USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment

Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken,
John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, Roger P. Wattenhofer

Microsoft Research, Redmond, WA 98052

{adya, bolosky, mcastro, gcermak, rchaiken, johndo, howell, lorch, theimer}@microsoft.com;
wattenhofer@inf.ethz.ch

Abstract

Farsite is a secure, scalable file system that logically functions as a centralized file server but is physically distributed among a set of untrusted computers. Farsite provides file availability and reliability through randomized replicated storage; it ensures the secrecy of file contents with cryptographic techniques; it maintains the integrity of file and directory data with a Byzantine-fault-tolerant protocol; it is designed to be scalable by using a distributed hint mechanism and delegation certificates for pathname translations; and it achieves good performance by locally caching file data, lazily propagating file updates, and varying the duration and granularity of content leases. We report on the design of Farsite and the lessons we have learned by implementing much of that design.

1. Introduction

This paper describes Farsite, a serverless distributed file system that logically functions as a centralized file server but whose physical realization is dispersed among a network of untrusted desktop workstations. Farsite is intended to provide both the benefits of a central file server (a shared namespace, location-transparent access, and reliable data storage) and the benefits of local desktop file systems (low cost, privacy from nosy sysadmins, and resistance to geographically localized faults). Farsite replaces the physical security of a server in a locked room with the virtual security of cryptography, randomized replication, and Byzantine fault-tolerance [8]. Farsite is designed to support typical desktop file-I/O workloads in academic and corporate environments, rather than the high-performance I/O of scientific applications or the large-scale write sharing of database applications. It requires minimal administrative effort to initially configure and practically no central administration to maintain. With a few notable exceptions (such as crash recovery and interaction between multiple Byzantine-fault-tolerant groups), nearly all of the design we describe has been implemented.

Traditionally, file service for workstations has been provided either by a local file system such as FFS [28] or by a remote server-based file system such as NFS [39] or AFS [21]. Server-based file systems provide a shared namespace among users, and they can offer greater file reliability than local file systems because of better maintained, superior quality, and more highly redundant components. Servers also afford greater physical security than personal workstations in offices. However, server-based systems carry a direct cost in equipment, physical plant, and personnel beyond those already sunk into the desktop infrastructure commonly found in modern companies and institutions. A server requires a dedicated administrative staff, upon whose competence its reliability depends [19] and upon whose trustworthiness its security depends [47]. Physically centralized servers are vulnerable to geographically localized faults, and their store of increasingly sensitive and valuable information makes them attractive, concentrated targets for subversion and data theft, in contrast to the inherent decentralization of desktop workstations.

In designing Farsite, our goal has been to harness the collective resources of loosely coupled, insecure, and unreliable machines to provide logically centralized, secure, and reliable file-storage service. Our system protects and preserves file data and directory metadata primarily through the techniques of cryptography and replication. Since file data is large and opaque to the system, the techniques of encryption, one-way hashing, and raw replication provide means to ensure its privacy, integrity, and durability, respectively. By contrast, directory metadata is relatively small, but it must be comprehensible and revisable directly by the system; therefore, it is maintained by Byzantine-replicated state-machines [8, 36] and specialized cryptographic techniques that permit metadata syntax enforcement without compromising privacy [15]. One of Farsite's key design objectives is to provide the benefits of Byzantine fault-tolerance while avoiding the cost of full Byzantine agreement in the common case, by using signed and dated certificates to cache the authorization granted through Byzantine operations.

Both Farsite's intended workload and its expected machine characteristics are those typically observed on desktop machines in academic and corporate settings. These workloads exhibit high access locality, a low persistent update rate, and a pattern of read/write sharing that is usually sequential and rarely concurrent [22, 48]. The expected machine characteristics include a high fail-stop rate (often just a user turning a machine off for a while) [6] and a low but significant rate [41] of malicious or opportunistic subversion. In our design, analysis, evaluation, and discussion, we focus on this environment, but we note that corporate administrators might choose to supplement Farsite's reliability and security by adding userless machines to the system or even running entirely on machines in locked rooms.

Farsite requires no central administration beyond that needed to initially configure a minimal system and to authenticate new users and machines as they join the system. Administration is mainly an issue of signing certificates: Machine certificates bind machines to their public keys; user certificates bind users to their public keys; and namespace certificates bind namespace roots to their managing machines. Beyond initially signing the namespace certificate and subsequently signing certificates for new machines and users, no effort is required from a central administrator.

There are many directions we could have explored in the Farsite design space that we have chosen not to. Farsite is not a high-speed parallel I/O system such as SGI's XFS [43], and it does not efficiently support large-scale write sharing of files. Farsite is intended to emulate the behavior of a traditional local file system, in particular NTFS [42]; therefore, it introduces no new user-visible semantics, such as an object-model interface, transactional support, versioning [39], user-specifiable file importance, or Coda-like [22] hooks for application-specific conflict resolvers to support concurrent file updates during disconnected operation.

We have implemented most – but not all – of the design described in this paper. The exceptions, which mainly relate to scalability and crash recovery, are itemized in section 6 and identified throughout the text with the term *Farsite design*, indicating a mechanism that we have designed but not yet implemented.

The following section presents a detailed overview of the system. Section 3, the bulk of the paper, describes the mechanisms that provide Farsite's key features. Section 4 describes our prototype implementation. Section 5 presents analytical arguments that show the Farsite design to be scalable, and it presents empirical measurements that show our prototype's performance to be acceptable. We discuss future work in section 6, related work in section 7, and conclusions in section 8.

2. System Overview

This section begins by presenting the assumptions and technology trends that underlie Farsite's design. Then, it introduces the design by explaining the fundamental concept of namespace roots, describing our trust model and certification mechanisms, outlining the system architecture, and describing a few necessary semantic differences between Farsite and a local file system.

2.1 Design Assumptions

Most of our design assumptions stem from the fact that Farsite is intended to run on the desktop workstations of a large corporation or university. Thus, we assume a maximum scale of $\sim 10^5$ machines, none of which are dedicated servers, and all of which are interconnected by a high-bandwidth, low-latency network whose topology can be ignored. Machine availability is assumed to be lower than that of dedicated servers but higher than that of hosts on the Internet; specifically, we expect the majority of machines to be up and accessible for the majority of the time. We assume machine downtimes to be generally uncorrelated and permanent machine failures to be mostly independent. Although Farsite tolerates large-scale read-only sharing and small-scale read/write sharing, we assume that no files are both read by many users and also frequently updated by at least one user. Empirical data [6, 48] corroborates this set of assumptions.

We assume that a small but significant fraction of users will maliciously attempt to destroy or corrupt file data or metadata, a reasonable assumption for our target environment but an unreasonably optimistic one on the open Internet. We assume that a large fraction of users may independently and opportunistically attempt to read file data or metadata to which they have not been granted access. Each machine is assumed to be under the control of its immediate user, and a party of malicious users can subvert only machines owned by members of the party.

We assume that the local machine of each user can be trusted to perform correctly for that user, and no user-sensitive data persists beyond user logoff or system reboot. This latter assumption is known to be false without a technique such as crypto-paging [50], which is not employed by any commodity operating system including Windows, the platform for our prototype.

2.2 Enabling Technology Trends

Two technology trends are fundamental in rendering Farsite's design practical: a general increase in unused disk capacity and a decrease in the computational cost of cryptographic operations relative to I/O.

