxos protocol guarantees the safety property: no value is chosen unless it is proposed by a member; and no two distinct values are both chosen. The Paxos protocol however relies on the existence of clocks on the members with bounded clock skews to overcome the impossibility result of the liveness property [21]: the eventual termination of the protocol.

The viewstamped replication version of the protocol is less complex and easier to understand, and therefore it is chosen to be briefly described here. Viewstamped replication protocol is based on the primary copy technique. One replica is designated as the primary and the others as backups. Only the primary can take client requests and process them; it notifies the backups of what it has done. The primary also assigns a monotonically increasing request ID to each request and ensures that requests are sent to backups in order. A view is a subset of the replicas capable of communicating with each other; each view is uniquely identified by a view ID. All the replicas constantly send "I'm alive" messages to each other to detect "view changes" which may be caused by replica crashes, replica reboots, network partitions and network reparations. Once a replica detects a view change, it initiates a view change protocol that aims to elect a new primary which has to be accepted by a majority of the replicas to ensure there is at most one primary in the same group. The combination of a request ID and a view ID is called a viewstamp and viewstamps are used to detect lost information during view changes. However, if a request has

[7] Often called processes.

been processed by a majority of replicas, the state changes are guaranteed to survive into subsequent views.

RPDS clients, such as *v*FS nodes, have the knowledge of the membership of the RPDS group, but do not get notifications when primaries get elected; they discover such information by sending requests to any random member. A RPDS member R_1 that is currently not the primary rejects all requests and informs clients about the current primary R_2 it knows about, and clients will try to send requests to R_2. If when requests arrive at R_2, the primary changes again, R_2 informs clients about the more up-to-date primary information. The process may go on and on until the primary stabilizes although it usually takes only one round. If clients talk to dead R_2, the requests will time out and clients will try another random member. A simple but effective optimization technique to the primary discovery protocol is caching the location of the primary on clients since the group primary changes very infrequently.

3.9.2 Read Optimization

In the Paxos protocol, read operations are also replicated to a majority of the group members otherwise stale information may be returned since other replicas may have formed a different view and elected a different primary. This increases the observable latency of any NFS client request that involves access of RPDS such as a request with a vhandle that can not be resolved by a *v*FS node. RPDS introduces the primary status lease into the Paxos protocol; the lease assumes the existence of clocks on all the replicas with clock skews smaller than 1 second. The assumption is a fairly weak one given that Network Time Protocol[47] (NTP) implementations based on Marzullo's thesis work [42] is widely available already; these implementations achieve accuracies within 100 milliseconds over the public Internet and 200 microseconds or

better in local area networks. Each time a primary p is elected by at least a majority of the replicas, they promise not to participate in any view change protocol within $t + 1$ seconds (t is usually 2 or 3) even after a reboot, unless the view change is initiated by p itself. With such a promise from a majority of the replicas, p is sure that no new primary can be elected within t seconds and therefore, p may respond to read operations without worrying about returning stale information before the lease expires. The primary p also renews its lease $t/2$ seconds after it grabs the primary status to avoid unnecessary primary switching. The primary status lease improves read operation latencies at the price of availability: if the primary replica crashes, RPDS will be unavailable for up to $t + 1$ seconds. The temporary unavailability problem does not pose a significant problem since NFS clients retransmit requests.

Since the primary RPDS replica handles all the read requests and coordinates write requests, it apparently needs more computation power than backup replicas. It is therefore beneficial to have one machine that is more powerful in RPDS to function as the primary replica. This machine can not be manually assigned the primary status, but RPDS can be configured such that when multiple replicas are competing for the primary status, preference is given to this special candidate.

3.10 Summary

NFS protocol virtualization enables *v*FS to aggregate the resources of file servers together and make the servers appear like a single one to clients. Clients can interact with this virtual server only by sending NFS requests and observing its ID space, name space and file contents. No special handling is required for file content, but it is crucial to ensure that clients observe one coherent ID space and name space.

The basic technique for ID space virtualization is ID rewriting based on either a local table managed by each *v*FS node or a global table; in the latter case, the global table is managed by a light weight replicated persistent directory service (RPDS) and aggressively cached by each *v*FS node. Name space virtualization leverages file servers for all the operations within the same subtree and relies on RPDS to handle operations that cross subtree boundaries. RPDS is built upon a well known asynchronous distributed protocol— Paxos protocol—with a couple of optimization techniques to improve the latency of read operations. Besides ID space virtualization and name space virtualization, *v*FS also manages the uniform credential space and provides its own lock service.

Chapter 4

Transparent Migration

Protocol virtualization breaks the bindings between clients and file servers, and aggregates the storage, computation, and networking capacity of all servers together. In addition to protocol virtualization, *v*FS provides the glue for independent file servers to cooperate with each other by adding the capability to transparently move objects among file servers without clients noticing. Such capability is referred to as *transparent migration*. Transparent migration breaks the bindings between data and servers and along with protocol virtualization, enables resource sharing. With transparent migration, *v*FS can treat the NFS file servers as completely interchangeable with each other.

Transparent migration operations should not be visible to *v*FS clients and they need to satisfy two important consistency requirements: 1) during and after migration, clients should not be able to observe any changes in file system tree structure, file data or file metadata; and 2) changes made by clients during migration should not be affected either. Ideally, clients should also be shielded from performance degradation during migration [39]. In fact, the design of migration focuses more on maintaining

system simplicity and ensuring correctness than on migration speed.

Transparent migration is usually initiated automatically by a Migration Policy Manager. Periodically, the migration policy manager collects file server performance statistics and capacity information from *v*FS nods. If the performance and/or capacity statistics match any of the load balancing, capacity balancing or data life cycle management rules predefined by administrators, the migration policy manager initiates transparent migration. Administrators may also manually initiate transparent migration mostly for "one-shot" events such as online maintenance and file server addition or obsolescence. Both automatic and manual migration go through the same process and *v*FS guarantees that the system does not start a new migration session before the previous one finishes, a design tradeoff made to contain the system complexity.

The migration process in *v*FS is orchestrated by a Migration Coordinator which implements the complex migration logic; individual *v*FS nodes that actually perform the data copying only implement relatively simple migration primitives. The coordinator invokes these primitives on the *v*FS nodes and oversees the entire migration process. This separation of migration primitives from migration algorithms not only keeps the design of the critical *v*FS nodes simple, but also allows more sophisticated migration algorithms to be developed in the future and different migration algorithms to coexist with each other. The migration coordinator also maintains a migration log that keeps track of each step of the migration process to assist crash recovery. Finally, the coordinator can be instructed to control the migration speed in order to reduce the performance impact on normal client operations that occur during migration.

The rest of this chapter is organized as follows. Section 4.1 presents the concept of choke point that is critical to ensuring migration consistency, and explains the

various *v*FS migration primitives and the actual migration algorithms. Section 4.2 briefly studies how transparent migration can be used to support file server load balancing, capacity balancing and tiered storage management. Finally Section 4.3 summarizes this chapter. The rest of this chapter may also use migration in place of transparent migration for brevity.

4.1 Migration

The basic migration unit in *v*FS is a subtree. One naive solution to migration is to first "lock" the whole subtree by temporarily denying client access, and then copying the whole subtree to a different server before re-enabling client access. This solution works well for very small subtrees, but employing it for large subtree migration introduces intolerable service disruption which a good design should avoid at all cost. We observe that in NFS protocol a) all data operations involve only one object and b) all metadata operations involve at most two objects under the same directory[1]. The *v*FS migration design recognizes this opportunity and selectively "locks" the subtree one object at a time, minimizing file service disruption. In addition, *v*FS even allows most common operations from clients on an "locked" object by temporarily replicating these requests, minimizing the impact migration may have on normal client operations.

Given a subtree, the migration coordinator traverses it in pre-order and while visiting each object, it coordinates among all the *v*FS nodes the migration of this object from one file server to another through a three phase process: 1) mirroring the object at the destination file server; 2) switching client access over to the mirror

[1] *v*FS only allow rename operation within the same directory.

atomically; and 3) deleting the object from the source file server. *v*FS satisfies the transparency requirements throughout all three phases.

4.1.1 Choke Point Mechanism

The mirroring phase creates a replica of the original object at the destination server. An important problem during this phase is how to allow client operations, including ones that directly modify the object, to continue safely alongside the ongoing migration. We note that the mirroring itself could take a significant amount of time for big objects and simply blocking client operations even for a single object during the entire mirroring process would result in downtime and the violation of the transparency requirement. *v*FS employs a "choke point" technique to provide transparency during the migration operation. For each object being migrated, all *v*FS nodes are instructed by the migration coordinator to forward requests related to the object to a certain designated choke point node so that it can manage all operations (client-generated or those arising from migration) on the object. Any *v*FS node can serve as a choke point for migrating any object and it is the responsibility of the migration coordinator to decide which *v*FS nodes should be the choke point for which objects. In practice, such a decision is made according to load balancing or security policies.

The choke point is able to control all the operations on the object to make consistent mirrors. In the switch-over phase, the choke point can switch from the old object to the new object instantly and the change is visible to all *v*FS nodes (and by all clients indirectly) simultaneously. After the switch-over phase, *v*FS nodes stop forwarding requests to the choke point; instead, the new requests are forwarded to the new destination server as part of the normal virtualization process. The last phase of migrating an object deletes the copy from the original file server, since it is no

longer consistent with the object after the switch-over phase. Note that this phase can be safely delayed if the storage capacity occupied by the object is not immediately needed.

The next section describes the details of how a choke point operates when migrating a file, when migrating a full directory subtree, and especially how a choke point interacts with the migration coordinator which orchestrates the entire migration process.

4.1.2 Migrating a File

The *v*FS node chosen as the choke point for migrating a file reads successive portions of the file from the source server and writes them to the destination file, created by the migration coordinator on the destination server. During the copying process, the choke point receives all client requests for the object. It forwards all the client read requests to the source file server and all the client write requests to both the source and the destination server. When all the data has been copied, both the source copy and the destination copy have identical content. At this stage, the choke point first temporarily blocks client access to the file and then does two things: 1) mirroring the file attributes including access bits and various timestamps, and 2) updating both its local v-table and the global v-table so that the file's vhandle is mapped to the phandle of the new copy. Once this is all completed, the choke point returns the new v-table entry to the migration coordinator. Upon receiving the new v-table entry, the coordinator propagates it to all other *v*FS nodes and instructs them to stop forwarding all requests regarding the file to the choke point and instead start processing them locally. When all the *v*FS nodes are updated, the coordinator informs the choke point that it is off duty. While an object is being mirrored, besides read

and write operations and metadata update operations which are treated the same as writes, clients may also issue object RENAME and DELETE requests. The mirroring process is not affected by RENAME requests as will be made clear in Section 4.1.3, but upon the success of a DELETE operation, the choke point immediately stops the mirroring process, informs the migration coordinator and deletes the new copy lazily. Procedure MIGRATEFILE in Algorithm 1 describes the main flow of the migration coordinator when orchestrating the migration of a single file. The algorithm takes four arguments as input: the file f to be migrated, the destination directory p under which to create the new file, the server S_f that manages f and the server S_p that manages p. The algorithm employs the notation FUNCTION$[M]$(`arg`) to represent a synchronous or asynchronous RPC call to function FUNCTION on machine M with arguments `arg`. Note in Algorithm 1 that the coordinator sends instructions to the *v*FS nodes in parallel to reduce latency.

Procedure MIRRORFILE in Algrithm 2 presents the pseudo code of the main path for a *v*FS node when designated as the choke point for a file migration. The procedure mirrors file f managed by server S_f to file f' managed by server $S_{f'}$. Line 3 to Line 12 describe the phase where the choke point asynchronously duplicates f to f' while forwarding read type of requests to S_f and forking write type of requests to both S_f and $S_{f'}$. At line 14, the algorithm blocks clients and other *v*FS nodes from accessing f by delaying processing their requests and at line 18 those requests are processed in order. Since it only takes a few LAN round trips to mirror the file attributes and updating the corresponding v-table entry in RPDS, the impact to clients is barely noticeable.

