Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks

Ben Pfaff, Tal Garfinkel, Mendel Rosenblum
{blp,talg,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Virtual disks are the main form of storage in today's virtual machine environments. They offer many attractive features, including whole system versioning, isolation, and mobility, that are absent from current file systems. Unfortunately, the low-level interface of virtual disks is very coarse-grained, forcing all-or-nothing whole system rollback, and opaque, offering no practical means of sharing. These problems impose serious limitations on virtual disks' usability, security, and ease of management.

To overcome these limitations, we offer Ventana, a *virtualization aware file system*. Ventana combines the file-based storage and sharing benefits of a conventional distributed file system with the versioning, mobility, and access control features that make virtual disks so compelling.

1 Introduction

Virtual disks, the main form of storage in today's virtual machine environments, have many attractive properties, including a simple, powerful model for versioning, rollback, mobility, and isolation. Virtual disks also allow VMs to be created easily and stored economically, freeing users to configure large numbers of VMs. This enables a new usage model in which VMs are specialized for particular tasks.

Unfortunately, virtual disks have serious shortcomings. Their low-level isolation prevents shared access to storage, which hinders delegation of VM management, so users must administer their own growing collections of machines. Rollback and versioning takes place at the granularity of a whole virtual disk, which encourages mismanagement and reduces security. Finally, virtual disks' lack of structure obstructs searching or retrieving data in their version histories [34].

Conversely, existing distributed file systems support fine-grained controlled sharing, but not the versioning, isolation, and encapsulation features that make virtual disks so useful.

To bridge the gap between these two worlds, we present Ventana, a *virtualization aware file system* (VAFS). Ventana extends a conventional distributed file system with versioning, access control, and disconnected operation features resembling those available from virtual disks. This attains the benefits of virtual disks, without compromising usability, security, or ease of management.

Unlike traditional virtual disks whose allocation and composition is relatively static, in Ventana storage is ephemeral and highly composable, being allocated on demand as a *view* of the file system. This allows virtual machines to be rapidly created, specialized, and discarded, minimizing the storage and management overhead of setting up a new machine.

We describe the principles behind virtualization aware file systems. We also present our prototype implementation of Ventana and explore the practical benefits that a VAFS offers to VM users and administrators.

We will begin by examining the properties of virtual disks. Section 2 explores their limitations, to motivate our desire for file system-based virtual machine storage, then Section 3 details virtual disks' compelling features. Section 4 shows how to integrate these features into a distributed file system by presenting Ventana, a virtualization aware file system. Section 5 focuses on our prototype implementation of Ventana and Section 6 demonstrates a usage scenario. Sections 7 and 8 discuss related and future work and Section 9 concludes.

2 Motivation

Virtual machines are changing the way that users perceive a "machine." Traditionally, machines were static entities. Users had one or a few, and each machine was treated as general-purpose. The design of virtual machines, and even their name, has largely been driven by this perception.

However, virtual machine usage is changing as users discover that a VM can be as temporary as a file. VMs can be created and destroyed at will, checkpointed and versioned, passed among users, and specialized for particular tasks. Virtual disks, that is, files used to simulate disks, aid these more dynamic uses by offering fully encapsulated storage, isolation, mobility, and other benefits that will be discussed fully in Section 3.

Before that, to motivate our work, we will highlight the significant shortcomings of virtual disks. Most importantly, virtual disks offer no simple way to share read and write access between multiple parties, which frustrates delegating VM management. At the same time, the dynamic usage model for VMs causes them to proliferate, which introduces new security and management risks and makes such delegation sorely needed [9, 31].

Second, although it is easy to create multiple hierarchical versions of virtual disks, other important activities are difficult. A normal file system is easy to search with command-line or graphical tools, but searching through multiple versions of a virtual disk is a cumbersome, manual process. Deleting sensitive data from old versions of a virtual disk is similarly difficult.

Finally, a virtual disk has no externally visible structure, which forces entire disks to roll back at a time, despite the possible negative consequences [9]. Whether they realize it or not, whole-disk rollback is hardly ever what people actually want. For example, system security precludes rolling back password files, firewall rules, encryption keys, and binaries patched for security, and functionality may be impaired by rolling back network configuration files. Furthermore, the best choice of version retention policy varies from file to file [23], but virtual disks can only distinguish version policies on a whole-disk level.

These limitations of virtual disks led us to question why they are the standard form of storage in virtual environments. We concluded that their most compelling feature is compatibility. All of their other features can be realized in a network file system. By adopting a widely used network file system protocol, we can even achieve reasonable compatibility.

The following section details the virtual disk features that we wish to integrate into a network file system. The design issues raised in this integration are then covered in Section 4.

3 Virtual Disk Features

Virtual disks are, above all, backward compatible, because they provide the same block-level interface as physical disks. This section examines other important features that virtual disks offer, such as versioning, isolation, and encapsulation, and the usage models that they enable. This discussion shapes the design for Ventana presented in the next section.

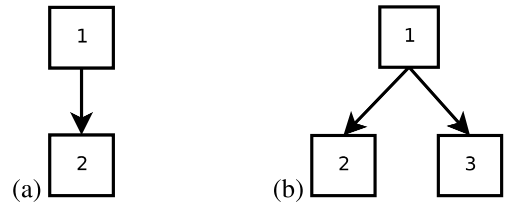

FIGURE 1: Snapshots of a VM: (a) first two snapshots; (b) after resuming again from snapshot 1, then taking a third snapshot.

3.1 Versioning

Because any saved version of a virtual machine can be resumed any number of times, VM histories take the form of a tree. Consider a user who "checkpoints" or "snapshots" a VM, permanently saving the current version as version 1. He uses the VM for a while longer, then checkpoints it again as version 2. So far, the version history is linear, as shown in Figure 1(a). Later, he again resumes from version 1, uses it for a while, then snapshots it another time as version 3. The tree of VMs now looks like Figure 1(b). The user can resume any version any number of times and create new snapshots based on these existing versions, expanding the tree.

Virtual disks efficiently support this tree-shaped version model. A virtual disk starts with an initial or "base" version that contains all blocks (all-zero blocks may be omitted), corresponding to snapshot 1. The base version may have any number of "child" versions, and so may those versions recursively. Thus, like virtual machines, the versions of virtual disks form a tree. Each child version contains only a pointer to its parent and those blocks that differ from its parent. This copy-on-write sharing allows each child version to be stored in space proportional to the differences between it and its parent. Some implementations also support content-based sharing that shares identical blocks regardless of parent/child relationships.

Virtual disk versioning is useful for short-term recovery from mistakes, such as inadvertently deleting or corrupting files, or for long-term capture of milestones in configuration or development of a system. Linear history also effectively supports these usage models. But hierarchical versions offer additional benefits, described below.