A very large fraction of the disk space on desktop workstations is unused, and disk capacity is increasing at a faster rate than disk usage. In 1998, measurements of 4800 desktop workstations at Microsoft [13] showed that 49% of disk space was unused. In 1999 and 2000, we measured the unused portions to be 50% and 58%.

As computation speed has increased with Moore's Law, the cost of cryptographic operations has fallen relative to the cost of the I/O operations that are the mainstay of a file system. We measured a modern workstation and found a symmetric encryption bandwidth of 72 MB/s and a one-way hashing bandwidth of 53 MB/s, both of which exceed the disk's sequential-I/O bandwidth of 32 MB/s. Encrypting or decrypting 32 KB of data adds roughly one millisecond of latency to the file-read/write path. Computing an RSA signature with a 1024-bit key takes 6.5 milliseconds, which is less than one rotation time for a 7200-RPM disk.

The large amount of unused disk capacity enables the use of replication for reliability, and the relatively low cost of strong cryptography enables distributed security.

2.3 Namespace Roots

The primary construct established by a file system is a hierarchical directory namespace, which is the logical repository for files. Since a namespace hierarchy is a tree, it has to have a root. Rather than mandating a single namespace root for any given collection of machines that form a Farsite system, we have chosen to allow the flexibility of multiple roots, each of which can be regarded as the name of a virtual file server that is collaboratively created by the participating machines. An administrator creates a namespace root simply by specifying a unique (for that administrator) root name and designating a set of machines to manage the root. These machines will form a Byzantine-fault-tolerant group (see subsection 2.5), so it is not crucial to select a specially protected set of machines to manage the root.

2.4 Trust and Certification

The security of any distributed system is essentially an issue of managing trust. Users need to trust the authority of machines that offer to present data and metadata. Machines need to trust the validity of requests from remote users to modify file contents or regions of the namespace. Security components that rely on redundancy need to trust that an apparently distinct set of machines is truly distinct and not a single malicious machine pretending to be many, a potentially devastating attack known as a Sybil attack [11].

Farsite manages trust using public-key-cryptographic certificates. A certificate is a semantically meaningful data structure that has been signed using a private key. The principal types of certificates are namespace certificates, user certificates, and machine certificates. A *namespace certificate* associates the root of a file-system namespace with a set of machines that manage the root metadata. A *user certificate* associates a user with his personal public key, so that the user identity can be validated for access control. A *machine certificate* associates a machine with its own public key, which is used for establishing the validity of the machine as a physically unique resource.

Trust is bootstrapped by fiat: Machines are instructed to accept the authorization of any certificate that can be validated with one or more particular public keys. The corresponding private keys are known as *certification authorities* (*CAs*). Consider a company with a central certification department, which generates a public/private key pair to function as a CA. Every employee generates a public/private key pair, and the CA signs a certificate associating the employee's username with the public key. For a small set of machines, the CA generates public/private key pairs and embeds the public keys in a namespace certificate that designates these machines as managers of its namespace root. The CA then performs the following four operations for every machine in the company: (1) Have the machine generate a public/private key pair; (2) sign a certificate associating the machine with the public key; (3) install the machine certificate and root namespace certificate on the machine; and (4) instruct the machine to accept any certificate signed by the CA.

Farsite's certification model is more general than the above paragraph suggests, because machine certificates are not signed directly by CAs but rather are signed by users whose certificates designate them as authorized to certify machines, akin to Windows' model for domain administration. This generalization allows a company to separate the responsibilities of authenticating users and machines, e.g., among HR and IT departments. A single machine can participate in multiple CA domains, but responsibility granted by one CA is only delegated to other machines authorized by that same CA.

A machine's private key is stored on the machine itself. In the Farsite design, each user private key is encrypted with a symmetric key derived from the user's password and then stored in a globally-readable directory in Farsite, where it can be accessed upon login. CA private keys should be kept offline, because the entire security of a Farsite installation rests on their secrecy.

User or machine keys can be revoked by the issuing CA, using the standard techniques [29]: Signed revocation lists are periodically posted in a prominent, highly replicated location. All certificates include an expiration date, which allows revocation lists to be garbage collected.

2.5 System Architecture

We now present a high-level overview of the Farsite architecture, beginning with a simplified system and subsequently introducing some of Farsite's key aspects.

2.5.1 Basic System

Every machine in Farsite may perform three roles: It is a *client*, a member of a *directory group*, and a *file host*, but we initially ignore the latter of these. A client is a machine that directly interacts with a user. A directory group is a set of machines that collectively manage file information using a Byzantine-fault-tolerant protocol [8]: Every member of the group stores a replica of the information, and as the group receives client requests, each member processes these requests deterministically, updates its replica, and sends replies to the client. The Byzantine protocol guarantees data consistency as long as fewer than a third of the machines misbehave.

Consider a system that includes several clients and one directory group. For the moment, imagine that the directory group manages all of the file-system data and metadata, storing it redundantly on all machines in the group. When a client wishes to read a file, it sends a message to the directory group, which replies with the contents of the requested file. If the client updates the file, it sends the update to the directory group. If another client tries to open the file while the first client has it open, the directory group evaluates the specified sharing semantics requested by the two clients to determine whether to grant the second client access.

2.5.2 System Enhancements

The above-described system provides reliability and data integrity through Byzantine replication. However, it may have performance problems since all file-system requests involve remote Byzantine operations; it does not provide data privacy from users who have physical access to the directory group members; it does not have the means to enforce user access control; it consumes a large amount of storage space since files are replicated on every directory group member, and it does not scale.

Farsite enhances the basic system in several ways. It adds local caching of file content on the client to improve read performance. Farsite's directory groups issue leases on files to clients, granting them access to the files for a specified period of time, so a client with an active lease and a cached copy of a file can perform operations entirely locally. Farsite delays pushing updates to the directory group, because most file writes are deleted or overwritten shortly after they occur [4, 48]. To protect user privacy and provide read-access control, clients encrypt written file data with the public keys of all authorized readers; and the directory group enforces write-access control by cryptographically validating requests from users before accepting updates. Since Byzantine groups fail to make progress if a third or more of their members fail, directory groups require a high replication factor (e.g., 7 or 10) [8]. However, since the vast majority of file-system data is opaque file content, we can offload the storage of this content onto a smaller number of other machines ("file hosts"), using a level of indirection. By keeping a cryptographically secure hash of the content on the directory group, putative file content returned from a file host can be validated by the directory group, thereby preventing the file host from corrupting the data. Therefore, Farsite can tolerate the failure of all but one machine in a set of file hosts, thus allowing a far lower replication factor (e.g., 3 or 4) [6] for the lion's share of file-system data.

As machines and users join a Farsite system, the volume of file-system metadata will grow. At some point, the storage and/or operation load will overwhelm the directory group that manages the namespace. The Farsite design addresses this problem by allowing the directory group to delegate a portion of its namespace to another directory group. Specifically, the group randomly selects, from all of the machines it knows about, a set of machines that it then instructs to form a new directory group. The first group collectively signs a new namespace certificate delegating authority over a portion of its namespace to the newly formed group. This group can further delegate a portion of its region of the namespace. Because each file-system namespace forms a hierarchical tree, each directory group can delegate any sub-subtree of the subtree that it manages without involving any other extant directory groups.