Since *v*FS nodes need to make arbitrary changes on the file servers to make migration possible, they use the ROOT user ID to override all access checking on the file

Algorithm 1 Migration coordinator's algorithm for migrating a file

```
1:  procedure MigrateFile(S_f, f, S_p, p)          ▷ At the migration coordinator.
2:      f' ← CreateEmptyFile[S_p](p, f.name)
3:      V_c ← SelectChokePoint(f)

4:      for all V_i : vFS nodes do                  ▷ Notify nodes in parallel
5:          StartRequestForwarding[V_i](f, V_c)
6:      end for
7:      WaitForResponses                            ▷ Synchronization

8:      Succ ← StartMirrorFile[V_c](f, f')          ▷ f may be deleted
9:      if Succ = true then
10:         for all V_i : vFS nodes do              ▷ Notify nodes in parallel
11:             StopRequestForwarding[V_i](f, f → (S_p, f'))
12:         end for
13:     else                                        ▷ File deleted or failure
14:         for all V_i : vFS nodes do              ▷ Send requests in parallel
15:             StopRequestForwarding[V_i](f)
16:         end for
17:     end if
18:     WaitForResponses                            ▷ Synchronization

19:     ChokePointOffDuty[V_c](f)                   ▷ Choke point back to normal
20: end procedure
```

servers. The file servers are always configured to allow these accesses by turning off the user ID rewriting options (e.g., ROOT_SQUASH). These options are implemented by *v*FS for the virtual volumes, so there is no loss of functionality or additional security risks as long as the file servers are placed in a private network directly accessible only by the *v*FS nodes as gateways.

Algorithm 2 A file migration choke point *v*FS node's algorithm

```
1: procedure MirrorFile(f, S_f, f', S_f')
2:     Read[S_f](f)                                   ▷ Asynchronous read
3:     while not EOF(f) do
4:         if data from S_f then
5:             Write[S_f'](f', data)                  ▷ Mirror file data
6:             Read[S_f](f)                           ▷ Read more data asynchronously
7:         else if Remove request then
8:             Remove[S_f'](f')                       ▷ Delete mirror
9:             Remove[S_f](f)                         ▷ Delete original file
10:            Return
11:        else if read type requests then
12:            Forward[S_f](f, req)                   ▷ Read requests forwarded to source file
13:        else[Write type requests]
14:            Forward[S_f(f, req)                    ▷ Write type requests forked to both files
15:            Forward[S_f'](f', req)
16:        end if
17:    end while
18:    Put all requests related to f in a Lock queue  ▷ Lock
19:    attr ← GetAttr[S_f](f)
20:    SetAttr[S_f'](f', attr)
21:    UpdateVTable[RPDS](f → (S_f', f'))
22:    Dequeue the requests and resume normal processing   ▷ Unlock
23: end procedure
```

4.1.3 Migrating a Directory

The migration of a directory is a recursive process consisting of the following steps:

1. The migration coordinator first creates a mirror directory on the destination

server, then creates a jdirectory consisting of both the source directory and the destination (mirror) directory. The static object creation policy is applied to the jdirectory which creates all new objects under the mirror directory. Since only RPDS can guarantee the consistency of jdirectories, all *v*FS nodes are therefore instructed to forward their operations on the jdirectory to RPDS.

2. The coordinator retrieves a list of the objects under the source directory (via READDIR or READDIRPLUS operations).

3. For each file object in the list, the coordinator starts the file migration process as described in Section 4.1.2. For each directory object in the list, the coordinator starts the same process as described in this section, recursively.

4. Once all the objects from the source directory are migrated to the mirror directory, RPDS temporarily blocks client access to the directory and performs the switch-over process which involves a) mirroring the directory attributes; b) modifying the v-table such that the vhandle of the source directory is mapped to the new mirror directory; and c) removing the jdirectory virtualization, making the mirror directory directly manageable from all *v*FS nodes.

5. The coordinator propagates the new v-table entry to all the *v*FS nodes and instructs them to stop the forwarding of requests regarding the directory.

The static object creation policy that directs all new objects to mirror directories is very critical to the predictable completion time of migration and it avoids the need of migrating objects created after the start of migration; otherwise clients may create new objects at a faster rate than the migration speed, preventing migration from completing. During the recursive process of migrating a directory, RPDS is used to handle requests regarding multiple descendent directories, as many as the depth

of the directory. The slightly more formal description of the directory migration recipe for the coordinator is described in Algorithm 3 where d is the directory to be migrated, S_d is the server that manages d, p is the parent directory under which d's mirror will be created, and S_p is the server that manages p. RPDS's algorithm is not much different from its normal behavior. The migration of a subtree is essentially the same as migrating a directory except that the top level directory in the subtree is turned into a permanent rather than temporary junction directory. The root of a virtual volume is a normal non-root directory managed by a server and therefore has a parent on the server. Should various maintenance needs arise, it can also be migrated like any other directories.

Figure 4.1–4.4 depicts the process of a subtree migration. There are three trees in all four figures: the first tree is the file system on server S_1, the second tree is the file system on server S_2 and the third tree is clients' view of the virtual volume. The root of the virtual volume is managed by S_1 as S_1`:/0/root`. Figure 4.1 shows the initial state of the system in which S_2 doesn't host any objects, and now suppose subtree c, including object c, d, $e/$, f, g and h, are chosen to be migrated to S_2 under directory S_2`:/1`. Figure 4.2 shows that first a directory with a unique name[2] is created under directory S_2`:/1`, referred to as `uniq`, such that there is no name clashes even if other subtrees are later on scheduled to be migrated to S_2`:/1` as well. S_1`:/0/root` and S_2`:/1/uniq` are then turned into a jdirectory which uses the default object creation policy: new objects are always created under S_1`:/0/root`. Figure 4.3 shows the state of the system after directory S_1`:/0/root/c` has been partially migrated to S_2 as S_2`:/1/uniq/c` and object S_1`:/0/root/c/d` has been migrated to S_2 as S$_2$`:/1/uniq/c/d`. S_1`:/0/root/c` and S_2`:/1/uniq/c` are then turned

[2]The hexidecimal representation of (S_1, S_1`:/0/root`'s phandle) is a good candidate.

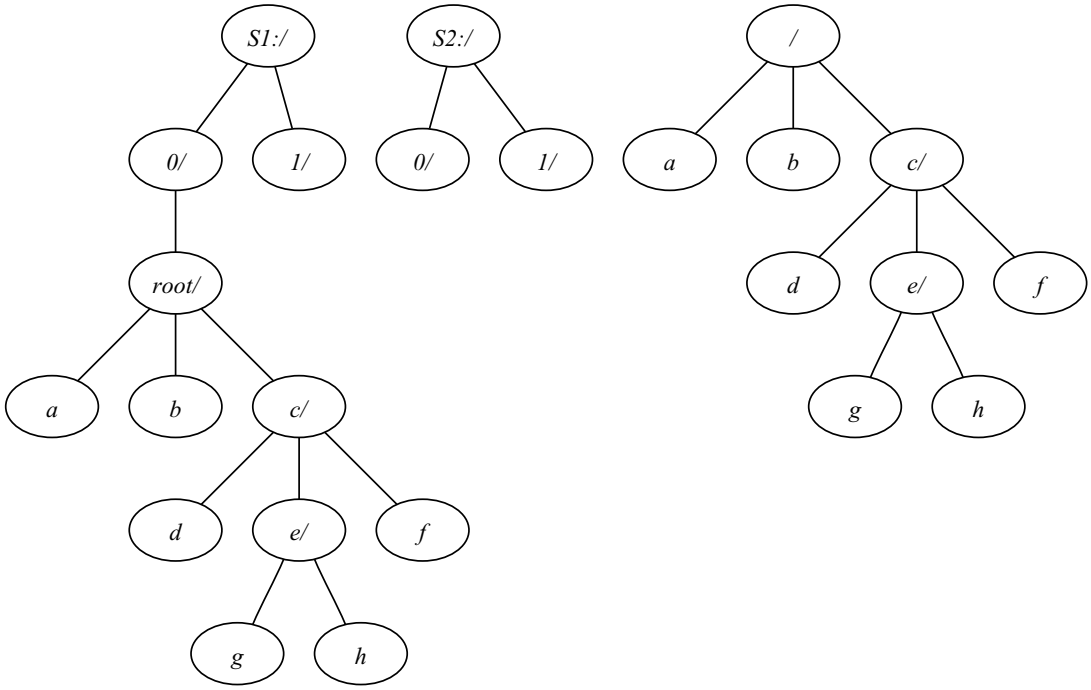

Figure 4.1: Before the migration of subtree c that includes object c, d, $e/$, f, g and h. The first tree in the figure is the file system on server S_1, the second tree is the file system on server S_2 and the third tree is clients' view of the virtual volume.

into another jdirectory to provide unchanged virtual volume view to clients. This jdirectory, however uses a different static object creation policy such that a new object `i`, `/c/i` in clients' view, is always created under S_2`:/1/uniq/c`. The last figure, Figure 4.4, shows the state of the system at the end of migration. The whole subtree c has been successfully migrated from S_1 to S_2, the temporary jdirectory consisting of S_1:/0/root/c and S_2:/1/uniq/c and the temporary jdirectory consisting of S_1:/o/root/c/e and S_2:/1/uniq/c/e (not shown in the picture) is gone, but the jdirectory consisting of S_1`:/0/root` and S_2`:/1/uniq` is made permanent. All these figures also demonstrate that during the whole migration process, clients always see the same consistent virtual volume.

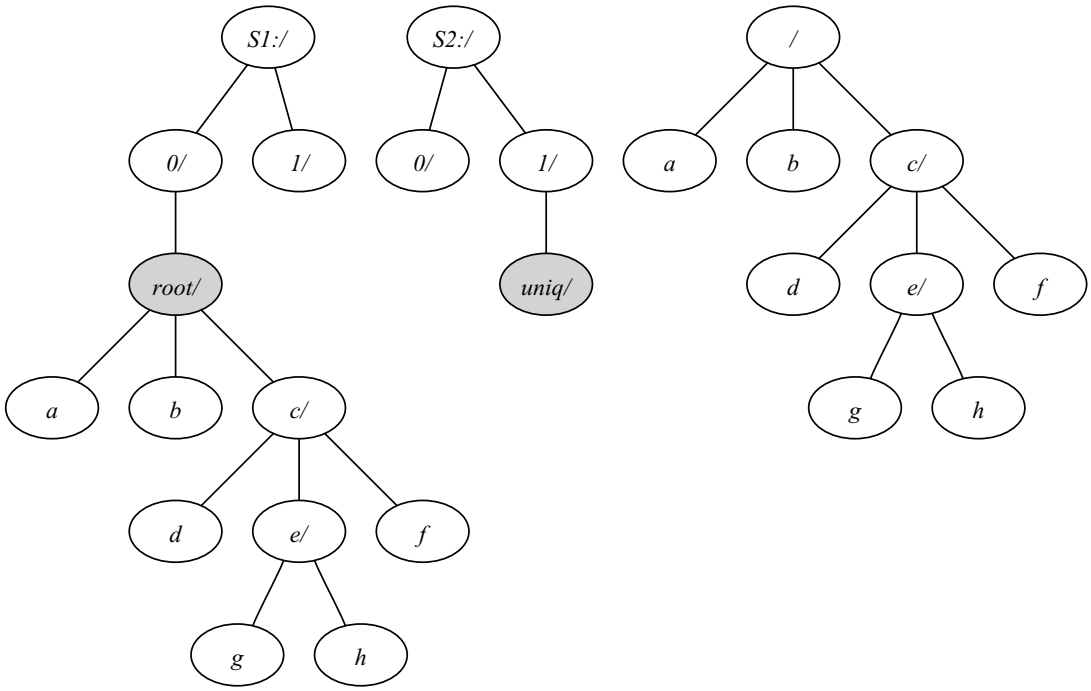

Figure 4.2: Subtree c is chosen to be migrated to S_2`:/1`. First a directory with a unique name `uniq` is created under S_2`:/1` and then S_1`:/0/root` and S_2`:/1/uniq` are turned into a jdirectory. This jdirectory uses the default static object creation policy: new objects are always created under S_1`:/0/root`.