Specialization Virtual disks enable versions to be used for specialization, analogous to the use of inheritance in object-oriented languages. Starting from a base disk, one may fork multiple branches and install a different set of applications in each one for a specialized task, then branch these for different projects, and so on. This is easily supported by virtual disks, but today's file systems have no close analogue.

Non-Persistence Virtual disks support "non-persistent storage." That is, they allow users to make temporary changes to disks during a given run of a virtual machine, then throw away those changes once the run is complete. This usage pattern is handy in many situations, such as software testing, education, electronic "kiosk" applications, and honeypots. Traditional file systems have no concept of non-persistence.

3.2 Isolation

Everything in a virtual machine, including virtual disks, exists in a protection domain decoupled from external constraints and enforcement mechanisms. This supports important changes in what users can do.

Orthogonal Privilege With the contents of the virtual machine safely decoupled from the outside world, access controls are put into the hands of the VM owner (often a single user). There is thus no need to couple them to a broader notion of principals. Users of a VM are provided with their own "orthogonal privilege domain." This allows the user to use whatever operating systems or applications he wants, at his discretion, because he is not constrained by the normal access control model restricting who can install what applications.

Name Space Isolation VMs can serve in the same role filled by `chroot`, BSD `jails`, application sandboxes, and similar mechanisms. An operating system inside a VM can even be easier to set up than more specialized, OS-specific jails that require special configuration. It is also easier to reason about the security of such a VM than about specialized OS mechanisms. A key reason for this is that VMs afford a simple mechanism for name space isolation, i.e. for preventing an application confined to a VM modifying outside system resources. The VM has no way to name anything outside the VM system without additional privilege, e.g. access to a shared network. A secure VMM can isolate its VMs perfectly.

3.3 Encapsulation

A virtual disk fully encapsulates storage state. Entire virtual disks, and accompanying virtual machine state, can easily be copied across a network or onto portable media, notebook computers, etc.

Capturing Dependencies The versioning model of virtual disks is coarse-grained, at the level of an entire disk. This has the benefit of capturing all possible dependencies with no extra effort from the user. Thus, short-term "undo" using a virtual disk can reliably back out operations with complex dependencies, such as installation or removal of a major application or device driver, or a complex, automated configuration change.

Full capture of dependencies also helps in saving milestones in the configuration of a system. The snapshot will not be broken by subsequent changes in other parts of the system, such as the kernel or libraries, because those dependencies are part of the snapshot [13].

Finally, integrating dependencies simplifies and speeds branching. To start work on a new version of a project or try out a new configuration, all the required pieces come along automatically. There is no need to again set up libraries or configure a machine.

Mobility A virtual disk can be copied from one medium to another without retaining any tie to its original location. Thus, it can be used while disconnected from the network. Virtual disks thereby offer mobility, the ability to pick up a machine and go.

Merging and handling of conflicts has long been an important problem for file systems that support disconnected operation [16], but there is no automatic means to merge virtual disks. Nevertheless, virtual disks are useful for mobility, indicating that merging is not important in the common case. (In practice, when merging is important, users tend to use revision control systems.)

4 Design

This section describes Ventana, an architecture for a virtualization aware file system. Ventana resembles a conventional distributed file system in that it provides centralized storage for a collection of file trees, allowing transparency and collaborative sharing among users. Ventana's distinction is its versioning, isolation, and encapsulation features to support virtualization, based on virtual disk support for these same features,

The high-level architecture of Ventana can apply to various low-level architectures: centralized or decentralized, block-structured or object-structured, etc. We restrict this section to essential, high-level design elements. The following section discusses specific choices made in our prototype.

We adopt the convention that an operating system inside a virtual machine is a *guest OS*. Ventana's clients run in virtual machines.

Ventana offers the following abstractions:

Branches Ventana supports VM-style versioning with *branches*. A *private branch* is created for use primarily by a single VM, making the branch effectively private, like a virtual disk. A *shared branch* is intended for use by multiple VMs. In a shared branch, changes made from one VM are visible to the others, so these branches can be used for sharing files, like a conventional network file system.

Non-persistent branches, whose contents do not survive across reboots are also provided, as are *volatile branches*, whose contents are never stored on a central server, and are deleted upon migration. These features are especially useful for providing storage for caches and cryptographic material that for efficiency or security reasons, respectively, should not be stored or migrated.

Branches are detailed in Section 4.1.

Views Ventana is organized as a collection of file trees. To instantiate a VM, a *view* is constructed by mapping one or more of these trees into a new file system name space. For example, a base operating system, add-on applications, and user home directories might each be mounted from a separate file tree.

This provides a basic model for supporting name space isolation and allows for rapid synthesis of new virtual machines, without the space or managment overhead normally associated with setting up a new virtual disk.

Section 4.2 describes views in more detail.

Access Control File permissions in Ventana must satisfy two kinds of needs: those of the guest OSes to partition functionality according to the guests' own principals, and those of users to control access to confidential information. Ventana provides orthogonal types of *file ACLs* to satisfy these needs.

Ventana also offers *branch ACLs* which support common VM usage patterns, such as one user granting others permission to clone a branch and modify the copy (but not the original), and *version ACLs* which alleviate security problems introduced by file versioning.

Section 4.3 describes access control in Ventana.

Disconnected Operation Ventana allows for a very simple model of mobility by supporting disconnected operation, through a combination of aggressive caching and versioning. Section 4.4 talks about disconnected operation in Ventana.

4.1 Branches

Some conventional file systems support versioning of files and directories. Details about which versions are retained, when older versions are deleted, and how older versions are named vary. However, in all of them, versioning is "linear," that is, at any point in each file has a unique latest version.

When versions form a tree that grows in more than one direction, asking for the latest version of a file can be ambiguous. The file system must provide a way for users to express where in the tree to look for a file version.

To appreciate these potential ambiguities, consider an example. Ziggy allows Yves, Xena, and Walt to each fork a personalized version of her VM. The version tree for a file personalized by each person would look something like Figure 2(a). If an access to a file by default refers to the latest version anywhere in the tree, then each person's changes would appear in the others' VMs. Thus, the tree of versions would act like a chain of linear versions.

In a different situation, suppose Vince and Uma use a shared area in the file system for collaboration. Most of the time, they do want to see the latest version of a file. Thus, the version history of such a file should be linear, with each update following up on the previous one, resembling Figure 2(b).

The essential difference between these two cases is intention. The version tree alone cannot distinguish between desires for shared or personalized versions of the file system without knowledge of intention.

Consider another file in Ziggy's VM. If only Yves has created a personalized version of the file, then the version tree looks like Figure 2(c). The shape of this tree cannot be distinguished from an early version of Figure 2(b). Thus, Ventana must provide a way for users to specify their intentions.

4.1.1 Private and Shared Branches

Ventana introduces *branches* to resolve the above difficulty. A branch is a linear chain in the tree of versions. Because a branch is linear, it is meaningful to refer to the latest version of a file in a branch, or the version at a particular point in time.