In this enhanced system, when a client wishes to read a file, it sends a message to the directory group that manages the file's metadata. This group can prove to the client that it has authority over this directory by presenting a namespace certificate signed by its parent group, another certificate signed by its parent's parent, and so on up to the root namespace certificate signed by a CA that the client regards as authoritative. The group replies with these namespace certificates, a lease on the file, a one-way hash of the file contents, and a list of file hosts that store encrypted replicas of the file. The client retrieves a replica from a file host and validates its contents using the one-way hash. If the user on the client machine has read access to the file, then he can use his private key to decrypt the file. If the client updates the file, then, after a delay, it sends an updated hash to the directory group. The directory group verifies the user's permission to write to the file, and it instructs the file hosts to retrieve copies of the new data from the client. If another client tries to open the file while the first client has an active lease, the directory group may need to contact the first client to see whether the file is still in use, and the group may recall the lease to satisfy the new request.

2.6 Semantic Differences from NTFS

Despite our goal of emulating a local NTFS file system as closely as possible, performance or behavioral issues have occasionally motivated semantics in Farsite that diverge from those of NTFS.

If many (e.g., more than a thousand) clients hold a file open concurrently, the load of consistency management – via querying and possibly recalling leases – becomes excessive, reducing system performance. To prevent this, the Farsite design places a hard limit on the number of clients that can have a file open for concurrent writing and a soft limit on the number that can have a file open for concurrent reading. Additional attempts to open the file for writing will fail with a sharing violation. Additional attempts to open the file for reading will not receive a handle to the file; instead, they will receive a handle to a snapshot of the file, taken at the time of the open request. Because it is only a snapshot, it will not change to reflect updates by remote writers, and so it may become stale. Also, an open snapshot handle will not prevent another client from opening the file for exclusive access, as a real file handle would. An application can query the Farsite client to find out whether it has a snapshot handle or a true file handle, but this is not part of NTFS semantics.

NTFS does not allow a directory to be renamed if there is an open handle on a file in the directory or in any of its descendents. In a system of 10^5 machines, there will almost always be an open handle on a file somewhere in the namespace, so these semantics would effectively prevent a directory near the root of the tree from ever being renamed. Thus, we instead implement the Unix-like semantics of not name-locking an open file's path.

The results of directory rename operations are not propagated synchronously to all descendent directory groups during the rename operation, because this would unacceptably retard the rename operation, particularly for directories near the root of the namespace tree. Instead, they are propagated lazily, so they might not be immediately visible to all clients.

Windows applications can register to be informed about changes that occur in directories or directory subtrees. This notification is specified to be best-effort, so we support it without issuing read leases to registrants. However, for reasons of scalability, we only support notification on single directories and not on subtrees.

3. File System Features

This section describes the mechanisms behind Farsite's key features, which include reliability and availability, security, durability, consistency, scalability, efficiency, and manageability.

3.1 Reliability and Availability

Farsite achieves reliability (long-term data persistence) and availability (immediate accessibility of file data when requested) mainly through replication. Directory metadata is replicated among members of a directory group, and file data is replicated on multiple file hosts. Directory groups employ Byzantine-fault-tolerant replication, and file hosts employ raw replication.

For redundantly storing file data, Farsite could have used an erasure coding scheme [3] rather than raw replication. We chose the latter in part because it is simpler and in part because we had concerns about the additional latency introduced by fragment reassembly of erasure-coded data during file reads. Some empirical data [25] suggests that our performance concerns might have been overly pessimistic, so we may revisit this decision in the future. The file replication subsystem is a readily separable component of both the architecture and the implementation, so it will be straightforward to replace if we so decide.

With regard to reliability, replication guards against the permanent death of individual machines, including data-loss failures (such as head crashes) and explicit user decommissioning. With regard to availability, replication guards against the transient inaccessibility of individual machines, including system crashes, network partitions, and explicit shutdowns. In a directory group of R_D members, metadata is preserved and accessible if no more than $\lfloor (R_D - 1) / 3 \rfloor$ of the machines die. For files replicated on R_F file hosts, file data is preserved and accessible if at least one file host remains alive.

In the Farsite design, when a machine is unavailable for an extended period of time, its functions migrate to one or more other machines, using the other replicas of the file data and directory metadata to regenerate that data and metadata on the replacement machines. Thus, data is lost permanently only if too many machines fail within too small a time window to permit regeneration.

Because the volume of directory data is much smaller than that of file data, directory migration is performed more aggressively than file-host migration: Whenever a directory group member is down or inaccessible for even a short time, the other members of the group select a replacement randomly from the set of accessible machines they know about. Since low-availability machines are – by definition – up for a smaller fraction of time than high-availability machines, they are more likely to have their state migrated to another machine and less likely to be an accessible target for migration from another machine. These factors bias directory-group membership toward highly available machines, without introducing security-impairing non-randomness into member selection. This bias is desirable since

Byzantine agreement is only possible when more than two thirds of the replicas are operational. The increase in the (light) metadata workload on high-availability machines is compensated by a small decrease in their (heavy) file storage and replication workload.

Farsite improves global file availability by continuously relocating file replicas at a sustainable background rate [14]. Overall mean file uptime is maximized by achieving an equitable distribution of file availability, because low-availability files degrade mean file uptime more than high-availability files improve it. Therefore, Farsite successively swaps the machine locations of replicas of high-availability files with those of low-availability files, which progressively equalizes file availability. Simulation experiments [14] driven by actual measurements of desktop machine availability show that Farsite needs to swap 1% of file replicas per day to compensate for changes in machine availability.

File availability is further improved by caching file data on client disks. These caches are not fixed in size but rather hold file content for a specified interval called the *cache retention period* (roughly one week) [6].

3.2 Security

3.2.1 Access Control

Farsite uses different mechanisms to enforce write- and read-access controls. Because directory groups only modify their shared state via a Byzantine-fault-tolerant protocol, we trust the group not to make an incorrect update to directory metadata. This metadata includes an *access control list* (*ACL*) of public keys of all users who are authorized writers to that directory and to files therein. When a client establishes cryptographically authenticated channels to a directory group's members, the channel-establishment protocol involves a user's private key, thereby authenticating the messages on that channel as originating from a specific user. The directory group validates the authorization of a user's update request before accepting the update.

Because a single compromised directory-group member can inappropriately disclose information, Farsite enforces read-access control via strong cryptography, as described in the next subsection.

3.2.2 Privacy

Both file content and user-sensitive metadata (meaning file and directory names) are encrypted for privacy.

When a client creates a new file, it randomly generates a symmetric *file key* with which it encrypts the file. It then encrypts the file key using the public keys of all authorized readers of the file, and it stores the file key encryptions with the file, so a user with a corresponding private key can decrypt the file key and therewith the file. Actually, there is one more level of indirection because of the need to identify and coalesce identical files (see subsection 3.6.1) even if they are encrypted with different user keys: The client first computes a one-way hash of each block of the file, and this hash is used as a key for encrypting the block. The file key is used to encrypt the hashes rather than to encrypt the file blocks directly. We call this technique *convergent encryption* [12], because identical file plaintext converges to identical ciphertext, irrespective of the user keys. Performing encryption on a block level enables a client to write an individual block without having to rewrite the entire file. It also enables the client to read individual blocks without having to wait for the download of an entire file from a file host.