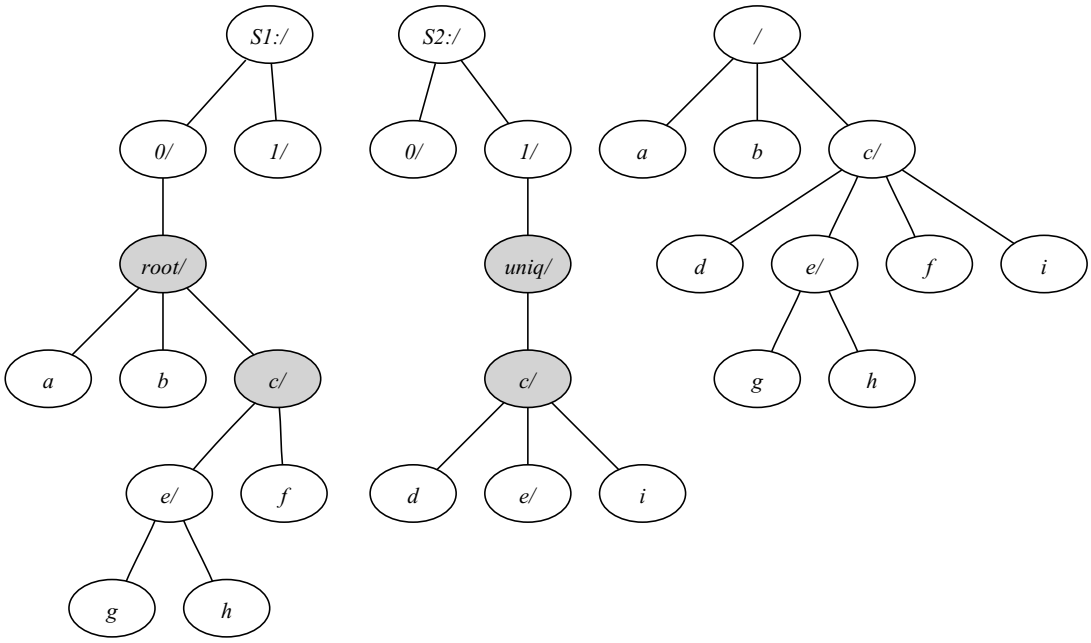

Figure 4.3: Directory S_1`:/0/root/c` has been partially migrated to S_2 as S_2`:/1/uniq/c` and object `:/0/root/c/d` has been migrated to S_2 as S_2`:/1/uniq/c/d`. S_1`:/0/root/c` and S_2`:/1/uniq/c` are turned into a jdirectory to provide unchanged virtual volume view to clients. This jdirectory's static object creation policy directs all new objects to S_2`:/1/uniq/c`.

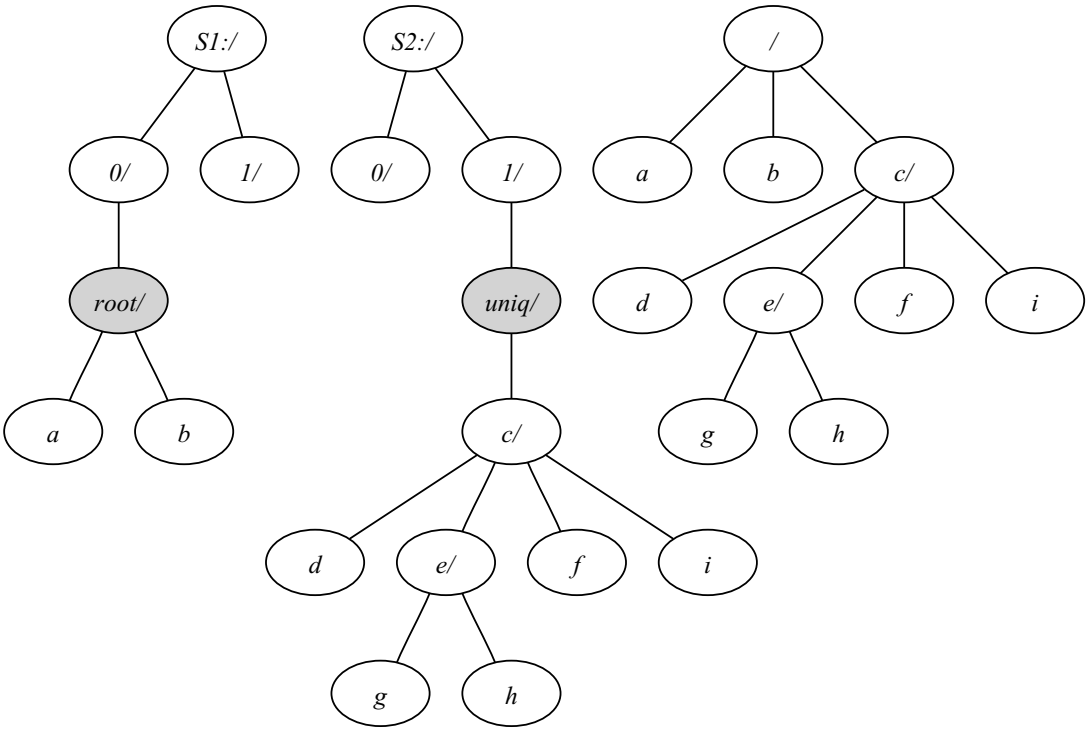

Figure 4.4: The end of migration. The temporary jdirectory consisting of S_1:/0/root/c and S_2:/1/uniq/c is gone, but the jdirectory consisting of S_1`:/0/root` and S_2`:/1/uniq` is made permanent. During the entire migration process, clients' view of the virtual volume remains unchanged.

Algorithm 3 Migration coordinator's algorithm for migrating a directory

```
 1: procedure MigrateDirectory(S_d, d, S_p, p)
 2:     d' = Mkdir[S_p](p, d.name)                    ▷ Create an empty directory
 3:     Attach[RPDS](d, d', static policy)            ▷ All new objects created under d'

 4:     for all V_i : vFS nodes do
 5:         StartForwarding[V_i](d, RPDS)             ▷ Send in parallel
 6:     end for
 7:     WaitForResponses

 8:     L = Readdirplus[S_d](d)                       ▷ Get the list of objects under d
 9:     for all o : L do                              ▷ For each object under d
10:         if o is a directory then
11:             call MigrateDirectory(S_d, o, S_p, d')        ▷ Recursion
12:         else
13:             call MigrateFile(S_d, o, S_p, d')     ▷ described in section 4.1.2
14:         end if
15:     end for

16:     for all V_i : vFS nodes do
17:         StopForwarding[V_i](d, d→(S_d', d'))      ▷ Send in parallel
18:     end for
19:     WaitForResponses
20: end procedure
```

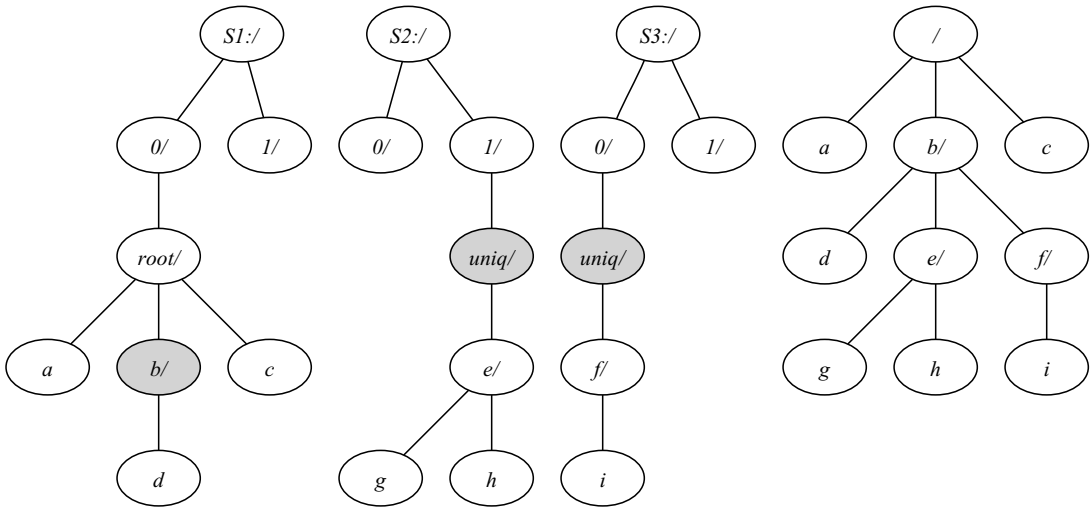

Figure 4.5: The subtree distribution of a sample virtual volume. The three trees on the left are the file systems on server S_1, S_2 and S_3 respectively and the tree on the right is the virtual volume view. Directory S_1:/0/root/b, S_2:/1/uniq and S_3`:/0/uniq` are constituents of jdirectory `/b`.

4.1.4 Virtual Volume Fragmentation and Subtree Merger

Since subtrees can only be "carved" out of existing subtrees, excessive or careless use of migration may get virtual volumes fragmented; that is each virtual volume is splitted into a large number of relatively small subtrees. When a virtual volume becomes fragmented, RPDS needs to manage more jdirectories and therefore may throttle the throughput of the entire system. Furthermore, migration on the fragmented virtual volume also becomes less effective since each subtree is too small. *v*FS allows adjacent subtrees in the virtual volume name space, whether physically hosted on the same server or not, to be merged, reducing the total number of subtrees and jdirectories, and enabling the restructuring of subtrees. Subtree mergers are also a very effective measure that is usually taken when shrinking the server set.

Figure 4.5 shows the subtree distribution of a sample virtual volume. The three trees on the left are file systems on three servers and the subtree on the right is the uni-

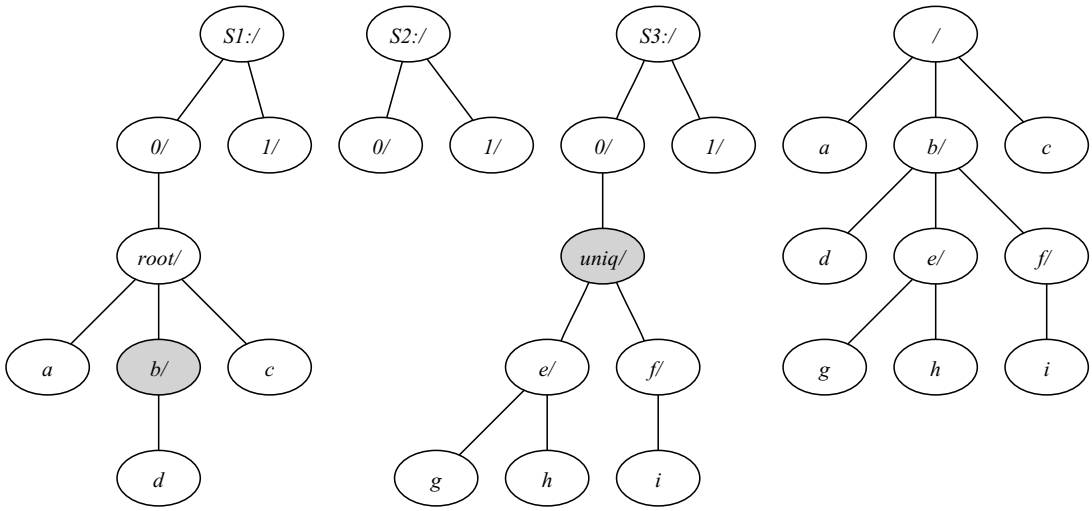

Figure 4.6: After subtree S_2`:/1/uniq` is migrated to S_3 and merged with S_3`:/0/uniq`, there is only two constituent directories in jdirectory `/b`.

fied virtual volume view. In this virtual volume, directory S_1:/0/root/b, S_2:/1/uniq and S_3`:/0/uniq` are constituent directories of jdirectory `/b`. All these three subtrees can be merged with one another. For example, subtree S_2`:/1/uniq` may be migrated to S_3 and merged with S_3`:/0/uniq` or vice versa. The result of such a merger is shown in Figure 4.6. Note that now jdirectory `/b` only has two constituent directories. Once all three subtrees are merged together and hence stored on and managed by the same server, `/b` becomes a normal directory. *v*FS nodes no longer need to forward operation regarding `/b` to RPDS and all the overhead associated with maintaining `/b` as a jdirectory is eliminated.