A branch begins as an exact copy of the contents of some other branch at the current time, or at a chosen earlier time. After creation, the new branch and the branch that was copied are independent, so that modifying one has no effect on the other.

Branches are created by copying. Thus, multiple branches may contain the same version of a file. Therefore, for a file access to be unambiguous, both a branch and a file must be specified. Mounting a tree in a virtualization aware file system requires specifying the branch to mount.

If a single client wants a private copy of the file tree, a *private branch* is created for its exclusive use. Like a file system on a virtual disk, a private branch will only be modified by a single client in a single VM, but in other respects it resembles a conventional network file system. In particular, access to files by entities other than the guest that "owns" the branch is easily possible, enabling centralized management such as scanning for malware, file backup, and tracking VM version histories.

If multiple clients mount the same branch of a Ventana file tree, then those clients see a shared view of the files

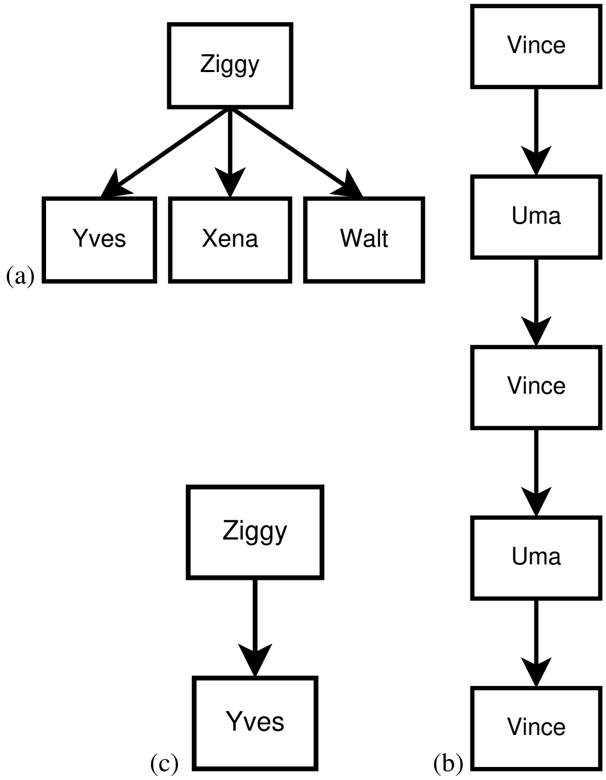

FIGURE 2: Trees of file versions when (a) Ziggy allows Yves, Xena, and Walt to fork personalized versions of his VM; (b) Vince and Uma collaboratively edit a file; and (c) Ziggy's VM has been forked by Yves, as in (a), but not yet by Xena or Walt.

it contains. As in a conventional network file system, a change made by one client in such a *shared branch* will be immediately visible to the others. Of course, propagation of changes between clients is still subject to the ordinary issues of cache consistency in a network file system.

The distinction between shared and private branches is simply the number of clients expected to write to the branch. If necessary, centralized management tools can modify files in a so-called "private" branch (e.g. to quarantine malware) but this is intended to be uncommon. Either type of branch might have any number of read-only clients.

A single file might have versions in shared and private branches. For example, a shared branch used for collaboration between several users might be forked off into a private branch by another user for some experimental changes. Later, the private branch could be discarded or consolidated into the shared branch.

4.1.2 Other Types of Branches

In addition to shared and private branches, there are several other useful qualifiers to attach to file trees.

Files in a *non-persistent branch* are deleted when a VM is rebooted. These are useful for directories of temporary files such as `/tmp`.

Files in a *volatile branch* are also deleted on reboot. They are never stored permanently on the central server, and are deleted when a VM is migrated from one physical machine to another. They are useful for caches (e.g. `/var/cache` on GNU/Linux) that need not be migrated and for storing security tokens (e.g. Kerberos tickets) that should not reside on a central server.

Maintaining any version history for some files is an inherent security risk [9]. For example, the OpenSSL cryptography library stores a "random seed" file in the file system. If this is stored in a snapshot, every time a given snapshot is resumed, the same random seed will be used. In the worst case, we will see the same sequence of random numbers on every execution. Even in the best case, its behavior may be easier to predict, and if old versions are kept, then it may be possible to guess past behavior (e.g. keys generated in past runs).

Ventana offers *unversioned files* as a solution. Unversioned files are never versioned, whether linearly or in a tree. Changes always evolve monotonically forward with time. Applications for unversioned files include storing cryptographic material, firewall rules, password files, or any other configuration state where rollback would be problematic.

4.2 Views

Ventana is organized as a set of file trees, each of which contains related files. For example, some file trees might contain root file systems for booting various operating systems (Linux, Windows XP, ...) and their variants (Debian, Red Hat, SP1, SP2, ...). Another might contain file systems for running various local or specialized applications. A third would have a hierarchy for each user's files.

Creating a new VM mainly requires synthesizing a *view* of the file system for the VM. This is accomplished by mapping one or more trees (or parts of trees) into a new namespace. For example, the Debian root file system might be combined with a set of applications and user home directories. Thus, OSes, applications, and users can easily "mix and match" in a Ventana environment.

Whether each file tree in a view is mounted in a shared or a private branch depends on the user's intentions. The root file system and applications could be mounted in private branches to allow the user to update and modify his own system configuration. Alternately, they could be mounted in shared branches (probably read-only) to allow maintenance to be done by a third party. In the latter case, some parts of the file system would still need to be private, e.g. `/var` under GNU/Linux. Home directories would likely be shared, to allow the user to see a consistent view of his and others' files regardless of the VM viewing them.

4.3 Access Control

Access control is different in virtual disks and network file systems. The guest OS controls every byte on a virtual disk. It is responsible for tracking ownership and permissions and making access control decisions in the file system. The virtual disk itself has no access control responsibility. A VAFS cannot use this scheme, because allowing every guest OS to access any file, even those that belong to other VMs, is obviously unacceptable. There must be enough control in the system to prevent abuse.

Access control in a conventional network file system is the reverse of the situation for a virtual disk. The file server is ultimately in charge of access control. As a network file system client, a guest OS can deny access to its own processes, but it cannot override the server's refusal to grant access. Commonly, NFS servers deny access as the superuser ("squash root") and CIFS and AFS servers grant access only via principals authenticated to the network.

This style of access control is also, by itself, inappropriate in a VAFS. Ventana should not deny a guest OS control over its own binaries, libraries, and applications. If these were, for example, stored on an NFS server configured to "squash root," the guest OS would not be able to create or access any files as the superuser. If they were stored on a CIFS or AFS server, the guest OS would only be able to store files as users authenticated to the network. In practice this would prevent the guest from dividing up ownership of files based on their function (system binaries, print server, web server, mail server, ...), as many systems do.