To prevent members of a directory group from viewing file or directory names, they are encrypted by clients before being sent to the group, using a symmetric key that is encrypted with the public keys of authorized directory readers and stored in the directory metadata [15]. To prevent a malicious client from encrypting a syntactically illegal name [31], the Farsite design uses a technique called *exclusive encryption*, which augments a cryptosystem in a way that guarantees that decryption can only produce legal names no matter what bit string is given as putative ciphertext [15].

3.2.3 Integrity

As long as fewer than one third of the members of a directory group are faulty or malicious, the integrity of directory metadata is maintained by the Byzantine-fault-tolerant protocol. Integrity of file data is ensured by computing a Merkle hash tree [30] over the file data blocks, storing a copy of the tree with the file, and keeping a copy of the root hash in the directory group that manages the file's metadata. Because of the tree, the cost of an in-place file-block update is logarithmic – rather than linear – in the file size. The hash tree also enables a client to validate any file block in logarithmic time, without waiting to download the entire file. The time to validate the entire file is linear in the file size, not log-linear, because the count of internal hash nodes is proportional to the count of leaf content nodes.

3.3 Durability

When an application creates, modifies, renames, or deletes a file or directory, these updates to metadata are committed only on the client's local disk and not by a Byzantine operation to the directory group, due to the high cost of the latter. The updates are written into a log (much as in Coda [22]), which is compressed when possible via techniques such as removing matching create-delete operation pairs. The log is pushed back to the directory group periodically and also when a lease is recalled, as described in subsection 3.4. Because

client machines are not fully trusted, the directory group verifies the legality of each log entry before performing the update operation that it specifies.

When a client machine reboots after a crash, it needs to send committed updates to the directory group and have the group accept them as validly originating from the user. The two obvious ways of accomplishing this are unacceptable: Private-key-signing every committed update would be prohibitively expensive (roughly a disk seek time), and holding the user's private key on the client through a crash would open a security hole.

Instead, when first contacting a directory group, a client generates a random *authenticator key* and splits it into secret shares [46], which it distributes among members of the directory group. This key is not stored on the client's local disk, so it is unavailable to an attacker after a crash (modulo the lack of crypto-paging in the underlying operating system – see subsection 2.1). With this key, the client signs each committed update using a message authentication code (MAC) [29]. (Symmetric-key MACs are much faster than public-key signatures.) When recovering from a crash, the client sends the MAC to the directory group along with the locally committed updates. In a single transaction, the group members first batch the set of updates, then jointly reconstruct the authenticator key, validate the batch of updates, and discard the key. Once the key has been reconstructed, no further updates will be accepted. (The recovery phase of this process has not yet been implemented.)

Modifying a file (unlike creating, renaming, or deleting it) affects not only the file metadata but also the file content. It is necessary to update the content atomically with the metadata; otherwise, they may be left in an inconsistent state following a crash, and the file content will be unverifiable. For the rare [48] cases when an existing block is overwritten, the new content is logged along with the metadata before the file is updated, so a partial write can be rolled forward from the log. When a client appends a new block to the end of a file, the content need not be logged; it is sufficient to atomically update the file length and content hash after writing the new block. If a crash leaves a partially written block, it can be rolled backward to an empty block [20].

3.4 Consistency

The ultimate responsibility for consistency of file data and directory metadata lies with the directory group that manages the file or directory. However, temporary, post-hoc-verifiable control can be loaned to client machines via a lease mechanism. There are four *classes* of leases in Farsite: content leases, name leases, mode leases, and access leases. They are described in the following four subsections.

3.4.1 Data Consistency

Content leases govern which client machines currently have control of a file's content. There are two content-lease *types*: *read/write* and *read-only*, and they support single-writer multi-reader semantics. A read-only lease assures a client that the file data it sees is not stale. A read/write lease entitles a client to perform write operations to its local copy of the file. Applications are not aware of whether their clients hold content leases.

When an application opens a file, the client requests a content lease from the directory group that manages the file. When the application closes the file, the client does not immediately cancel its lease, since it may soon need to open that file again [48]. If another client makes a valid request for a new content lease while the first client holds a conflicting lease, the directory group will recall the lease from the first client. When this client returns the lease, it will also push all of its logged updates to the directory group. The group will apply these updates before issuing the new content lease to the other client, thereby maintaining consistency.

Since read/write and write/write file sharing on workstations is usually sequential [22, 48], it is often tolerably efficient to ping-pong leases between clients as they make alternate file accesses. However, as a performance optimization, the Farsite design includes a mechanism similar to that in Sprite [35]: redirecting concurrent non-read-only accesses through a single client machine. This approach is not scalable, but Farsite is not designed for large-scale write sharing. We have not yet determined an appropriate policy for when concurrent non-read-only accesses should switch from lease ping-pong to single-client serialization.

Content leases have variable granularity: A lease may cover a single file, or it may cover an entire directory of files, similar to a volume lease in AFS [21]. Directory groups may issue broader leases than those requested by the client, if no other clients have recently made conflicting accesses to the broader set of files.

Because clients can fail or become disconnected, they may be unable to respond to a lease recall. To prevent this situation from rendering a file permanently inaccessible, leases include expiration times. The lease time varies depending upon the type of lease (i.e., read-only leases last for longer than read/write leases) and upon the observed degree of sharing on the file. The directory group treats lease expiration identically to a client's closing the file without making further updates.

There are several options for handling lease expiration on a disconnected client: The most pessimistic strategy is to close all handles to the file and drop any logged updates, since it may not be possible to apply the updates if the file is modified by another client after the

lease expires. We could, however, save the logged updates and, if the file has not changed, apply them after re-establishing communications with the directory group. We may wish to keep file handles open for read access to potentially stale data, or – most optimistically – even allow further updates, since we might be able to apply these later. We have not yet explored this space nor implemented any mechanisms.

Since the Windows directory-listing functions are specified to be best-effort, we support them using a snapshot of directory contents rather than with a lease.

As described in subsection 2.7, the number of leases issued per file is limited for performance reasons.

3.4.2 Namespace Consistency

Shared file-system namespaces commonly [34] contain regions that are private to particular users. To allow clients to modify such regions without having to frequently contact the directory group that manages the region, Farsite introduces the concept of *name leases*.

Name leases govern which client machine currently has control over a name in the directory namespace. There is only one type of name lease, but it has two different meanings depending on whether a directory (or file) with that name exists: If there is no such extant name, then a name lease entitles a client to create a file or directory with that name. If a directory with the name exists, then the name lease entitles a client to create files or subdirectories under that directory with any non-extant name. This dual meaning implies that when a client uses a name lease to create a new directory, it can then immediately create files and subdirectories in that directory. To rename a file or directory, a client must obtain a name lease on the target name.

Like content leases, name leases can be recalled if a client wants to create a name that falls within the scope of a name lease that has been issued to another client. The discussion above regarding expiration of content leases also applies to name leases.

3.4.3 Windows File-Sharing Semantics

Windows supports application-level consistency by providing explicit control over file-sharing semantics. When an application opens a file, it specifies two parameters: (1) the access mode, which is the types of access it wants, and (2) the sharing mode, which is the types of access it is willing to concurrently allow others. There are many different access modes, but from the perspective of a distributed system, they can be distilled down to three (and finer distinctions can be enforced locally by the client): read access, write access, and delete access. There are also three sharing modes: read sharing, write sharing, and delete sharing, which permit other applications to open the file for read access, write access, and delete access, respectively.