4.1.5 Crash Recovery During Migration

The migration is a relatively sophisticated process. If any component fails during the process, *v*FS is required to guarantee that the integrity of related virtual volumes is not affected. The integrity requirement for normal file system crash recovery is

that file systems be brought back to a consistent state while losing as little data as possible. Besides integrity, as a shared infrastructure, *v*FS migration is also required to guarantee that the performance of the virtual volumes involved in the migration not be significantly impacted should any component fail for an extended period of time.

A *v*FS node stores the vhandles of the objects being migrated and the addresses of the choke points responsible for mirroring those objects. Such information is kept only in RAM until the current migration session finishes. Upon restart, possibly after a crash, a *v*FS node first queries RPDS to obtain the location of the migration coordinator if there is any, then retrieves the object vhandles and the choke point locations from the migration coordinator. For objects for which it is their choke point, the *v*FS node picks up the choke point responsibility and restarts the idempotent mirroring process; for other objects being migrated the *v*FS node starts forwarding related requests to their choke points. This design maintains *v*FS nodes' stateless nature and keeps *v*FS nodes simple.

Most of the migration related states are kept in a migration log persistently stored and maintained by the migration coordinator. During the course of migration, it logs every single step it takes to the migration log. For example, before designating a node as a choke point, it logs a "DESIGNATE_CHOKE_POINT" record containing the node address and the vhandle of the object; and before instructing all *v*FS nodes to forward their requests to the choke point, it logs a special "FORWARD_REQUEST" record. Once a record is logged, the migration coordinator blocks on the next step operation with retries until it succeeds. While a *v*FS node is down, be it a choke point or a non-choke point, the migration process will not proceed until the failed machine comes back online and recovers from crash or is explicitly removed from the

*v*FS node set by administrators. If the choke point is removed from the *v*FS node set, the coordinator will choose a different *v*FS node as the choke point. If the coordinator itself crashes, it reads the migration log and then restarts the migration from where it left off before the crash. In the extremely rare case where the coordinator crashes and can not even recover from the migration logs which might have been corrupted due to disk failures, the coordinator declares current migration session complete and leaves behind some temporary jdirectories[3] and one file still being handled by a choke point. Note that in this case, the virtual volume is still consistent and there is no data loss. A new coordinator may register with RPDS any time. It first queries all the *v*FS nodes and determines the file being migrated if any, then instructs all the *v*FS nodes to stop forwarding requests related to the file. Administrators can explicitly start a new session to continue the unfinished subtree migration and remove the temporary jdirectories.

4.2 Automated Migration Case Studies

Migration can be routinely initiated by a migration policy manager according to rules predefined by administrators. In this section, we will study how different rules can be created to deal with three major server management cases: file server load balancing, file server capacity management, and tiered storage management.

4.2.1 File Server Load Balancing

Enterprises expect maximum performance and maximum efficiency from storage infrastructure, driven by both customer needs and competition. Transparent migration

[3]The maximal number of jdirectories left behind is the depth of the subtree being migrated.

provides the basic mechanism to reconfigure storage servers and eliminate performance bottlenecks without interrupting data access. The automated file server load management consists of three parts: monitoring and collecting server performance statistics; identifying overloaded subtrees; and initiate migration if necessary.

The duty of server performance statistics collecting is assumed by both the vFS nodes and the migration policy manager. Each vFS node is instrumented to passively measure the number of NFS operations handled per second by each server, denoted by r_i and the average latency of responses from each server, denoted by l_i. Since different vFS nodes may communicate with the same server, these measurement results are then collected and summarized by the migration policy manager to derive a better estimate of load $\overline{l_i}$ and average response latency $\overline{r_i}$ for each server. These two numbers can be used independently or combined as the final server load metric L_i. A relatively simple model may define the load as $L_i = \overline{l_i} \cdot \overline{r_i}^2$. L_i is defined as $\overline{l_i} \cdot \overline{r_i}^2$ rather than $\overline{l_i} \cdot \overline{r_i}$ to compensate for the fact that when a system is overloaded, the average latency goes up but the number of operations it can handle goes down. In a typical enterprise environment, file servers with very different throughput and latency characteristics abound and therefore administrators should be allowed to specify a weight factor w_i for each server. The adjusted server load $L_i' = L_i \cdot w_i = \overline{l_i} \cdot \overline{r_i}^2 \cdot w_i$ is used as the real load metric. Let $L_{max}^{'}$ be the load on the most loaded server and $L_{min}^{'}$ be the load on the least loaded server, and $I_l = L'_{max}/L'_{min}$ be the load imbalance ratio, the migration policy manager may decide that significant load imbalances exists when $I_l > \delta_l (\delta_l > 1)$, where δ_l is an administrator defined threshold, and the situation continues for an extended period of time. The selection of δ_l should be fairly conservative so as to provide a safety net around inaccurate estimates of w_i and temporary load spikes; to avoid load oscillation where subtrees are moved around without really smoothing out

load imbalance; and to avoid introducing too many jdirectories into the system.

Besides what is described above, any existing or future approaches to file server load modeling should all be applicable to identifying load imbalance. Once the most loaded server, referred to as S, and the least loaded server, referred to as s, are identified, the next step is to identify exactly which subtree should be migrated from the S to s. The policy manager counts the number of NFS operations on each subtree recursively by collecting the per object number maintained by each *v*FS node and a subtree is chosen such that its migration can balance out the load difference between S and s the fastest.

4.2.2 File Server Capacity Management

Simply buying and owning more storage does not solve all capacity problems; efficient capacity management requires that storage administrators find and get value from existing under-utilized file servers before buying more of them. *v*FS manages the server set as a resource pool and hence has good visibility into the distribution of all stored data. Transparent migration adds the ability to move data without interrupting data access users desire. Similar to load management, capacity management requires the collecting of server storage utilization information, the identification of suitable subtrees and the initiation of migration. Residual capacity C_i for each server can be collected simply by issuing FSSTAT requests to that server periodically; in fact, *v*FS nodes are already doing so to virtualize FSSTAT requests from clients, as described in Section 3.6. Also similar to load management, the policy manager periodically collects the storage each file, each directory and each subtree consumes recursively from servers. The policy manager uses C_i to determine the server S with the most residual capacity $C_{max} = max\{C_i\}$ and the server s with the least residual capac-

ity $C_{min} = min\{C_i\}$. Once the capacity imbalance factor $I_c = C_{max}/C_{min}$ exceeds $\delta_c(\delta_c > 1)$, another administrator specified threshold, capacity re-balancing is considered necessary. The size information of each subtree stored on s is used to choose the best candidate subtree the migration of which can balance the residual capacity between S and s the fastest. In a slightly more sophisticated model, the rate at which a particular subtree is growing over some chosen period can also be calculated and taken into the subtree selection process. This model may lead to more even capacity distribution with fewer number of subtree migrations.

4.2.3 Tiered Storage Management

The amount of data in enterprises is growing at unprecedented rates. In addition, data is being retained for longer periods of time due to business and regulatory constraints. While not all this data is critical to the day-to-day business operations, it must all be stored, protected and remain readily accessible should the need arise. In order to reduce the average cost per byte, enterprises are increasingly deploying more servers with tiered capacity and performance characteristics. Some servers are fibre channel based NAS, some can employ cost focused serial ATA (SATA) technology, yet the rest can be potentially slower content addressable storage [16, 55] or capacity optimized storage [17]. Tiered storage environment allows managers to assign data to the most appropriate platform as the value of data changes to reduce the average cost per byte. More valuable and more frequently accessed data can be assigned to high-performance hardware, while less critical and less frequently accessed data can be moved to less expensive storage with lower performance.

Transparent migration enables the relocation of data among different tiers without interrupting data access and the policy manager can automate the tiered storage

management with very little assistance from administrators; they only need to provide a few simple policies to help data classification based on file age. File age based policies allow administrators to classify files based on either the last-modified or last-accessed attributes. This is extremely useful for automatically migrating files which have not been accessed or modified for some extended period of time to a different tier of storage. A good example is the massive number of accounting reports each enterprise maintains; those haven't been accessed or modified for a few months should probably quickly be moved to archival storage. Of course, *v*FS's migration unit is a subtree, not an individual file, therefore what the policy manager keeps track of is the average last-modification time or average last-access time of subtrees rather than files. Once the data classification policies are determined, administrators specify where data in each category should be stored. Unlike load and capacity balancing management, tiered storage management usually does not need to happen immediately and therefore administrators can also decide exactly when the subtrees should be migrated so as to minimize impact on normal client traffic.

4.3 Summary

*v*FS not only aggregates the resources of file servers together, but also enables the transparent migration of subtrees from one server to another, making file servers completely interchangeable with each other. Migration is orchestrated by a migration coordinator which implements reliable logging for crash recovery and directs individual *v*FS nodes to accomplish the whole task. Subtree migration is carried out in a recursive fashion, file by file and directory by directory. Temporary jdirectories are used judiciously to ensure the timely completion of migration even though more

files or directories are still being generated. Different choke points take care of the actual migration of each file or object and handle all the requests related to the file or directory being migrated to ensure consistency. Transparent migration builds the foundation for a variety of automated management tasks: file server load management, capacity management and tired storage management. By migrating subtrees from busy servers to less busy ones, server load can be balanced; by migrating subtrees away from servers with less capacity, server capacity can be balanced; by migrating infrequently accessed subtrees to servers with lower performance and more economical hardware and software, the average storage cost per byte can be reduced.

Chapter 5

*v*FS Federation

The set of *v*FS nodes serving the same virtual volume are federated and can be used interchangeably for dynamic load balancing and failure handling purposes, eliminating the fixed bindings between clients and *v*FS nodes. The principle of federation appeared in all major distributed file systems, but *v*FS federation strives to preserve the traditional NFS model and access semantics. *v*FS federation provides two forms of dynamic load balancing: mount-time load balancing and online load balancing. The former allows different clients to be associated with different *v*FS nodes when a virtual volume is mounted and hence creates good initial load distribution among *v*FS nodes. The latter, when configured, can dynamically change the association between clients and their assigned *v*FS nodes if significant load imbalance among the *v*FS nodes is observed, without interrupting client access. Besides balancing load among themselves, a *v*FS node can also completely take over the responsibility of another transparently without causing client access disruption. *v*FS federation compliments both protocol virtualization which breaks the fixed bindings between clients and file servers and transparent migration which breaks the bindings between data and file

servers.