Ventana solves the problem of access control through multiple types of ACLs: *file ACLs*, *version ACLs*, and *branch ACLs*. For any access to be allowed, it must be permitted by all three applicable ACLs. Each kind of ACL serves a different primary purpose. The three types are described individually below.

4.3.1 File ACLs

File ACLs provide protection on files and directories that users conventionally expect and OSes conventionally provide. Ventana supports two types of file ACLs that provide orthogonal privileges. *Guest file ACLs* are primarily for guest OS use. Guest OSes have the same level of control over guest file ACLs that they do over permissions in a virtual disk. In contrast, *server file ACLs* provide protection that guest OSes cannot bypass, similar to permissions enforced by a conventional network file server.

Both types of file ACLs apply to individual files. They are versioned in the same way as other file metadata. Thus, revising a file ACL creates a new version of the file with the new file ACL. The old version of the file continues to have the old file ACL.

Guest file ACLs are managed and enforced by the guest OS using its own rules and principals. Ventana merely provides storage. These ACLs are expressed in the guest OS's preferred form. We have so far implemented only the 9-bit `rwxrwxrwx` access control lists used by the Unix-like guest OSes. Guest file ACLs allow the guest OS to divide up file privileges based on roles.

Server file ACLs, the other type of file ACL, are managed and enforced by Ventana and stored in Ventana's own format. Server file ACLs allow users to control access to files across all file system clients.

4.3.2 Version ACLs

A version ACL applies to a version of a file. They are stored as part of a version, not as file metadata, so that changing a version ACL does not create a new file version. Every version of a file has an independent version ACL. Conversely, when multiple branches contain the same version of a file, that single version ACL applies in each case. Version ACLs are not versioned themselves. Like server file ACLs, version ACLs are enforced by Ventana itself.

Version ACLs are Ventana's solution to a class of security problem common to all versioning file systems. Suppose Terry creates a file and writes confidential data to it. Soon afterward, Terry realizes that the file's permissions incorrectly allow Sally to read it, so he corrects the permissions. In a file system without versioning, the file would then be safe from Sally, as long as she had not already read it. If the permissions on older file versions are fixed, however, Sally can still access the older version of the file.

A partial solution to Terry's problem is to grant access to older versions based on the current version's permissions, as Network Appliance filers do [32]. Now, suppose Terry edits a file to remove confidential information, then grants read permission to Sally. Under this rule, Sally can then view the older, confidential versions of the file, so this rule is also flawed.

Another idea is to add a permission bit to each file's metadata that determines whether a user may read a file once it has been superseded by a newer version, as in the S4 self-securing storage system [27]. Unfortunately, modifying permissions creates a new version (as does any change to file metadata) and only the new version is changed. Thus, this permission bit is effective only if the user sets it before writing confidential data, so it would not protect Terry.

Only two version rights exist. The "r" (read) version right is Ventana's solution to Terry's problem. At any time, Terry can revoke the read right on old versions of files he has created, preventing access to those file versions. The "c" (change) right is required to change a version ACL. It is implicitly held by the creator of a version. (Any given file version is immutable, so there is no "write" right.)

4.3.3 Branch ACLs

A branch ACL applies to all of the files in a particular branch and controls access to current and older versions of files. Like version ACLs, branch ACLs are accessed with special tools and enforced by Ventana.

The "n" (newest) branch right permits read access to the latest version of files in a branch. It also controls forking the latest version of the branch.

In addition to "n", the "w" (write) right is required to modify any files within a branch. A user who has "n" but not "w" may fork the branch. Then, as owner of the new branch, he may change its ACL and modify the files in the new branch. This does not introduce a security hole because the user may only modify the files in the new branch, not those in the old branch. The user's access to files in the new branch are, of course, still subject to Ventana file ACLs and version ACLs.

The "o" (old) right is required to access old versions of files within a branch. This right offers an alternate solution to Terry's problem of insecure access to old versions. If Terry controls the branch in which the old versions were created, then he can use its branch ACL to prevent other users from accessing old versions of any file in the branch. This is thus a simpler but less focused approach than adjusting the appropriate version ACL.

The "c" (change) right is required to change a branch ACL. It is implicitly held by the owner of a branch.

4.4 Disconnected Operation

Virtual disks can be used while disconnected from the network, as long as the entire disk has been copied onto the disconnected machine. Thus, for a virtualization aware file system to be as widely useful as a virtual disk, it must also gracefully tolerate network disconnection.

Research in network file systems has identified a number of features required for successful disconnected operation [16, 15, 12]. Many of these features apply to Ventana in the same way as conventional network file systems. Ventana, for example, can cache file system data and metadata on disk, which allows it to store enough data and metadata to last the period of disconnection. Our prototype caches entire files, not individual blocks, to avoid the need to allow reading only part of a file during disconnection, which is surprising at best. Ventana can also buffer changes to files and directories and write them back upon reconnection. Some details of these features of Ventana are included in the description of our prototype (see Section 5).

Handling conflicts, that is, different changes to the same files, is a thorny issue in a design for disconnected operation. Fortunately, earlier studies of disconnection have shown conflicts to be rare in practice [16]. In Ventana conflicts may be even rarer, because they cannot occur in private branches. Therefore, Ventana does not try to intelligently handle conflicts. Instead, changes by disconnected clients are committed at the time of reconnection, regardless of whether those files have been changed in the meantime by other clients. If manual merging is needed in shared branches, it is still possible based on old versions of the files. To make it easy to identify file versions just before reconnection, Ventana creates a new branch just before it commits the disconnected changes.

5 Prototype

To show that our ideas can be realized in a practical and efficient way, we developed a simple prototype of Ventana. This section describes the prototype's design and use.

The Ventana prototype is written in C. We developed it under Debian GNU/Linux "unstable" on *x*86 PCs running Linux 2.6.*x*, using VMware Workstation 5.0 as VMM. The servers in the prototype run as Linux user processes and communicate over TCP using the GNU C library implementation of ONC RPC [26].

Figure 3 outlines Ventana's structure, which is described in more detail below.

5.1 Server Architecture

A conventional file system operates on what Unix calls a block device, that is, an array of numbered blocks. Our prototype is instead layered on top of an *object store* [10, 7]. An object store contains *objects*, sparse arrays of bytes numbered from zero to infinity, similar to files. In the Ventana prototype, objects are immutable.

The object store consists of one or more *object servers*, each of which stores some of the file system's objects and provides a network interface for storing new objects and retrieving the contents of old ones. Objects are identified by randomly selected 128-bit integers called *object numbers*. Object numbers are generated randomly to allow them to be chosen without coordination between hosts. Collisions are unlikely as long as significantly fewer than 2^{64} have been generated, according to the "birthday paradox" [25].