As an example, if an application tries to open a file with read access mode, the open will fail if another application has the file open without read sharing mode. Conversely, if an application tries to open a file without read sharing mode, the open will fail if another application has the file open with read access mode.

To support these semantics, Farsite employs six types of *mode leases*: *read*, *write*, *delete*, *exclude-read*, *exclude-write*, and *exclude-delete*. When a client opens a file, Farsite translates the requested access and share modes into the corresponding mode-lease types: Each access mode implies a need for the corresponding mode lease, and the lack of each sharing mode implies a need for the corresponding exclude lease. For example, if an application opens a file for read access, the client will request a read mode lease, and if the open allows only read sharing, then it will also request exclude-write and exclude-delete mode leases. When a directory group processes a client's open request, it determines whether it can issue the requested leases without conflicting with any extant mode leases on the file. If it cannot, it contacts the client or clients who hold conflicting mode leases to see if they are willing to have their leases revoked or downgraded, which they may be if their applications no longer have open handles on the file. The mode-lease conflicts are the obvious ones: read conflicts with exclude-read, write with exclude-write, and delete with exclude-delete.

3.4.4 Windows Deletion Semantics

Windows has surprisingly complex deletion semantics: A file is deleted by opening it, marking it for deletion, and closing it. Since there may be multiple handles open on a file, the file is not truly deleted until the last handle on a deletion-marked file is closed. While the file is marked for deletion, no new handles may be opened on the file, but any application that has an open delete-access handle can clear the deletion mark, thereby canceling the deletion and also permitting new handles on the file to be opened.

To support these semantics, the Farsite design employs three types of *access leases*: *public*, *protected*, and *private*. A public access lease indicates that the lease holder has the file open. A protected lease includes the meaning of a public lease, and it further indicates that no other client will be granted access without first contacting the lease holder. A private lease includes the meaning of a protected lease, and it further indicates that no other client has any access lease on the file. When a client opens a file, the managing directory group checks to see whether it has issued a private or protected access lease to any other client. If it has, it downgrades this lease to a public lease, thereby forcing a metadata pushback (as described in subsection 3.3), before issuing an access lease to the new client.

To mark a file for deletion, a client must first obtain a private or protected access lease on the file. If the directory group thereafter receives a different client's request to open the file, downgrading this access lease to a public lease will force a metadata pushback, thus informing the directory group of whether the client has delete-marked the file. If the file has been marked, then the directory group will deny the open request.

If a client has a private access lease on a file, it is guaranteed that no other client has an open handle, so it can delete the file as an entirely local operation.

3.5 Scalability

The Farsite design uses two main mechanisms to keep a node's computation, communication, and storage from growing with the system size: hint-based pathname translation and delayed directory-change notification.

When a directory group becomes overloaded, at least $\lceil (2 R_D + 1) / 3 \rceil$ of its R_D members can sign a certificate that delegates part of its namespace to another group. When a client attempts to open a file or directory with a particular pathname, it needs to determine which group of machines is responsible for that name. The basis mechanism is to contact successive directory groups until the responsible group is found. This search begins with the group that manages the root of the namespace, and each contacted group provides a delegation certificate that indicates which group to contact next.

This basis approach clearly does not scale, because all pathname translations require contacting the root directory group. Therefore, to perform translations in a scalable fashion, the Farsite design uses a hint-based scheme that tolerates corrupt directory group members, group membership changes, and stale delegations: Each client maintains a cache of pathnames and their mappings to directory groups, similar to prefix tables in Sprite [35]. A client translates a path by finding the longest-matching prefix in its cache and contacting the indicated directory group. There are three cases: (1) Because of access locality, the most common case is that the contacted group manages the pathname. (2) If the group manages only a path prefix of the name, then it replies with all of its delegation certificates, which the client adds to its hint cache. (3) If the group does not manage a path prefix of the name, then it informs the client, which removes the stale hint that led it to the incorrect group. In the latter two cases, the client again finds the longest-matching prefix and repeats the above steps. Because of the signed delegation certificates, no part of this protocol requires Byzantine operations. Each step either removes old information or adds new information about at least one pathname component, so the translation terminates after no more than twice the number of components in the path being translated.

Directory-change notification is a Windows mechanism that allows applications to register for callbacks when changes occur to a specified directory. The archetypal example is Windows Explorer, which registers a notification for the directory currently displayed. Since Windows specifies this notification to be best-effort, the Farsite design supports it in a delayed manner: For any directory for which a notification has been registered, the directory group packages the complete directory information, signs it to authenticate its contents, and sends it to the registered clients. The transmission work can be divided among members of the directory group and also among clients using application-level multicast [16]. Farsite clients automatically register for directory-change notification when a user lists a directory, so repeat listings need not make multiple remote requests.

3.6 Efficiency

3.6.1 Space Efficiency

File and directory replication dramatically increase the storage requirements of the system. To make room for this additional storage, Farsite reclaims space from incidentally duplicated files, such as workgroup-shared documents or multiple copies of common applications.

Measurements of 550 desktop file systems at Microsoft [6] show that almost half of all occupied disk space can be reclaimed by eliminating duplicates. The Farsite design detects files with duplicate content by storing file-content hashes in a scalable, distributed, fault-tolerant database [12]. It then co-locates replicas of identical files onto the same set of file hosts, where they are coalesced by Windows' Single Instance Store [7].

3.6.2 Time Efficiency (Performance)

Many of the mechanisms already described have been designed in part for their effect on system performance. Caching encrypted file content on client disks improves not only file availability but also file-read performance (further improved by caching decrypted file content in memory). Farsite's various lease mechanisms and its hint-based pathname translation not only reduce the load on directory groups but also eliminate the latency of network round trips. Performance is the primary motivation behind limiting the count of leases per file, using Merkle trees for data integrity, and MAC-logging metadata updates on clients.

In addition, Farsite inserts a delay between the creation or update of a file and the replication of the new file content, thus providing an opportunity to abort the replication if the operation that motivated it is superseded. Since the majority of file creations and updates are followed by deletions or other updates shortly thereafter [4, 48], this delay permits a dramatic reduction in network file-replication traffic [6].

3.7 Manageability

3.7.1 Local-Machine Administration

Although Farsite requires little central administration, users may occasionally need or want to perform local-system administrative tasks, such as hardware upgrades, software upgrades, and backup of private data.

When using a local file system, upgrading a machine's hardware by replacing the disk (or by replacing the entire machine) necessitates copying all file-system structure and content from the old disk to the new one. In Farsite, file data and metadata are replicated on other machines for reliability, so removing the old disk (or decommissioning an old machine) is merely a special case of hardware failure. In its capacities as file host and directory group member, the machine's functions will be migrated; and in its capacity as a client, the hint cache and content cache will gradually refill. However, since users replace their machines far more frequently than machines fail, Farsite's reliability is substantially improved by providing users a means to indicate their intention to decommission a machine or disk, spurring a preemptive migration of the machine's functions [6].

To address the need for upgrades and bug fixes, Farsite supports interoperation between machines running different versions of its software by including major and minor version numbers in connection-establishment messages. Minor-version changes can – by definition – be ignored, and later versions of the software can send and understand all earlier major versions of messages. This backward compatibility permits users to self-pace their upgrades, using a suggestion-based model similar to Windows Update [32]. Updated executables need not be sourced centrally: If they are signed using a private key whose public counterpart is baked into the Farsite code, upgrades can be obtained from any other machine running a more recent version of Farsite code.