The next three sections in this chapter present mount-time load balancing, online load balancing and online failover respectively in detail, and the last section summarizes.

5.1 Mount-Time Load Balancing

A virtual volume in *v*FS is known to clients by a DNS [5] name. For example, the virtual volume for storing archival data in a company may be called `arch.company.com`. A client trying to access the volume first resolves `arch.company.com` to an IP address through a DNS server, then uses that IP address to mount the volume and retrieve the vhandle of its root. The client also sends all subsequent file system requests specific to the virtual volume to this IP address. All *v*FS nodes exporting the same virtual volume of interest are functionally equivalent to clients, *v*FS can therefore employ a smart DNS server to make just-in-time decisions as to which *v*FS node a particular client should be assigned to by returning the IP addresses of different nodes, providing the basic mechanism to balance the load on different *v*FS nodes at client mount time. Mount-time load balancing is very easy to understand, very easy to implement and especially attractive to desktops managed by automount, which automatically unmounts a virtual volume when a long enough volume access gap is observed and remounts the volume should it be accessed again.

A well known technique called Round-Robin DNS [32, 15] is widely used for web server load balancing by returning different IP addresses from a list in a round robin fashion. Content Delivery Networks (CDN) [69, 4] also use customized DNS servers to return the IP addresses of web severs that are geographically close to clients, so

as to improve client perceived latency and reduce wide area network traffic. Web server load balancing has three characteristics: (1) server assignment decisions are made on a per request basis; (2) each web request is short-lived and (3) the resource consumption of different requests has very small variance. The component in *v*FS that decides the *v*FS node assignment for clients is called *mount-time load balancer*, which is essentially a smart DNS server. Unlike other smart DNS servers described above, the mount-time load balancer's assignment of a client to a *v*FS node is permanent until online load balancing is triggered, and each client can place drastically different load onto *v*FS. Therefore, every assignment decision made by the mount-time load balancer has significant impact. It takes into account the expected traffic volume of the potential client and assigns it to the *v*FS node with least current real-time load. The expected traffic volume of the potential client, measured in the number of requests per second, can be inferred from historical data or be given explicitly as administrators' estimate in the case of a new client. The current real-time load on each *v*FS node is similarly measured in the average number of requests it receives per second over some relatively long period, and reported periodically to the mount-time load balancer.

So far clients have been assumed to always access a virtual volume by resolving its DNS name first, in reality they may access *v*FS nodes using their IP addresses directly. In order to ensure the effectiveness of mount-time load balancing, the Time-to-Live (TTL) of the name resolution results returned by the mount-time load balancer is set to only one or two seconds. Standard conforming clients cannot cache the results longer than specified by the TTL. *v*FS however also provides an ephemeral access control mechanism that ensures clients can only access the *v*FS nodes they are assigned to. Normally, *v*FS nodes reject mount requests from all clients. Before returning the

IP address of a *v*FS node to a client, the mount-time load balancer directs the *v*FS node to open the admission gate for the specific client for a period that is only two times as long as the TTL.

Like any other component, the mount-time load balancer may also fail, but it is a relatively simple component with very light load and thus can be made reliable very easily through replication. In addition, clients usually are configured with both a primary DNS server and a secondary DNS server which is resorted to when the primary fails. What each *v*FS node needs to do to take advantage of this reliability mechanism is to report its load to both mount-time load balancers so that both can respond to client name resolution requests independently.

5.2 Online Load Balancing

The mount-time load balancer allows the creation of good initial distribution of load among *v*FS nodes. However, since the NFS protocol binds the clients to a specific file server after a volume is mounted, additional techniques are necessary to provide the flexibility of balancing load among the *v*FS nodes and enable moving responsibility of handling client requests from one *v*FS node to another in case significant load imbalance takes place. This problem has two basic challenges: how to redirect clients to send requests to a different node without violating the NFS protocol; and how to shift only some of the clients. *v*FS solves this problem by requiring each *v*FS node to own multiple identities by binding to multiple IP addresses[1]. The first two columns in Table 5.1 describe a sample *v*FS federation configuration in which each *v*FS node binds to three IP addresses. For example, V_1 binds to 192.168.0.1, 192.168.0.2 and

[1] All commonly used operating systems now support the binding of multiple IP addresses to the same network interface.

*v*FS node	IP Addresses	Load (#req/sec)	Total load (#req/sec)
V_1	192.168.0.1	300	1200
	192.168.0.2	300	
	192.168.0.3	600	
V_2	192.168.0.4	100	600
	192.168.0.5	200	
	192.168.0.6	300	

Table 5.1: In a sample *v*FS federation, each *v*FS node owns three identifies by binding to three IP addresses. The first column is the name of *v*FS nodes, the second column is the IP addresses each *v*FS node binds to, the third column is the number of requests per second each IP address receives and the last column is the total number of requests per second each *v*FS node receives.

192.168.0.3. When a *v*FS node reports its load to the mount-time load balancer, it reports the load on a per IP basis. The mount-time load balancer can thus assign a client to the least loaded IP bound to the least loaded *v*FS node. The third and fourth columns in Table 5.1 describe the load on each IP and the aggregated load on each *v*FS node. Since V_2 has less load (1200 req/sec) than V_1 (600 req/sec) and 192.168.0.4 has the least load on V_2, the next client will be assigned to 192.168.0.4.

Besides the mount-time load balancer, an *online balancer* (often co-located with the mount-time load balancer) also receives the load report from each *v*FS node. The online load balancer periodically checks the ratio of the aggregated load on the most loaded node N to that on the least loaded node n. Once the ratio exceeds a predefined threshold, online load balancing is triggered and the online load balancer picks an IP address on N such that if all the clients assigned to that IP are reassigned to n, the load difference between N and n is minimized. For example, in Table 5.1, V_1 has more load than V_2. If we were to balance the load between V_1 and V_2, we would pick 192.168.0.2 since reassigning all clients currently assigned to 192.168.0.2 completely balances out the load on V_1 and V_2. As modifications to client side configuration

*v*FS node	IP Addresses	Load (#req/sec)	Total load (#req/sec)
V_1	192.168.0.1	300	900
	~~192.168.0.2~~	~~300~~	
	192.168.0.3	600	
V_2	192.168.0.2	300	900
	192.168.0.4	100	
	192.168.0.5	200	
	192.168.0.6	300	

Table 5.2: V_1 has more load than V_2. Reassigning clients currently associated with 192.168.0.2 from V_1 to V_2 achieves perfect balance.

or software are not an option, the "reassignment" of clients is actually achieved by rebinding the chosen 192.168.0.2 to V_2, a process we call *IP-address takeover*. The online load balancer first instructs V_1 to unbind from 192.168.0.2, and then upon confirmation from V_1 requests that V_2 bind to 192.168.0.2. The IP address configuration of the sample *v*FS when the address takeover completes is described in Table 5.2, all client requests send to 192.168.0.2 will be handled by V_2.

The IP address binding and rebinding rely on the Address Resolution Protocol [51], a standard protocol designed to resolve an IP address to a Media Access Control (MAC) layer Ethernet address, the ultimate network interface card (NIC) identity used by hardware to deliver bits. If 192.168.0.2 is mapped to the NIC address of V_1, requests destined to 192.168.0.2 will be delivered to V_1, but if 192.168.0.2 is mapped to the NIC address of V_2, those requests will be delivered to V_2 instead. By binding to an IP address p, the node broadcasts the mapping from p to its NIC address periodically and also responds with its NIC address to queries from any machine in the same subnet trying to resolve p. It is this extra layer of IP address to MAC address translation that frees clients from fixed bindings to *v*FS nodes.

Note that, this form of load balancing is not fine-grained, e.g., at an individual

request level, but IP address based load transferring among *v*FS nodes is very fast and very effective. The drawback is that the short term load spikes may not be handled gracefully.

5.3 Failover

The IP address takeover process is also critical to the transparent failover of a *v*FS node to another in masking node failures. In a *v*FS federation, each *v*FS node, say A, is peered with another *v*FS node, say B within the same subnet. Constantly, A and B send to each other probing requests; these probing requests when received are placed in a separate high priority queue and are responded to immediately with the list of IP addresses the nodes bind to. If responses are not received from its peer after a few tries, a node considers its peer dead and starts taking over all its peer's IP addresses through a process similar to what is described in Section 5.2. The whole process is transparent to all clients, though the underline RPC protocol may need to retransmit some timed-out messages, which is also an automatic process. Apparently, this failover mechanism only works under the "fail-stop" failure model.

Due to the stateless nature of *v*FS nodes, the failover process is almost painless. However, when A takes over all the identifies of its dead peer B, it goes through the same crash recovery procedure mostly to ensure that it assumes B's choke point responsibility, as described in Section 4.1.5, as if B just rebooted quickly.

5.4 Summary

*v*FS nodes not only manage the backend file servers as a resource pool, making them interchangeable, but also form a federation among themselves to achieve better re-

liability and better load balancing. A *v*FS federation relies on a mount-time load balancer, an intelligent DNS server with *v*FS node load knowledge, to assign new clients to lightly loaded *v*FS nodes. Once new clients are assigned to *v*FS nodes, the load balancing responsibility shifts to the online load balancer. The online load balancer monitors the load on each *v*FS node periodically, and instructs heavily loaded nodes to shift portions of the load to lightly loaded nodes. The load shifting process employs a technique called IP address takeover that allows *v*FS nodes to dynamically bind to different identities. Besides balancing load among themselves, *v*FS nodes also monitor the health of one another and take over all the identifies and assume all the responsibilities of failed nodes to provide uninterrupted file service to clients.

Chapter 6

Evaluation

This chapter starts by briefly describing a *v*FS prototype built to estimate the implementation complexity of file server virtualization. The prototype is also used by a few sets of experiments designed to establish the feasibility of the new file virtualization layer. In these experiments, a cluster of about 30 machines spread out to three racks function as NFS servers, NFS clients and *v*FS nodes. All of these machines have identical configurations: dual Pentium III 1GHz CPUs, 2GB memory, a 36GB 15K RPM internal disk, and a 1Gbps ethernet interface. They all run Fedora Core 2 (Linux version 2.6.5) as their operating system.

The overhead of the *v*FS prototype is studied in Section 6.2. It focuses on the additional user perceived latency introduced by the *v*FS layer for various types of NFS requests, and based on the CPU time spent on each NFS request estimates the throughput a highly optimized implementation may achieve. Section 6.3 studies the scalability of the prototype as the number of servers managed increases and as the number of *v*FS nodes operating in a federation collaboratively increases. The *v*FS node failover characteristics are demonstrated in Section 6.4 and it shows that a *v*FS

node can fail over to a different one well before the UDP-based RPC messages timeout and thus doesn't causing any file service disruption. Section 6.5 shows that *v*FS federations can dynamically and transparently migrate data to increase throughput and to balance the load among the file servers.

For throughput-related experiments, the industry standard SPECsfs97 [65] workload generated by the Fstress NFS benchmark tool [6] is used. Fstress is a freely available, low-overhead load generator widely used to emulate the SPECsfs workload with high accuracy.

6.1 *v*FS Prototype Implementation

We have implemented a *v*FS prototype on Linux that runs in user space, including the *v*FS nodes, RPDS, the migration coordinator, the mount-time *v*FS node load balancer and the online *v*FS node load balancer. The prototype supports NFS version 3 over both UDP and TCP for normal traffic, but online *v*FS load balancing and transparent failover are supported only for NFS over UDP. The communication among *v*FS nodes and various other *v*FS components uses standard RPC over UDP configured for a fixed number of retransmissions with fixed time outs. Each of these components has a few event-driven non-blocking "fast path" threads and a few "slow path" threads that communicate with each other via local sockets. The number of fast path threads is usually equal to the number of CPUs/cores to fully exploit hardware computation resources, and they process all the requests that only require relatively simple logic, more akin to a packet filter for NFS requests/responses. The slow path threads process requests that require communication with other components in the system. For example, a simple NFS request is processed in a fast path thread if it needs to be

translated and forwarded to only one server and the required translation information is available locally. If the translation information is only available through RPDS, the request is passed on to slow path threads.