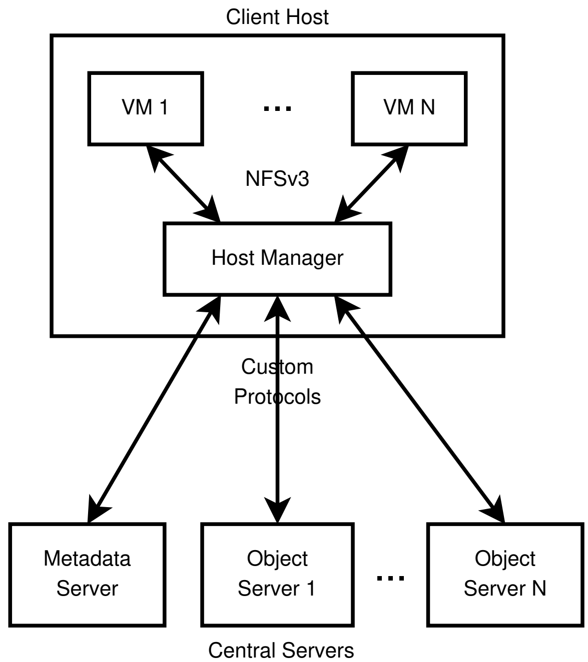

FIGURE 3: Structure of Ventana. Each machine whose VMs use Ventana runs a host manager. The host manager talks to the VMs over NFSv3 and to Ventana's centralized metadata and object servers over a custom protocol.

Each version of a file's data or metadata is stored as an object. When a file's data or metadata is changed, the new version is stored as a new object under a new object number. The old object is not changed and it may still be accessed under its original object number. However, this does not mean that every intermediate change takes up space in the object store, because client hosts (that is, machines that run Ventana clients in VMs) consolidate changes before they commit a new object.

As in an ordinary file system, each file is identified by an inode number, which is again a 128-bit, randomly selected integer. Each file may have many versions across many branches. When a client host needs to know what object stores the latest version of a file in a particular branch, it consults the *version database* by contacting the *metadata server*. The metadata server maintains the version database that tracks the versions of each file, the *branch database* that tracks the file system's branch structure, the database that associates branch names and numbers, and the database that stores VM configurations.

5.2 Client Architecture

The *host manager* is the client-side part of the Ventana prototype. One copy of the host manager runs on each platform and services any number of local client VMs. Our prototype does not encapsulate the host manager itself in a VM.

For compatibility with existing clients, the host manager includes a NFSv3 [2] server for clients to use for file access. NFSv3 is both easy to implement and widely supported, even on Windows (with Microsoft's free Services for Unix).

The host manager maintains in-memory and on-disk caches of file system data and metadata. Objects may be cached indefinitely because they are immutable. Objects are cached in their entirety to simplify implementing the prototype and to enable disconnected operation (see Section 5.2.3). Records in the version and branch databases are also immutable, except for the ACLs they include, which change rarely. In a shared branch, records added to the version database to announce a new file version are a cache consistency issue, so the host manager checks the version database for new versions on each access (except when disconnected). In a private branch, normally only one client modifies the branch at a time, so that client's host manager can cache data in the branch for a long time (or until the client VM is migrated to another host), although other hosts should check for updates more often.

The host manager also buffers file writes. When a client writes a file, the host manager writes the modified file to the local disk. Further changes to the file are also written to the same file. If the client requests that writes be committed to stable storage, e.g. to allow the guest to flush its buffer cache or to honor an `fsync` call, then the host manager commits the modified files to the local disk. Commitment does not perform a round trip on a physical network.

5.2.1 Branch Snapshots

After some amount of time, the host manager takes a snapshot of outstanding changes within a branch. Users can also explicitly create (and optionally name) branch snapshots. A snapshot of a branch is created simply by forking of the branch, which has the desired effect because forking a branch copies its content. In fact, copying occurs on a copy-on-write basis, so that the first write to any of the files in the snapshot creates and modifies a new copy of the file. Creating a branch also inserts a record in the branch database.

After it takes a snapshot, the host manager uploads the objects it contains into the object store. Then, it sends records for the new file versions to a metadata server, which commits them to the version database in a single atomic transaction. The changes are now visible to other clients.

The host manager assumes that private branch data is relatively uninteresting to clients on other hosts, so it takes snapshots in private branches relatively rarely (every 5 minutes). On the other hand, other users may be actively using files in shared branches, so the host manager takes snapshots often (every 3 seconds).

Because branch snapshots are actually branches themselves, older versions of files can be viewed using regular file commands by first adding the snapshot branch to the view in use. Branches created as snapshots are by default read-only, to reduce the chance of later confusion if a file's "older version" actually turns out to have been modified.

5.2.2 Views and VMs

Multiple branches can be composed into a view. Ventana describes a view with a simple text format that resembles a Unix `fstab`, e.g.:

```
debian:/         /       shared,ro
home-dirs:/      /home   shared
bob-version:/    /proj   private
```

Each line describes a mapping between a branch, or a subset of a branch, and a directory within the view. We say that each branch is *attached* to its directory in the view.[1]

A VM comprises a view, plus configuration parameters for networking, system boot, and so on. A VM could be described by the view above followed by these additional options:

```
-pxe-kernel debian:/boot/vmlinuz
-ram 64
```

Ventana provides a utility to start a VM based on such a specification. Given the above VM specification, it would set up a network boot environment (using the PXE protocol) to boot the kernel in `/boot/vmlinuz` in the `debian` branch, then launch VMware Workstation for the user to allow the user to interact with the VM.

VM Snapshots Ventana supports snapshots of VMs just as it does snapshots of branches.[2] A snapshot of a VM is a snapshot of each branch in the VM's view combined with a snapshot of the VM's runtime state (RAM, device state, ...). To create a snapshot, Ventana snapshots the branches included in the VM, copies the runtime state file written by Workstation into Ventana as an unnamed file, and saves a description of the view and a pointer to the suspend file.

Later, another Ventana utility may be used to resume from the snapshot. When a VM snapshot is resumed, private branches have the contents that they did when the snapshot was taken, and shared branches are up-to-date. Ventana also allows resuming with a "frozen" copy of shared branches as of the time of the snapshot. Snapshots can be resumed any number of times, so resuming forks each private branch in the VM for repeatability.

[1] We use "attach" instead of "mount" because mounts are implemented inside an OS, whereas the guest OS that uses Ventana does not implement and is not aware of the view's composition.

[2] VMware Workstation has its own snapshot capability. Ventana's snapshot mechanism demonstrates VM snapshots might be integrated into a VAFS.

5.2.3 Disconnected Operation

The host manager supports disconnected operation, that is, file access is allowed even without connectivity to the metadata and object server. Of course, access is degraded during disconnection: only cached files may be read, and changes in shared branches by clients on the other hosts are not visible. Write access is unimpeded. Disconnected operation is implemented in the host manager, not in clients, so all clients support disconnected operation.