Backup processes are commonly used for two purposes: reliability and archiving. In Farsite, there is little need for the former, since the multiple on-line copies of each file on independent machines should be more reliable than a single extra copy on a backup tape whose quality is rarely verified. For archival purposes, automatic on-line versioning [40] would be a more valuable system addition than manual off-line backup. However, if users still wish to backup their own regions of the namespace, existing backup utilities should work fine, except for two problems: pollution of the local cache and weakening of privacy from storing decrypted data on tape. To address the first problem, we could exploit a flag that Windows backup utilities use to indicate their purpose in opening files: Farsite could respond to this flag by not locally caching the file. The second problem, which is outside of Farsite's domain, could be addressed by an encrypting backup utility.

3.7.2 Autonomic System Operation

Farsite administers itself in a distributed, Byzantine-fault-tolerant fashion. Self-administration tasks are either lazy follow-ups scheduled after client-initiated operations or continual background tasks performed periodically. The details of these tasks (file replication, replica relocation, directory migration, namespace delegation, and duplicate-file identification/coalescing) have been described above, but in this section we describe their substrate: two semantic extensions to the conventional model of Byzantine fault-tolerance, *timed Byzantine operations* and *Byzantine outcalls*.

In the standard conception of Byzantine fault-tolerant distributed systems [8], an operation is initiated by a single machine and performed by a Byzantine-fault-tolerant replica group. This modifies the shared state of the group members and returns a result to the initiator.

For lazy and periodic tasks (e.g., replica relocation), directory groups need to initiate operations in response to a timer, rather than in response to a client request. This is complicated because Byzantine replicas must perform operations in lock step, but the clocks of group members cannot be perfectly synchronized. Farsite supports timed Byzantine operations via the following mechanism: The Byzantine-replicated state includes R_D *member time* values, each associated with one of the R_D machines in the group. The kth largest member time (where $k = \lfloor (R_D - 1) / 3 + 1 \rfloor$) is regarded as the *group time*. When a machine's local time indicates that a timed operation should be performed, it invokes the Byzantine protocol to update its replicated member time to the machine's local time. By modifying one of the member times, this update may change the group time, in which case the group performs all operations scheduled to occur at or before the new group time.

Self-administration tasks invert the standard model of Byzantine-fault-tolerance: The directory group invokes an operation (e.g., instruction to create a file replica) on a single remote machine, which sends a reply back to the group. We perform these Byzantine outcalls using a hint-based scheme: A Byzantine operation updates the replicated state to enqueue a request to a remote machine. Then, members of the directory group, acting as individuals, send hint messages to the remote machine, suggesting that it invoke a Byzantine operation to see if the directory group has any work for it to do. When the machine invokes the Byzantine operation, the request is dequeued and returned to the invoking machine. The hint messages are staggered in time, so in the common case, after one member sends a hint message, the remote machine dequeues its request, preventing the other members from sending redundant hints. If the machine needs to reply to the group, it does so by invoking another Byzantine operation.

4. Implementation

Farsite is implemented as two components: a user-level service (daemon) and a kernel-level driver. The driver implements only those operations whose functionality or performance necessitates placement in the kernel: exporting a file-system interface, interacting with the cache manager, and encrypting and decrypting file content. All other functions are implemented in user level: managing the local file cache, fetching files from remote machines, validating file content, replicating and relocating files, issuing leases, managing metadata, and executing the Byzantine-fault-tolerant protocol.

Windows' native remote file system is RDR / SRV, a pair of file-system drivers that communicate via the CIFS protocol [42]. RDR lives under RDBSS, a driver that provides a general framework for implementing network file systems. RDR accepts file-system calls from applications and redirects them via the network to SRV running on another machine, which communicates with NTFS to satisfy the request.

We took advantage of this framework by implementing Farsite's kernel driver under RDBSS. The *Farsite Mini Redirector* (*FMR*) accepts file-system calls, interacts with Farsite's user-level component, uses NTFS as a local persistent store, and performs encryption and decryption on the file-I/O read and write paths.

Our user-level component is called the *Farsite Local Manager* (*FLM*). The control channel by which FLMs on different machines communicate is an encrypted, authenticated connection established on top of TCP. The data channel by which encrypted file content is retrieved from other machines is a CIFS connection established by the RDR and SRV components of the two machines. This architecture is illustrated in Fig. 1.

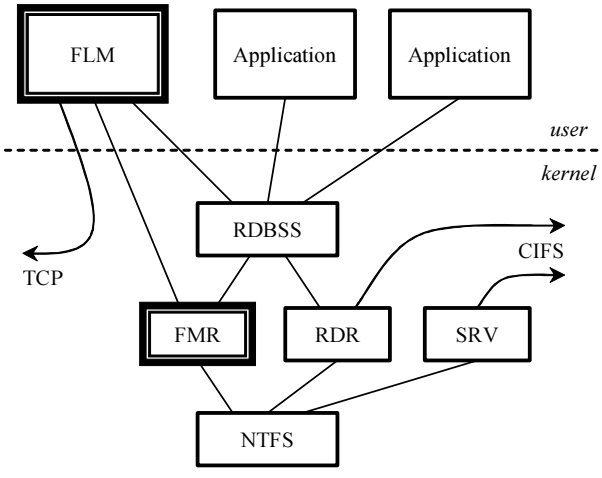

Fig. 1: Farsite components in system context

5. Evaluation

In the following two subsections, we evaluate Farsite's scalability by calculating the expected central loads of certification, directory access, and path translation as a function of system size; and we evaluate Farsite's performance by benchmarking our prototype.

5.1 Scalability Analysis

Farsite's scalability target is ~10^5 machines. Scale is potentially limited by two points of centralization: certification authorities and root directory groups.

Farsite's certification authorities sign certificates offline rather than interactively processing verifications online. Therefore, their workload is determined by the number of private-key signatures they have to perform, not by the much greater number of verifications required. If a company issues machine and user certificates with a one-month lifetime, then in a system of 10^5 machines and users, the CA will have to sign 2×10^5 certificates per month. Each RSA private-key signature takes less than 10 ms of CPU time, so the total computational workload is less than one CPU-hour per month.

Load on a directory group can be categorized into two general classes: the direct load of accesses and updates to the metadata it manages and the indirect load of performing pathname translations for directory groups that manage directories lower in the hierarchy. As a Farsite installation grows, the count of client machines in the system, all of which could concurrently access the same directory, also grows. Since the count of files per directory is independent of system scale [6], the direct load is bounded by limiting the number of outstanding leases per file (see subsection 2.7) and by distributing the transmission of directory-change notifications (see subsection 3.5) using application-level multicast, which is demonstrably efficient for such batch-update processes [16].

The indirect load on a directory group due to pathname translations is heavily reduced by the hint-caching mechanism described in subsection 3.5. When a new client joins the system, its first file access contacts the root directory group. If machines have a mean lifetime of one year [6], then a system of ~10^5 machines will see roughly 300 new machines per day, placing a trivial translation load on the root group from initial requests. Since the group responds with all of its delegation certificates, each client should never need to contact the root group again. Before a directory group's delegation certificates expire, it issues new certificates with later expiration dates and passes them down the directory-group hierarchy and on to clients.

5.2 Performance Measurements

For our performance evaluation, we configured a five-machine Farsite system using 1-GHz P3 machines, each with 512 MB of memory, two 114-GB Maxtor 4G120J6 drives, and a 100-Mbps Intel 82559 NIC, interconnected by a Cisco WS-C2948G network switch. Four machines served as file hosts and as members of a directory group, and one machine served as a client.