The main programming language used for the prototype is C++. Various user space utilities, load monitors and load balancers are implemented in Perl. The total number of lines of code adds up to around 30,000. A rough estimate of a full fledged implementation would double the number of lines of code, still very easily manageable.

6.2 Virtualization Overhead

This section mainly studies the overhead introduced by the *v*FS nodes when they interpose on the file server requests and replies and apply protocol virtualization. For this purpose, the client perceived latency is measured and the following seven NFS operations are chosen as our "microbenchmark": 1) NULL RPC call, 2) LOOKUP of a file handle, 3) READ one byte (RDS), 4) READ 4 KB data (RDL), 5) READDIRPLUS of a directory containing one single file (DIRS), 6) READDIRPLUS of a directory containing 32 files (DIRL), and 7) CREATE an empty file (CRT). These seven representative operations vary the amount of data contained in the requests/replies, the complexity and the amount of work that a *v*FS node has to perform for virtualization. The latency measurement experiments compare the perceived latency in three different cases: the baseline case, the *v*FS case and the RPC case. In the baseline (labeled Direct) case, a single client accesses the file server directly, corresponding to the traditional way of accessing an NFS server. The other two cases are accessing the file server through a *v*FS node and through a simple RPC forwarder that only passes requests and replies back and forth with minimal amount of intervening processing. The latter represents

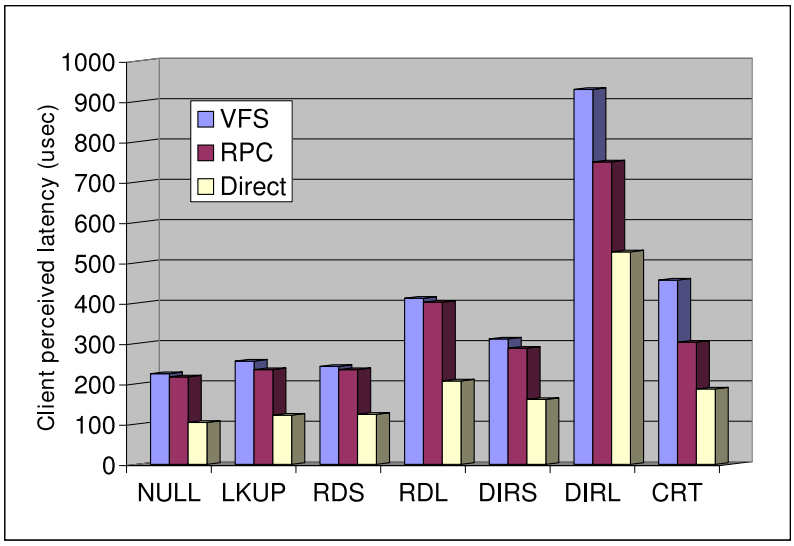

Figure 6.1: The client perceived latency of different NFS operations in three setups: accessing the server directly (Direct), accessing it through a RPC Forwarder (RPC) and accessing it through *v*FS (*v*FS).

the baseline for any in-band virtualization scheme.

The average latency measurement results are shown in Figure 6.1. For most of the operations, both the RPC-forwarder and the *v*FS roughly doubles the client perceived latency. This represents the cost of traversing the network *twice*, and it can only be improved slightly by implementing the packet forwarding in the kernel. In other cases, the extra latency incurred is proportional to the amount of virtualization work that needs to be performed. For instance, in the DIRL case where the overhead is proportional to the number of entries in the directory, the client perceived latency is only about 25% more than the simple RPC-forwarder, and is less than twice the original latency. The CREATE operation's overhead is also higher as it needs to store the newly created file handle v-table map entry to RPDS; this incurs additional

network accesses over the simple RPC forwarding case. Note that the latency numbers shown here are derived from the most pessimistic scenario in which the network latency between client machines and *v*FS nodes are the same as that between *v*FS nodes and file servers. In a more realistic setup, *v*FS nodes and file servers are physically close to each other while client machines are a few switches away. The extra network traversal between *v*FS nodes and file servers therefore may be insignificant. Section 6.3 will also demonstrate that the extra network overhead is usually dwarfed by disk latency as long as a file server is normally loaded (less than 50% of capacity).

The CPU overhead of handling each NFS operation puts an upper bound of throughput on a *v*FS node or a RPC forwarder. While processing a request, a RPC forwarder consumes CPU cycles context switching between the kernel and the user space and performing memory copy. On top of that, a *v*FS node incurs additional processing for parsing/reassembling RPC messages and virtualization. A heavily optimized implementation of *v*FS would reside in the kernel to avoid context switches and reduce memory copy, so the difference between the overheads introduced by a *v*FS node and a RPC forwarder approximates the "true" virtualization overhead. Table 6.1 compares the CPU processing overhead for the *v*FS and the RPC forwarding case and shows that the "true" virtualization overhead is small, within 16 microseconds, for the majority of the NFS operations. Given a highly optimized implementation, each of the slightly older CPU in our servers can handle up to 83,000 NFS operations per second. Of course, this is highly dependent on the operation mix and operations that incur more virtualization processing would bring this number down, to as low as about 4,700 operations per second for the DIRL case.

Op.	NULL	LKUP	RDS	RDL	DIRS	DIRL	CRT
RPC	45	50	50	87	53	103	53
*v*FS	55	66	62	100	84	316	110
diff.	10	16	12	13	31	213	57

Table 6.1: The time in microseconds a RPC forwarder and a *v*FS node spend processing typical operations, and the difference between the two.

6.3 Virtualization Scalability

This section studies the scalability of *v*FS by varying the number of file servers being virtualized and the number of *v*FS nodes collaborating in a virtualization federation. These experiments demonstrate that a *v*FS node can aggregate the throughput of multiple servers. They also show that *v*FS federations can support even more aggregate throughput from the virtualized file servers.

The first set of scalability experiments measures how many servers a single *v*FS node can handle before getting saturated. The prototypical implementation has significant room for optimization, hence, these results are highly pessimistic. Figure 6.2 shows the throughput of a single *v*FS node as the number of file servers being virtualized varies between 1 (1S) and 10 (10S), each handling a portion of the client workload. For each configuration, the same number of load generators as the number of servers are used to generate NFS client load. The request rate at each load generator ranges from 200 NFS operations per second (nfsops) to 1400 nfsops, creating a total load of up to 14,000 nfsops for the 10S configuration. The D curve, showing the throughput of accessing a file server directly, and 1S curve are almost identical. They also clearly reveal the capacity of a single file server: slightly below 2000 nfsops. Until the number of servers managed increases to 10, the *v*FS node is not a bottleneck; the aggregate throughput first grows linearly as each client increases its request

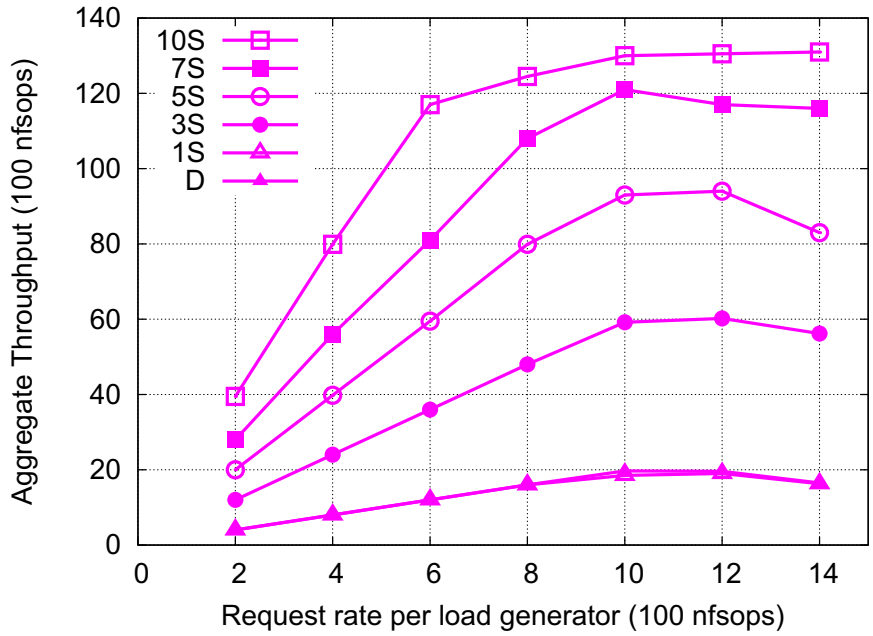

Figure 6.2: Throughput of a single *v*FS node aggregating multiple file servers. The number of file servers used ranges from 1 (1S) to 10 (10S). For each configuration, the number of load generators used is the same as the number of servers. The maximum offered load is 14,000 NFS operations per second.

rate, then starts dropping as the file servers get overloaded. For example, in the 5S setup, the aggregate throughput grows from about 2000 nfsops to about 9500 nfsops, at which point the 5 servers are saturated. Further increasing client request rate only leads to reduced throughput. The 10S curve levels out at around 13,000 nfsops, well below the throughput 10 servers can provide (20,000 nfsops), and further increasing client request rate doesn't lead to increased throughput. This suggests a single *v*FS node can support roughly 7 servers for the experimental configuration. Since CPU speed continues to improve at a faster rate than disk latency does, each *v*FS node is expected to be able to handle more file servers following the technology trend.

Figure 6.3 shows the client perceived latency of NFS operations for the same experiments. Again, the 1S curve and D curve are almost identical under different

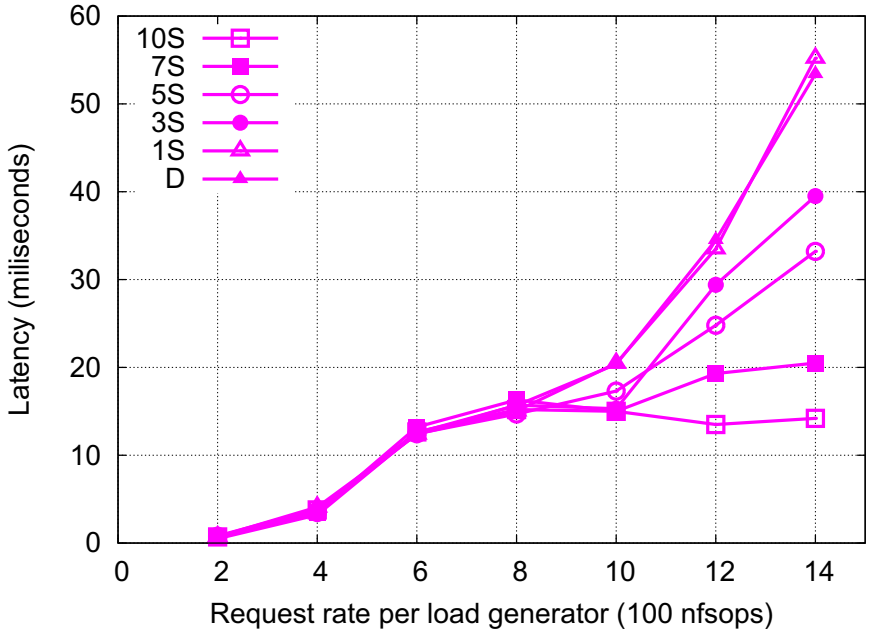

Figure 6.3: Client perceived latency of a single *v*FS node aggregating multiple servers. The *v*FS node does not add noticeable latency under a realistic workload. For each configuration, the number of load generators used is same as the number of servers.

request rates, indicating that the *v*FS node does not add noticeable latency under a realistic workload. In fact, the latency is dominated by disk latency; as the number of file servers being used increases and the load on each server decreases, the client-perceived latency drops.