We designed the prototype with disconnected operation in mind. Caching eliminates the need to consult the metadata and object servers for most operations, and on-disk caching allows for a large enough cache to be useful for extended disconnection. Whole-object caching avoids surprising semantics that would allow only part of a file to be read. Write buffering allows writing back changes to be delayed until reconnection.

We have not implemented user-configurable "hoarding" policies in the prototype. Implementing them as described by Kistler et al. [16] would be a logical extension.

6 Usage Scenario

This section presents a scenario for use of Ventana and shows how, in this setting, Ventana offers a better solution than both virtual disks and network file systems.

6.1 Scenario

We set our scene at Widgamatic, a manufacturer and distributor of widgets.

6.1.1 Alice the Administrator

Alice is Widgamatic's system administrator in charge of virtual machines. Software used at Widgamatic has diverse requirements, and Widgamatic's employees have widely varying preferences. Alice wants to accommodate everyone as much as she can, so she supports various operating systems: Debian, Ubuntu, Red Hat, and SUSE distributions of GNU/Linux, plus Windows XP and Windows Server 2003. For each of these, Alice creates a shared branch and installs the base OS and some commonly used applications. She sets the branch ACLs to allow any user to read, but not write, these branches.

Alice creates `common`, a second shared branch, to hold files that should be uniform company-wide, such as

```
                         ubuntu:/                 /                  shared,ro
                      home-dirs:/                 /home              shared
                           none                   /tmp               non-persistent
12ff2fd27656c7c7e07c5ea1e2da367f:/var             /var               private
                       cad-soft:/                 /opt/cad-soft      shared,ro
                         common:/etc/resolv.conf  /etc/resolv.conf   shared,ro
                         common:/etc/passwd       /etc/passwd        shared,ro
8368e293a23163f6d2b2c27aad2b6640:/etc/hostname    /etc/hostname      private
b6236341bd1014777c1a54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

FIGURE 4: Partial specification of the view for Bob's basic VM.