We collected a one-hour NTFS file-system trace from a developer's machine in our research group. The trace, collected from 11 AM to noon PDT on September 13, 2002, includes 450,164 file-system operations whose temporal frequency and type breakdown are closely representative of the file-system workload of this machine over the working hours in a measured three-week period. We replayed this trace on a Farsite client machine, and for comparison, we also replayed it onto a local NTFS partition and to a remote machine via CIFS.

Fig. 2 shows CDFs of operation timings during these experiments. For operations with very short durations (less than 7 µsec), Farsite is actually faster than NTFS, primarily due to a shorter code path. Nearly half of all operations are in this category. For the remainder, except for the slowest 1.4% of operations, Farsite's speed is between that of NTFS and CIFS.

We broke down the operation timings by operation type for the six types that took 99% of all I/O time. Farsite's mean operation durations are 2 to 4 times as long as those of NTFS for reads, writes, and closes; they are 9 times as long for opens; and they are 20 times as long for file-attribute and directory queries. Over the entire trace, Farsite displayed 5.5 times the file-I/O latency of NTFS. Some of this slowdown is due to making kernel/user crossings between the FMR and the FLM, and much of it is due to untuned code.

Relative to CIFS, Farsite's mean operation durations are 2 times as long on writes but only 0.4 times as long on reads and 0.7 times as long on queries. Overall, Farsite displayed 0.8 times the file-I/O latency of CIFS.

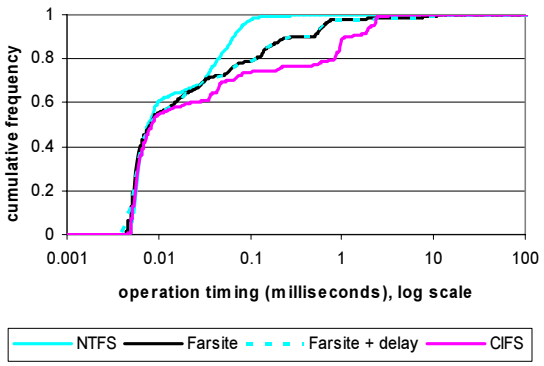

Fig. 2: CDF of trace-replay operation timings

Table 1: Andrew benchmark timings (seconds)

NTFS	CIFS	Farsite
1.9×10^4	3.4×10^4	3.7×10^4

To evaluate Farsite's sensitivity to network latency, we inserted a one-second delay into network transmissions. Because the vast majority of Farsite's operations are performed entirely locally, the effect on the log-scaled CDF was merely to stretch the thin upper tail to the right, which is nearly invisible in Fig 2. This delay did, however, add 10% to Farsite's total file-I/O latency.

Primarily because it is customary to do so, we also ran a version of the Andrew benchmark that we modified for Windows and scaled up by three orders of magnitude. It performs successive phases of creating directories, copying files, listing metadata, processing file content, and compiling the file-system code; each phase has a footprint of 5 – 11 GB. Table 1 shows total run times, but such Andrew benchmark results do not reflect a realistic workload. In particular, Farsite performs the directory-creation phase nearly twice as fast as NTFS, largely because NTFS is inefficient at creating a batch of millions of directories.

6. Future Work

Although we have implemented much of the Farsite design, several significant components remain, mainly those concerned with scalability (namespace delegation, distributed pathname translation, and directory-change notification) and those concerned with crash-recovery (directory-group membership change and torn-write repair). The mechanism for distributed duplicate detection is operational but not yet integrated into the rest of the system. Several smaller components have not yet been completed, including exclusive encryption of filenames, serializing multiple writers on a single machine, supporting lease expirations on clients, and full support for Windows deletion semantics.

Farsite requires two additional mechanisms for which we have not yet developed designs. First, to prevent a single user from consuming all available space in the system, we need a mechanism to enforce per-user space quotas; our intent is to limit each user's storage-space consumption to an amount proportional to that user's machines' storage-space contribution, with an expected proportionality constant near unity [6]. Second, Farsite relocates file replicas among machines according to the measured availability of those machines [14], which requires a mechanism to measure machine availability. Given Farsite's design assumptions, these mechanisms must be scalable, decentralized, fault-tolerant, and secure, making their design rather more demanding than one might initially presume.

7. Related Work

Farsite has many forebears in the history of network file systems. NFS [39] provides server-based, location-transparent file storage for diskless clients. AFS [21] improves performance and availability on disk-enabled workstations via leases and client-side file caching. In Sprite [35], clients use prefix tables for searching the file namespace. Coda [22] replicates files on multiple servers to improve availability. The xFS [1] file system decentralizes file-storage service among a set of trusted workstations, as does the Frangipani [45] file system running on top of the Petal [24] distributed virtual disk. All of these systems rely on trusted machines, and the decentralized systems (xFS and Frangipani) maintain per-client state that is proportional to the system size.

An important area of distributed-file-systems research, but one that is orthogonal to Farsite, is disconnected operation. The Coda [22], Ficus [37], and Bayou [44] systems researched this area extensively, and Farsite could adopt the established solution of application-specific resolvers for concurrent update conflicts.

Several earlier networked file systems have addressed one or more aspects of security. Blaze's Cryptographic File System [5] encrypts a single user's files on a client machine and stores the encrypted files on a server. BFS [8] replaces a single NFS server with a Byzantine-fault-tolerant replica group. SUNDR [27] guarantees file privacy, integrity, and consistency despite a potentially malicious server, but it does this by placing trust in all client machines (unlike Farsite, which requires only that each user trust the client machine he is directly using). SFS [26] constructs "self-certifying pathnames" by embedding hashes of public keys into file names, thus defending read-only data against compromised servers or compromised networks.

A number of distributed storage systems attempt to address the issue of scalability. Inspired by peer-to-peer file-sharing applications such as Napster [33], Gnutella [17], and Freenet [9], storage systems such as CFS [10] and PAST [38] employ scalable, distributed algorithms for routing and storing data. Widespread data distribution is employed by the Eternity Service's [2] replication system and by Archival Intermemory's [18] erasure-coding mechanism to prevent data loss despite attack by powerful adversaries, and PASIS [49] additionally employs secret sharing for data security. OceanStore [23] is designed to store all of the world's data (10^{23} bytes) using trans-continentally distributed, Byzantine-fault-tolerant replica groups to provide user-selectable consistency semantics. These systems have flat namespaces; they do not export file-system interfaces; and (with the exception of OceanStore) they are designed for publishing or archiving data, rather than for interactively using and updating data.

8. Conclusions

Farsite is a scalable, decentralized, network file system wherein a loosely coupled collection of insecure and unreliable machines collaboratively establish a virtual file server that is secure and reliable. Farsite provides the shared namespace, location-transparent access, and reliable data storage of a central file server and also the low cost, decentralized security, and privacy of desktop workstations. It requires no central-administrative effort apart from signing user and machine certificates.

Farsite's core architecture is a collection of interacting, Byzantine-fault-tolerant replica groups, arranged in a tree that overlays the file-system namespace hierarchy. Because the vast majority of file-system data is opaque file content, Farsite maintains only indirection pointers and cryptographic checksums of this data as part of the Byzantine-replicated state. Actual content is encrypted and stored using raw (non-Byzantine) replication; however, the architecture could alternatively employ erasure-coded replication to improve storage efficiency.