The second set of scalability experiments measures the throughput of a *v*FS federation consisting of multiple nodes aggregating the throughput of multiple file servers. For these experiment, all *v*FS nodes serve the same virtual volume, so the consistency overhead is included, and the number of *v*FS nodes in a federation varies between 1 (1V) and 5 (5V), with the federation managing 15 file servers capable of providing 30,000 nfsops in aggregate throughput. The number of load generators used is the federation size times the number of file servers (15). In addition, since each *v*FS node is too "powerful", it runs only one "fast-path" thread to limit its capacity. Figure 6.4

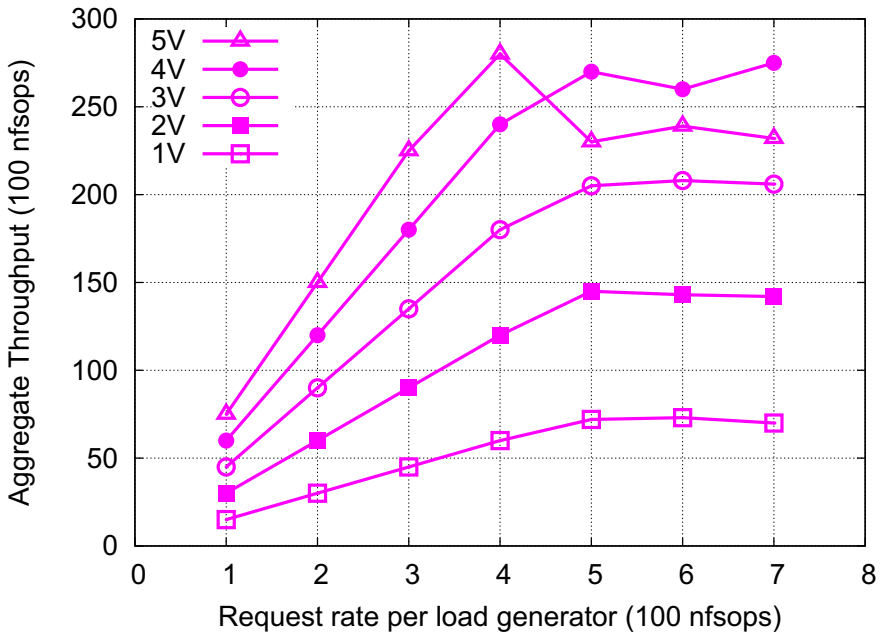

Figure 6.4: A federation of *v*FS nodes can collaboratively manage more servers. The number of *v*FS nodes in a federation varies between 1 and 5. The federation virtualizes 15 file servers. Each *v*FS node uses only one CPU so that more nodes are needed to saturate the file servers. The number of load generators is the federation size times the number of file servers (15).

shows the aggregate throughput of the *v*FS federation observed by clients. The 1V curve indicates that each *v*FS node peaks at about 7,000 nfsops. As the number of *v*FS nodes increases from 1 to 4, the aggregate throughput from the federation scales up almost linearly. Adding one more *v*FS node to the federation increases the total capacity to 35,000 nfsops, exceeding the aggregate throughput of all the file servers; therefore a big drop of the observed throughput from the federation happens when the per load generator request rate increases from 400 nfsops (or 30,000 nfsops in total) to 500 nfsops (or 37,5000 nfsops in total). Further increasing the request rate doesn't lead to drop in throughput as the five *v*FS nodes get saturated and limit the request rate file servers receive.

6.4 Transparent Failover

Recall that any *v*FS node can serve any client request and that this property can be used to support transparent failover among *v*FS nodes. This section studies the behavior of *v*FS nodes in the event of a node failure by measuring the client perceived latency. For these experiments, three client nodes are used to generate client load for a 2-node *v*FS federation. Each client node uses 10 threads to generate NFS requests in a tight loop, one request at a time, with a maximum of 30 requests pending at any given time. Node failures are simulated by turning off the power switch of a *v*FS node at random moments. Only those requests with client perceived latency exceeding the RPC retransmission period are counted. By doing so, the measured latency is biased towards the requests affected by the failover operation. The default RPC retransmission period is 5 seconds, after this period RPC starts retransmitting

Latency	RPC retransmission period			
	500 ms	1000 ms	2000 ms	5000 ms
min	1005 ms	1004 ms	2004 ms	5003 ms
avg	1007 ms	1005 ms	2004 ms	5004 ms
max	1009 ms	1006 ms	2005 ms	5005 ms

Table 6.2: Client perceived latency in the event of failover, with varying retransmission period.

for a maximum of 5 retries[1], i.e., for a period of 25 seconds.

Table 6.2 shows the latency of requests that exceed the RPC retransmission period, ranging from 500 ms to 5000 ms (the default), in the event of a failover. The results indicate that at most two RPC retransmissions are caused by the failover and none of the clients noticed the occurrence of node failures. The heartbeat message interval between the two *v*FS is 200ms and one node assumes its peer fails if it does not receive any response to four consecutive heartbeat messages. This means, the grace period for the failure detection in this experiment is about 800ms. This causes about two retransmissions on average when the RPC retransmission period is 500ms. For larger RPC retransmission periods only one retransmission is needed. The latency of the affected client requests is dominated by the number of retransmissions necessary, as is shown in Table 6.2. It is therefore safe to conclude that the failover mechanism in *v*FS is efficient and that it does not impact file service availability when *v*FS fail.

6.5 Migration

This section evaluates the effectiveness of the transparent migration mechanism for online data reorganization, and shows that *v*FS can improve the throughput of file ser-

[1]In fact, the majority of the NFS volumes in enterprises are mounted in *hard* mode, meaning that clients retry unlimited number of times until they succeed.

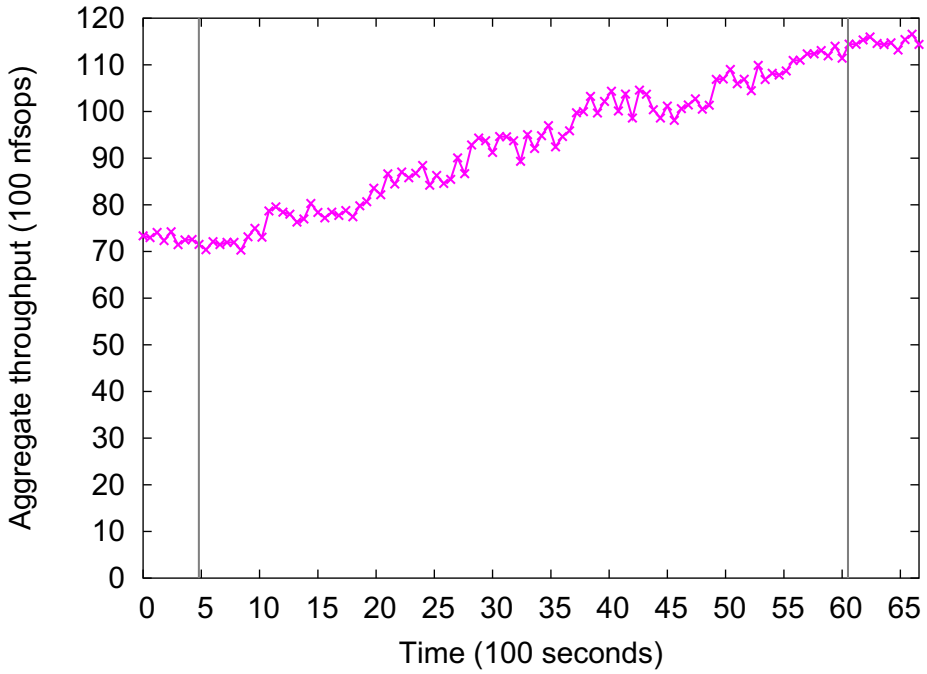

Figure 6.5: Aggregate throughput from a *v*FS federation during transparent online data migration period when it transfers data and load from busy servers to less busy ones.

vice by gradually and transparently migrating files and directories from busy servers to less loaded ones, without disrupting client access or requiring additional hardware/software investment.

Figure 6.5 plots the aggregate throughput from a *v*FS federation, which consists of 3 *v*FS nodes and manages 8 file servers, over a period of around two hours. During the entire period, 12 load generators apply the SPECsfs97 workload at a total rate of about 12,000 nfsops. All the load generators almost exclusively access the data hosted on the first half of the servers through *v*FS nodes, ignoring the rest of the servers. This scenario emulates potentially suboptimal data distribution or the change of data access pattern. The first 4 servers are overloaded by the clients, and thus yield only around 72,000 nfsops aggregate throughput, merely 60% their potential. Observing

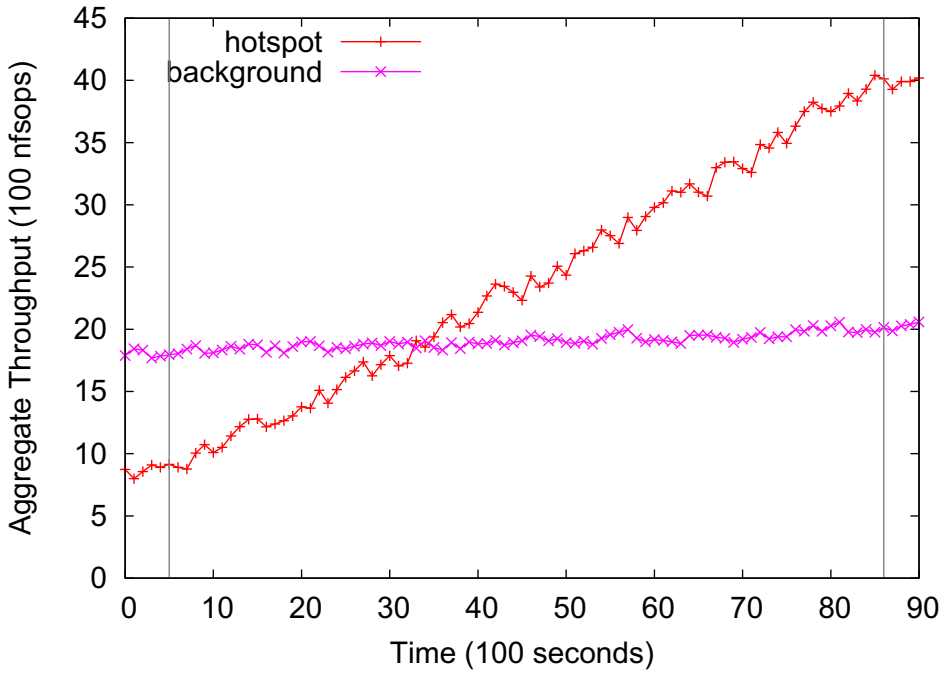

Figure 6.6: Aggregate throughput achieved by "background" clients and "hotspot" clients while a *v*FS federation splits the heavily accessed subtree to less loaded servers.

the load imbalance and attempting to correct it, *v*FS starts migrating a few subtrees from the first half of the servers to the second half of the servers at around 500 seconds from the beginning of the measurements, and finishes at around 6000 seconds. The starting and ending times are marked by the two vertical lines in Figure 6.5. During the one hour and a half period of data migration, *v*FS moved roughly 1/3 of the data, consisting of 3.6 GB data organized in about 120,000 files and directories from the busy file servers to less loaded ones. At the end of the migration, aggregate throughput is improved by about 65%, close to meeting the full throughput expectation from clients.