```
                    carl-debian:/                 /                  private
                      home-dirs:/                 /home              shared
                           none                   /tmp               non-persistent
                         common:/etc/resolv.conf  /etc/resolv.conf   shared,ro
                         common:/etc/passwd       /etc/passwd        shared,ro
b6236341bd1014777c1a54a8d2d03f7c:/etc/ssh/host_key /etc/ssh/host_key unversioned
```

FIGURE 5: Partial specification of the view for Carl's custom VM.

`/etc/hosts` and `/etc/resolv.conf`. Again, she sets branch ACLs to grant other users read-only access.

Alice also creates a shared branch for user home directories, called `home-dirs`, and adds a directory for each Widgamatic user in the root of this branch. Alice sets the branch ACL to allow any user to read or write the branch, and server file ACLs so that, by default, each user can read or write only his (or her) home directory. Users can of course modify server file ACLs in their home directories as needed.

6.1.2 Bob's Basic VM

Bob is a Widgamatic user with basic needs. Bob uses a utility written by Alice to create a Linux-based VM primarily from shared branches. Figure 4 shows part of the specification written by this utility.

The root of Bob's VM is attached to the Ubuntu shared branch created by Alice. This branch's ACL prevents Bob modifying files in the branch (it is attached read-only besides). The Linux file system is well suited for this situation, because its top-level hierarchies segregate files based on whether they can be attached read-only during normal system operation. The `/usr` tree is an example of a hierarchy that normally need not be modifiable.

The `/home` and `/tmp` trees are the most prominent examples of hierarchies that must be writable, so Bob's VM attaches a writable shared branch and a non-persistent branch, respectively, at these points. Keyword `none` in place of a branch name in `/tmp`'s entry causes an initially empty branch to be attached.

The filename/var hierarchy must be writable and persistent, and it cannot be shared between machines. Thus, Alice's utility handles `/var` by creating a fork of the Ubuntu branch, then attaching the forked branch's `/var` privately in the VM. The utility does not give the forked branch a name, so the VM specification gives the 128-bit branch identifier as 32 hexadecimal digits.

Bob needs to use the company's CAD software to design widgets, so the CAD software distribution is attached into his VM.

Most of the VM's configuration files in `/etc` receive their contents from the Ubuntu branch attached at the VM's root. Some, such as `/etc/resolv.conf` and `/etc/passwd` shown here, are attached from Alice's "common files" branch. This allows Alice to update a file in just that branch and have the changes automatically reflected in every VM. A few, such as `/etc/hostname` shown here, are attached from private branches to allow their contents to be customized for the particular VM. Finally, data that should not be versioned at all, such as the private host key used to identify an SSH server, is attached from an unversioned branch. The latter two branches are, like the `/var` branch, unnamed.

Bob's VM, and VMs created in similar ways, would automatically receive the benefits of changes and updates made by Alice as soon as she made them. They would also see changes made by other users to their home directories as soon as they occur.

6.1.3 Carl's Custom VM

Carl wants more control over his VM. He prefers Debian, which is available as a branch maintained by Alice, so he can base his VM upon Alice's. Carl forks a private branch from Alice's Debian branch and names the new branch `carl-debian`.

Carl integrates his branch into a VM of his own, using a specification that in part looks like Figure 5. Carl could write this specification by hand, or he might choose to start from one, like Bob's, generated by Alice's utility. Using a private branch as root directory means that Carl need not attach private branches on top of `/var` or `/etc/hostname`, making Carl's specification shorter than Bob's.

Even though Carl's base operating system is private, Carl's VM still attaches many of the same shared branches that Bob's VM does. Shared home directories and common configuration files ease Carl's administrative burden just as they do Bob's. He could choose to keep private copies of these files, but to little obvious benefit.

Carl bears more of the burden of his own system administration, because Alice's changes to shared branches do not automatically propagate to his private branch. Carl could use Ventana to observe how the parent `debian` branch has changed since the fork, or Alice could monitor forked branches to ensure that important patches are applied in a timely fashion.

6.1.4 Alice in Action

One morning Alice reads a bulletin announcing a critical security vulnerability in Mozilla Firefox. Alice must do her best to make sure that the vulnerable version is properly patched in every VM. In a VM environment based on virtual disks, this would be a daunting task. Ventana, however, reduces the magnitude of the problem considerably.

First, Alice patches the branches that she maintains. This immediately fixes VMs that use her shared branches, such as Bob's VM.

Second, Alice can take steps to fix others' VMs as well. Ventana puts a spectrum of options at her disposal. Alice could do nothing and assume that Bob and Carl will act responsibly. She could scan VMs for the insecure binary and email their owners (she can even check up on them later). She could patch the insecure binaries herself. Finally, she has many options for denying access to copies of the insecure binary: use a server file ACL to deny reading or executing it, use a Ventana version ACL to prevent reading it even as the older version of a file, use a branch ACL to deny any access to the branch that contains it (perhaps appropriate for long-unused branches), and so on. Alice can take these steps for any file stored in Ventana, whether contained in a VM that is powered on or off or suspended, or even if it is not in any VM or view at all.

Third, once the immediate problem is solved, Alice can work to prevent its future recurrence. She can configure a malware scanner to examine each new version of a file added to Ventana as to whether it is the vulnerable program and, if so, alert Alice or its owner (or take some other action). Thus, Alice has reasonable assurance that if this particular problem recurs, it can be quickly detected and fixed.

6.2 Benefits for Widgamatic

We now consider how Alice, Bob, Carl, and everyone else at Widgamatic benefit from using Ventana instead of virtual disks. We use virtual disks as our main basis of comparison because Ventana's advantages over conventional distributed file systems are more straightforward: they are the versioning, isolation, and encapsulation features that we intentionally added to it and have already described in detail.

6.2.1 Central Storage

It's easy for Bob or Carl to create virtual machines. When virtual disks are used, it's also easy for Bob or Carl to copy them to a physical machine or a removable medium, then lose or forget about the machine or the medium. If the virtual machine is rediscovered later, it may be missing fixes for important security problems that have arisen in the meantime.

Ventana's central storage makes it more difficult to lose or entirely forget about VMs, heading off the problem before it occurs. Other dedicated VM storage systems also yield this benefit [30, 31].

6.2.2 Looking Inside Storage

Alice's administration tasks can benefit from "looking inside" storage. Consider backup. Bob and Carl want the ability to restore old versions of files, but Alice can easily back up virtual disks only as a collection of disk blocks. Disk blocks are opaque, making it hard for Bob or Carl even to determine which version of a virtual disk contains the file to restore. Doing partial backups of virtual disks, e.g. to exclude blocks from deleted temporary files or paging files, is also difficult.

File-based backup, partial backup, and related features can be implemented for virtual disks, but only by mounting the virtual disk or writing code to do the equivalent. In any case, software must have an intimate knowledge of file system structures and must be maintained as those structures change among operating systems and over time. Mounting an untrusted disk can itself be a security hole [24].

On the other hand, Ventana's organization into files and directories gives it a higher level of structure that makes it easy to look inside a Ventana file system. Thus, file-based backup and restore requires no special effort in Ventana.

(Of course, in Ventana it is natural to use versioning to access file "backups" and ensure access by backing up Ventana servers' storage.)

6.2.3 Sharing

Sharing is an important feature of storage systems. Bob and Carl might wish to collaborate on a project, or Carl might ask Alice to install some software in his VM for him. Virtual disks make sharing difficult. Consider how Alice could access Carl's files if they were stored on a virtual disk. If Carl's VM were powered on or suspended, modifying his file system would risk the guest OS's integrity, because the interaction with the guest's data and metadata caches would be unpredictable. Even reading Carl's file system would be unreliable while it was changing, e.g. consider the race condition if a block from a deleted directory was reused to store an ordinary file block.

On the other hand, Ventana gives Alice full read and write access to virtual machines, even those that are online or suspended. Alice can examine or modify Carl's files, whether the VM or VMs that use them are running, suspended, or powered off, and Bob and Carl can work together on their project without introducing any special new risks.

6.2.4 Security

If Widgamatic's VMs were stored in virtual disks, Alice would have a hard time scanning them for malware. She could request that users run a malware scanner inside each of their VMs, but it would be difficult for her to enforce this rule or ensure that the scanner was kept up-to-date. Even if Bob and Carl carefully followed her instructions, VMs powered on after being off for a long time would be susceptible to vulnerabilities discovered in the meantime until they were updated.

Ventana allows Alice to deploy a scanner that can examine each new version of a file in selected branches, or in all branches. Conversely, when new vulnerabilities are found, it can scan old versions of files as well as current versions (as time is available). If malware is detected in Bob's branch, the scanner could alert Bob (or Alice), delete the file, change the file's permission, or remove the virus from the file. (Even in a private branch, files may be externally modified, although it takes longer for changes to propagate in each direction.)

Ventana provides another important benefit for scanners: the scanner operates in a protection domain separate from any guest operating system. When virtual disks store VMs, scanners normally run as part of the guest operating system because, as we've seen, even read-only access to active virtual disks has pitfalls. But this allows a "root kit" to subvert the guest operating system and the malware scanner in a single step. If Alice runs her scanner in a different VM, it must be compromised separately. Alice could even configure the scanner to run in non-persistent mode, so rebooting it would temporarily relieve any compromise, although of course not the underlying vulnerability.