Farsite is designed to support the file-I/O workload of desktop computers in a large company or university. It provides availability and reliability through replication; privacy and authentication through cryptography; integrity through Byzantine-fault-tolerance techniques; consistency through leases of variable granularity and duration; scalability through namespace delegation; and reasonable performance through client caching, hint-based pathname translation, and lazy update commit.

In large part, Farsite's design is a careful synthesis of techniques that are well known within the systems and security communities, including replication, Byzantine-fault-tolerance, cryptography, certificates, leases, client caching, and secret sharing. However, we have also developed several new techniques to address issues that have arisen in Farsite's design: Convergent encryption permits identifying and coalescing duplicate files encrypted with different users' keys. Exclusive encryption enforces filename syntax while maintaining filename privacy. A scalable, distributed, fault-tolerant database supports distributed duplicate-file detection. We use a novel combination of secret sharing, message authentication codes, and logging to enable secure crash recovery. Timed Byzantine operations and Byzantine outcalls supplement the conventional model of Byzantine fault-tolerance to permit directory groups to perform autonomous maintenance functions.

Analysis suggests that our design should be able to scale to our target of ~10^5 machines. Experiments demonstrate that our untuned prototype provides tolerable performance relative to a local NTFS file system, and it performs significantly better than remote file access via CIFS.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang. Serverless Network File Systems. *15th SOSP*, Dec 1995.

[2] R. J. Anderson, "The Eternity Service", *PRAGO-CRYPT '96*, CTU Publishing, Sep/Oct 1996.

[3] R. E. Blahut, *Theory and Practice of Error Control Codes*, Addison Wesley, 1983.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. "Measurements of a Distributed File System." 13th SOSP, Oct 1991.

[5] M. Blaze, "A Cryptographic File System for Unix", *Ist Computer and Comm. Security*, ACM, Nov 1993.

[6] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs", *SIGMETRICS 2000*, ACM, Jun 2000.

[7] W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur. Single Instance Storage in Windows 2000. *4th Usenix Windows System Symposium*, Aug 2000.

[8] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *3rd OSDI*, USENIX, Feb 1999.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System", *ICSI Workshop on Design Issues in Anonymity and Unobervability*, Jul 2000.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, "Wide-Area Cooperative Storage with CFS", *SOSP*, Oct 2001.

[11] J. R. Douceur, "The Sybil Attack", *1st IPTPS*, Mar 2002.

[12] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, M. Theimer, "Reclaiming Space from duplicate Files in a Serverless Distributed File System", *ICDCS*, Jul 2002.

[13] J. R. Douceur and W. J. Bolosky, "A Large-Scale Study of File-System Contents", *SIGMETRICS*, May 1999.

[14] J. R. Douceur and R. P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System", *20th SRDS*, IEEE, Oct 2001.

[15] J. R. Douceur, A. Adya; J. Benaloh; W. J. Bolosky; G. Yuval, "A Secure Directory Service based on Exclusive Encryption", (to appear) *18th ACSAC*, Dec 2002.

[16] J. Gemmell, E. M. Schooler, J. Gray, "Fcast Multicast File Distribution: 'Tune in, Download, and Drop Out'", *Internet, Multimedia Systems and Applications*, 1999.

[17] Gnutella, http://gnutelladev.wego.com.

[18] A. Goldberg and P. Yianilos, "Towards an Archival Intermemory", International Forum on Research and Technology Advances in Digital Libraries, Apr 1998.

[19] J. Gray. "Why do Computers Stop and What Can Be Done About It?", *5th SRDS*, Jan. 1986.

[20] J. Gray and A. Reuter, *Transaction Processiong: Concepts and Techniques*. Morgan Kaufmann, 1993.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, M. West, "Scale and Performance in a Distributed File System", *TOCS* 6(1), Feb 1988.

[22] J. Kistler, M. Satyanarayanan. Disconnected operation in the Coda File System. *TOCS* 10(1), Feb 1992.

[23] J. Kubiatowicz, et al., "OceanStore: An Architecture for Global-Scale Persistent Storage", *9th ASPLOS*, ACM, Nov 2000.

[24] E. Lee, C. Thekkath. Petal: Distributed virtual disks. *7th ASPLOS*, Oct 1996.

[25] M. Luby, "Benchmark Comparisons of Erasure Codes", http://www.icsi.berkeley.edu/~luby/erasure.html

[26] D. Mazières, M. Kaminsky, M. F. Kaashoek, E. Witchel, "Separating Key Management from File System Security", *SOSP*, Dec 1999.

[27] D. Mazières and D. Shasha, "Don't Trust Your File Server", 8th HotOS, May 2001.

[28] M. McKusick, W. Joy, S. Leffler, R. Fabry. A Fast File System for UNIX. *TOCS*, 2(3):181-197, Aug 1984.

[29] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.

[30] R. Merkle, "Protocols for Public Key Cryptosystems", *IEEE Symposium on Security and Privacy*, 1980.

[31] Microsoft, "File Name Conventions", *MSDN*, Apr 2002.

[32] Microsoft, "About Windows Update", http://v4.windowsupdate.microsoft.com/en/about.asp

[33] Napster, http://www.napster.com.

[34] E. Nemeth, G. Snyder, S. Seebass, T. R. Hein, *UNIX System Administration Handbook*, Prentice Hall, 2000.

[35] J. K. Ousterhout, A, R. Cherenson, F. Douglis, M. N. Nelson, B. B. Welch, "The Sprite Network Operating System", *IEEE Computer Group Magazine* 21 (2), 1988.

[36] M. Pease, R. Shostak, L. Lamport "Reaching Agreement in the Presence of Faults", *JACM* 27(2), Apr 1980.

[37] G. J. Popek, R. G. Guy, T. W. Page, J. S. Heidemann, "Replication in Ficus Distributed File Systems", IEEE Workshop on Management of Replicated Data, 1990.

[38] A. Rowstron and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility", *SOSP*, Oct 2001.

[39] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon. Design and Implementation of the Sun Network File System. *Summer USENIX Proceedings*, 1985.

[40] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, J. Ofir, "Deciding When to Forget in the Elephant File System", *17th SOSP*, Dec 1999.

[41] S. T. Shafer, "The Enemy Within", *Red Herring*, Jan 2002.

[42] D. A. Solomon and M. E. Russinovich, *Inside Microsoft Windows 2000 Third Edition*, Microsoft Press, 2000.

[43] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, "Scalability in the XFS File System", *USENIX*, 1996.

[44] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System, *15th SOSP*, 1995.

[45] C. Thekkath, T. Mann, E. Lee. Frangipani: A Scalable Distributed File System. *16th SOSP*, Dec 1997.

[46] M. Tompa and H. Woll, "How to Share a Secret with Cheaters", *Journal of Cryptology* 1(2), 1998.

[47] S. Travaglia, P. Abrams, *Bastard Operator from Hell*, Plan Nine Publishing, Apr 2001.

[48] W. Vogels. File system usage in Windows NT 4.0. *17th SOSP*, Dec 1999.

[49] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kilite, P. Khosla, "Survivalbe Information Storage Systems", *IEEE Computer* 33(8), Aug 2000.

[50] B. Yee and J. D. Tygar, "Secure Coprocessors in Electronic Commerce Applications", *USENIX 95*, 1995.