Besides correcting load imbalance, *v*FS can also alleviate or eliminate file access hotspots through migration. This is demonstrated through another experiment in

which 4 "hotspot" clients try to access the same subtree with 1000 nfsops request rate each and get the server hosting the subtree overloaded. This experiment is also configured with 3 *v*FS nodes and 8 file servers each of which carries a 250 nfsops constant load generated by 8 "background" clients. Figure 6.6 plots the aggregate throughput observed by "hotspot" clients and "background" clients while the *v*FS federation splits the heavily accessed subtree to other less loaded servers. As in Figure 6.5, the starting and ending time of migration are marked by two vertical lines. The "background" clients' throughput is barely affected by the migration, while the "hotspot" clients achieved 5 times throughput improvement when migration finishes.

Since subtrees are migrated file by file and directory by directory and it even generates additional load on the already busy servers, the migration process is by no means fast, but it is able to gradually and smoothly shed the load from busy servers until the throughput contribution from all servers is more balanced. In addition, since only the standard NFS interface is being used to migrate data, additional efficiencies that may result from specialized data transfer protocols (e.g., online compression) are not employed.

6.6 Summary

This chapter briefly described a user space *v*FS prototype on Linux with easily manageable implementation complexity. This prototype was used to measure the virtualization overhead for different NFS operations. The results indicate that most of the delay was caused by dual network traversals which is a minor problem in realistic network setups with realistic work load. A single *v*FS node without optimization is able to manage about 7 file servers in the experimental setup although optimized imple-

mentation is expected to have much improved performance. Adding more *v*FS nodes to a federation can aggregate the throughput of even more servers. Nodes within the same federation are able to take over the responsibility of failed ones quickly and transparently without clients noticing even with very stringent client retry configuration. Transparent online data migration also has proved to be an effective mechanism in correcting load imbalance and access hotspots on file servers.

Chapter 7

Conclusion

7.1 Thesis Contribution

This thesis argued for the separation of file service from file server infrastructure as a feasible approach to avoid file service disruption and to liberate system administrators from the management burden introduced by the deployment of large numbers of independent file servers. In particular, this thesis presented a new architecture called *v*FS to virtualize and federate already deployed file servers in large enterprises. This architecture enables exporting virtual volumes of configurable and scalable capacity and throughput without requiring any changes to the clients, the file servers and the protocol they use. *v*FS employs full protocol virtualization to aggregate the resources of file servers together under one umbrella of control by breaking the client-to-server bindings. *v*FS also introduces transparent migration to support the moving of files and directories among file servers even when they are being actively accessed. Transparent migration breaks the data-to-server bindings and enables virtual volumes to dynamically consumes resources from different file servers. *v*FS nodes exporting

the same virtual volume also form a federation among themselves to achieve load balancing and transparent failover.

This thesis also described a *v*FS prototype implementation in user space on Linux. The initial experiments indicate that the prototype adds very little performance overhead and that a single *v*FS node without optimization can virtualize about seven file servers under industry standard SPECsfs workload. The throughput of a *v*FS federation scales linearly as the number of file servers and *v*FS nodes increases. The experiments also show that *v*FS nodes transfer load or fail over to another with only a few RPC retransmissions, and can transparently migrate data to improve throughput and to balance the load among file servers.

To the best of our knowledge, *v*FS is the first scalable and complete NFS file server virtualization architecture. It also first introduces transparent data migration to NFS servers which were designed to work independently.

7.2 Future Work

There are a few ideas related to file system virtualization that are potentially worth more research effort. *v*FS can already maintain multiple asynchronous copies of subtrees to achieve better reliability and availability, a *v*FS design that supports synchronous subtree replication however is not explored in this thesis. Synchronous subtree replication provides even better reliability, but its disadvantage is its complexity. Given that the consistency semantics of NFS is already fairly loose, chances are that a semi-synchronous subtree replication model exists that may provide an "almost consistent" view to client applications while switching from one copy to another copy online. The complexity of this model should still be fairly manageable.

This thesis described an un-optimized *v*FS prototype and predicted the throughput a highly optimized version may achieve. Such a version however may still need to solve some non-trivial design and implementation issues. One obvious direction for optimization is to move the implementation into the OS kernel to avoid the user-kernel context switches and reduce the number of memory copies. Furthermore, since most of *v*FS processing is very simple and regular, it can even be offloaded onto network cards, further improving throughput and reducing latency [64]. Another performance related area worth further investigation is concurrent migration, as opposed to current sequential file-by-file and directory-by-directory migration, the result of a compromise for a simpler design and implementation. The challenge for concurrent migration, other than its complexity, is to strike a balance between migrating subtrees as fast as possible and interfering with normal client traffic as little as possible.

File servers and database servers are usually both widely deployed to manage unstructured and structured data in enterprises respectively. Virtualization has proved to be effective for managing heterogeneous single-server-architecture file servers, and it will be interesting to explore the idea of applying file server virtualization techniques to managing single-server-architecture database servers from different vendors. The Structured Query Language (SQL) may play the role of a common interface for heterogeneous database servers, similar to the NFS protocol for file servers. The database API is much richer than file system API and its consistency requirement is much stronger. Therefore database virtualization presents significantly more challenge. On the other hand, databases also present more flexible ways of distributing data. For example a database can be distributed by table, by row or by column. Various affinity requirements for different tables, rows or columns need to be satisfied if minimal query performance degradation is desired.

References

[1] File virtualization with Acopia's adaptive resource switching. http://www.acopia.com, 2004.

[2] Windows server 2003 active directory. http://www.microsoft.com/windowsserver2003/.

[3] Easing server sprawl and storage traffic load. http://itmanagement.earthweb.com/netsys/article.php/3305701, 2004.

[4] Content delivery network. http://www.akamai.com.

[5] Paul Albitz and Cricket Liu. *DNS and BIND.* O'Reilly & Associates Inc., Sebastopol, CA., USA, 5th edition.

[6] Darrell Anderson and Jeff Chase. Fstress: A flexible network file service benchmark. Technical Report TR-2001-2002, Department of Computer Science, Duke University, May 2001.

[7] Darrell Anderson, Jeffrey Chase, and Amin Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.

[8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[9] S. Baker and J. H. Hartman. The Mirage NFS router. In *Prof. 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, 2004.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[11] Mike Burrows. The chubby lock service loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI)*, November 2006.

[12] B. Callaghan and S. Singh. The autofs automounter. In *Proc. of the Summer 1993 USENIX Conference*, pages 59–68, Cincinnati, OH, 1993.

[13] Brent Callaghan and Tom Lyon. The automounter. In *Proc. 1989 Winter USENIX Technical Conf.*, pages 43–51, San Diego, California, 30 - 3 1989. USENIX Association.

[14] Common internet file systems (cifs) technical reference. Technical report, Storage Networking Industry Association (SNIA), March 2002.

[15] Mark Crovella, Mor Harchol-Balter, and Cristina D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load (extended abstract). In *Measurement and Modeling of Computer Systems*, pages 268–269, 1998.

[16] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *SIGOPS Operating System Review*, 35(5):202–215, 2001.

[17] Building practical data protection strategies. http://www.datadomain.com/, 2007.

[18] Distributed file system technology center. http://www.microsoft.com/windowsserver2003/.

[19] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[20] Powerpath-automated, non-disruptive path management. http://www.emc.com/, 2007.

[21] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[23] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM Symposium on Operating Systems Principles*, pages 150–162, New York, NY, USA, 1979. ACM Press.

[24] The Open Group. *Protocols for X/Open PC Internetworking: SMB, Version 2.* The Open Group, Oct 1992.

[25] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[26] Christopher Hertel and Christopher R. Hertel. *Implementing CIFS: The Common Internet File System.* Prentice Hall PTR, Aug 2003.

[27] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Francisco, CA, USA, 17–21 1994.

[28] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[29] IDC worldwide disk storage systems quarterly tracker. http://www.idc.com/getdoc.jsp?containerid=IDC_p4435.

[30] Mahesh Kallahalla, Mustafa Uysal, Ram Swaminathan, David E. Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris I. Dalton, and Frederic Gittler. SoftUDC: A software-based data center for utility computing. *Computer*, 37(11):38–46, 2004.

[31] Wataru Katsurashima, Satoshi Yamakawa, Takashi Torii, Jun Ishikawa, Yoshihide Kikuchi, Kouji Yamaguti, Kazuaki Fujii, and Toshihiro Nakashima. NAS switch: A novel CIFS server virtualization. In *MSS '03: Proceedings of the 20*

th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03), page 82. IEEE Computer Society, 2003.

[32] Eric Dean Katz, Michelle Butler, and Robert McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27(2):155–164, 1994.

[33] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[34] Mike Kazar. Spinserver systems and linux compute farms. Technical Report TR-3304, Network Appliance, February 2004.

[35] Andrew J. Klosterman and Gregory Ganger. Cuckoo: Layered clustering for NFS. Technical Report CMU-CS-02-183, Carnegie Mellon School of Computer Science, October 2002.

[36] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[37] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, 2001.

[38] Lightweight directory access protocol (v3). http://www.ietf.org/rfc/rfc2251.txt, December 1997.

[39] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST '02: Proceedings of the 1st*

USENIX Conference on File and Storage Technologies, page 21, Berkeley, CA, USA, 2002. USENIX Association.

[40] Lustre file system. http://www.lustre.org.

[41] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[42] K. A. Marzullo. *Maintaining the Time in a Distributed System: An Example of a Loosely-Coupled Distributed Service.* Ph.D. dissertation, Stanford University, Department of Electrical Engineering, February 1984.

[43] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[44] National Institute of Standards and Technology (NIST). *Publication YY: Announcement and Specifications for a Secure Hash Standard (SHS)*, January 22, 1992.

[45] NFS: Network file system protocol specification. http://www.ietf.org/rfc/rfc1094.txt, 1989.

[46] NFS version 3 protocol specification. http://www.ietf.org/rfc/rfc1813.txt, June 1995.

[47] Network time protocol (version 3) specification, implementation and analysis. http://www.ietf.org/rfc/rfc1305.txt, 1992.

[48] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: a general primary copy. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM Press.

[49] Panasas active scale file system (PanFS). http://www.panasas.com, 2004.

[50] Greg Papadopoulos. Moore's law ain't good enough. In *Keynote speech at Hot Chips X*, August 1998.

[51] David C. Plummer. An Ethernet Address Resolution Protocol. Technical Report 825, September 1982.

[52] Matrix server architecture. http://www.polyserve.com, 2004.

[53] Herman C. Rao and Larry L. Peterson. Accessing files in an internet: The jade file system. *Software Engineering*, 19(6):613–624, 1993.

[54] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[55] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Operating System Review*, 35(5):188–201, 2001.

[56] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[57] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Operating System Review*, 38(5):48–58, 2004.

[58] Samba. http://www.samba.org/.

[59] Raja R. Sambasivan, Andrew J. Klosterman, and Gregory R. Ganger. Replication policies for layered clustering of nfs servers. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 361–370, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[61] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. 2002.

[62] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[63] Storage Networking Industry Association. http://www.snia.org/.

[64] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 216–229, New York, NY, USA, 2001. ACM.

[65] Standard Performance Evaluation Corporation. SPEC SFS 3.0 run and report rules, 2001.

[66] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceeding of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[67] Veritas storage foundation 5.0 dynamic multi-pathing. http://eval.symantec.com/, May 2007.

[68] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[69] Limin Wang, Kyoung Soo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and security in the CoDeeN content distribution network. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[70] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Operating System Review*, 36(SI):195–209, 2002.

[71] XDR: External data representation standard. http://www.ietf.org/rfc/rfc1832.txt, 1995.

[72] Ken Yocum, Darrell Anderson, Jeff Chase, and Amin Vahdat. Anypoint: Extensible transport switching on the edge. In *Proc. 4th USENIX USITS*, March 2003.

[73] Xiang Yu. *Trading Capacity for Performance in Disk Arrays*. PhD thesis, Princeton University, 2003.