A host-based intrusion detection system could use a "lie detector" test that compares the file system seen by programs running inside the VM against the file system in Ventana to detect root kits, as in LiveWire [8].

6.2.5 Access to Multiple Versions

Suppose Bob wants to look at the history of a document he's been working on for some time. He wants to retrieve and compare all its earlier versions. One option for Bob is to read the old versions directly from older versions of the virtual disk, but this requires accurate interpretation of the file system, which is difficult to maintain over time. A more likely alternative for Bob is to resume or power on each older version of the VM, then use the guest OS to copy the file in that old VM somewhere convenient. Unfortunately, this can take a lot of time, especially if the VM has to boot, and every older version takes extra effort.

With Ventana, Bob can attach all the older versions of his branch directly to his view. After that, the different versions can be accessed with normal file commands: `diff` to view differences between versions, `grep` to search the history, and so on. Bob can also recover older versions simply by copying them into the his working branch.

7 Future Work

Ventana demonstrates the principles behind a VAFS, but many important issues remain to be explored, such as scalability and performance. We have measured Ventana's performance to be competitive with other user-level NFS servers in most cases with simple branching. However, deep chains of branches seem to introduce the need for compromise between storage efficiency and file lookup performance.

Storage reuse is another area for further work. The Ventana prototype does not have any mechanism for deleting data. We have not yet found a way to efficiently support both creation of branches and the determination that an object is no longer in use in any branch.

8 Related Work

Parallax [31] demonstrates that virtual disks can be stored centrally with very high scalability. Parallax allows virtual disks to be efficiently used and modified in a copy-on-write fashion by many users. Unlike Ventana, it does not allow cooperative sharing among users, nor does it enhance the transparency or improve the granularity of virtual disks.

VMware ESX Server includes the VMFS file system, which is designed for storing large files such as virtual disks [30]. VMFS allows for snapshots and copy-on-write sharing, but not the other features of a virtualization aware file system.

Live migration of virtual machines [4] requires the VM's storage to be available on the network. Ventana, as a distributed file system particularly suited to VM storage, provides a reasonable approach.

Whitaker et al. [33, 34] used whole-system versioning to mechanically discover the origin of a problem by doing binary search through the history of a system. They note the "semantic gap" in trying to relate changes to a virtual disk with higher-level actions. We believe that a VAFS, in which changes to files and directories may be observed directly, could help to reduce this semantic gap.

The Ventana prototype of course has much in common with other file systems. Object stores are an increasingly common way to structure file systems [10, 7, 28]. Objects in Ventana are immutable, which is unusual among object stores, although in this respect Ventana resembles the Cedar file system and, more recently, EMC's Centera system [11, 6]. PVFS2, a network file system for high-bandwidth parallel file I/O, is another file system that uses Berkeley DB databases to store file system metadata [21].

Many versioning file systems exist, in research systems such as Cedar, Elephant, and S4, and in production systems such as WAFL (used by Network Appliance filers) and VMS [11, 23, 27, 14, 18]. A versioning file system on top of a virtual disk allows old versions to be easily accessed inside the VM, but does not address the other downsides of virtual disks. None of these systems supports the tree-structured versions necessary to properly handle the natural evolution of virtual machines. The version retention policies introduced in Elephant might be usefully applied to Ventana.

Online file archives, such as Venti, also support accessing old versions of files, but again only linear versioning is supported [22].

Ventana's tree-structured version model is related to the model used in revision control systems, such as CVS [3]. A version created by merging versions from different branches has more than one parent, so versions in revision control systems are actually structured as directed acyclic graphs. Revision control systems would generally not be good "back end" storage for Ventana or another VAFS because they typically store only a single "latest" version of a file for efficient retrieval. Retrieving other versions, including the latest version of files in branches other than the "main branch," requires application of patches [29]. Files marked "binary," however, often include each revision in full, without using patches, so use of "binary" files might be an acceptable choice.

Vesta [13] is a software configuration management system whose primary file access interface is over NFS, like Ventana. Dependency tracking in Vesta allows for precise, high-performance, repeatable builds. Similar tracking by a VAFS might enable better understanding of which files and versions should be retained over the long term.

We proposed extending a distributed file system, which already supports fine-grained sharing, by adding versioning that supports virtual machines. An alternative is to allow virtual disks, which already support VM-style versioning, to support sharing by adding a locking layer, as can be done for physical disks [19, 1]. This approach requires committing to a particular on-disk format, which makes changes and extensions more difficult. It also either requires each client to understand the disk format, which is a compatibility issue, or use of a network proxy that does understand the format. In the latter case the proxy is equivalent to Ventana's host manager, and storage underlying it is really an implementation detail.

A "union" or "overlay" file system [20, 17] can stack a writable file system above layers of read-only file systems. If the top layer is the current branch and lower layers are the branches that it was forked from, something like tree versioning can be obtained. The effect is imperfect because changes to lower layers can "show through" to the top. Symbolic link farms can also stack layers of directories, often for multi-architecture software builds [5], but link farms are not transparent to the user or software.

9 Conclusion

Ventana is a *virtualization aware* distributed file system that provides the powerful versioning, security, and mobility properties of virtual disks, while overcoming their coarse-grained versioning and their opacity that frustrates cooperative sharing. This allows Ventana to support the rich usage models facilitated by virtual machines, while avoiding the security pitfalls, management difficulties, and usability problems that virtual disks suffer from.

We believe that virtualization aware file systems have an important role to play in the evolution of virtual machines from their physical machine inspired roots, toward being a more lightweight, flexible, and general-purpose mechanism for organizing systems.

Acknowledgements

This work was supported in part by the National Science Foundation under award 0121481 and by TRUST (Team for Research in Ubiquitous Secure Technology), which also receives support from the National Science Foundation under award CCF-0424422. We would like to thank Carl Waldspurger, Tim Mann, Jim Chow, Paul Twohey, Junfeng Yang, Joe Little, our shepherd Steve Hand, and the anonymous reviewers for their comments.

References

[1] R. C. Burns. *Data Management in a Distributed File System for Storage Area Networks*. PhD thesis, University of California Santa Cruz, March 2000.

[2] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995.

[3] P. Cederqvist et al. *Version Management with CVS*, 2005.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd Symposium on Network Systems Design and Implementation*. USENIX, 2005.

[5] P. Eggert. Multi-architecture builds using GNU `make`. http://make.paulandlesley.org/multi-arch.html, August 2000.

[6] EMC Corporation. EMC Centera content addressed storage system. http://www.emc.com/products/systems/centera.jsp, October 2005.

[7] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, Sardinia, Italy, July 2005.

[8] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[9] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *10th Workshop on Hot Topics in Operating Systems*, May 2005.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 272–284, New York, NY, USA, 1997. ACM Press.

[11] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Commununications of the ACM*, 31(3):288–298, 1988.

[12] J. S. Heidemann, T. W. Page, Jr., R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Workshop on the Management of Replicated Data*, pages 2–5, 1992.

[13] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Research Report 168, Compaq Systems Research Center, March 2001.

[14] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Technical report, Network Appliance, 1995.

[15] L. Huston and P. Honeyman. Disconnected operation for AFS. In *First Usenix Symposium on Mobile and Location-Independent Computing*, pages 1–10, August 1994.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[17] M. Klotzbuecher. mini_fo: The mini fanout overlay file system. http://www.denx.de/twiki/bin/view/Know/MiniFOHome, October 2005.

[18] K. McCoy. *VMS file system internals*. Digital Press, Newton, MA, USA, 1990.

[19] T. McGregor and J. Cleary. A block-based network file system. In *21st Australasian Computer Science Conference*, volume 20 of *Australian Computer Science Communications*, pages 133–144, Perth, February 1998. Springer.

[20] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Summer UKUUG Conference*, pages 1–9, London, July 1990.

[21] PVFS2: Parallel virtual file system 2. http://www.pvfs.org/pvfs2, October 2005.

[22] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[23] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *17th ACM Symposium on Operating Systems Principles*, pages 110–123, New York, NY, USA, 1999. ACM Press.

[24] C. Sar, P. Twohey, J. Yang, C. Cadar, and D. Engler. Discovering malicious disks with symbolic execution. In *IEEE Symposium on Security and Privacy*, May 2006.

[25] B. Schneier. *Applied Cryptography*. Wiley, 2nd edition, 1996.

[26] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, Aug. 1995.

[27] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *4th USENIX Symposium on Operating System Design and Implementation*, pages 165–180, 2000.

[28] C. F. Systems. Lustre. http://lustre.org/.

[29] W. F. Tichy. RCS—a system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.

[30] VMware ESX Server. http://www.vmware.com/products/esx, October 2005.

[31] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. In *10th Hot Topics in Operating Systems*. USENIX, May 2005.

[32] A. Watson, P. Benn, A. G. Yoder, and H. T. Sun. Multiprotocol data access: NFS, CIFS, and HTTP. Technical report, Network Appliance, 2005.

[33] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th Symposium on Operating Systems Design and Implementation*, December 2004.

[34] A. Whitaker, R. S. Cox, and S. D. Gribble. Using time travel to diagnose computer problems. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.