The Consequences of Decentralized Security in a Cooperative Storage System

Douglas Thain, Christopher Moretti, Paul Madrid, Philip Snowberger, and Jeffrey Hemmes
Department of Computer Science and Engineering
University of Notre Dame

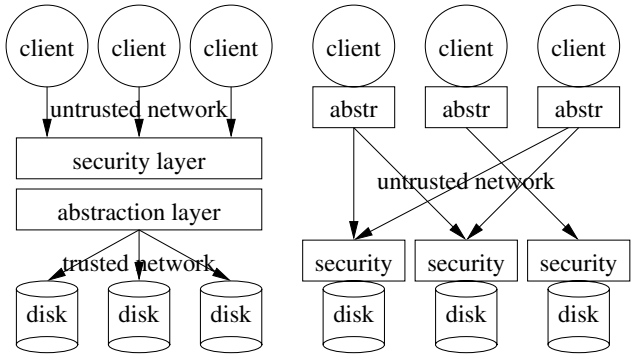
Abstract

Traditional storage systems have considered security as a problem to be solved at the perimeter: once a user is authenticated, each device internal to the system trusts the decision made elsewhere. However, as storage systems become ever more distributed, shared, and dynamic, it becomes necessary to enforce security at the boundaries of each storage device, rather than around the system as a whole. This form of decentralized security presents several new challenges in the design and implementation of distributed storage systems. We explore challenges in distributed file systems, third party transfer, active storage, and group management in the context of a 200-node cooperative storage system deployed at the University of Notre Dame. These explorations result in three recommendations for future system designs.

Key words: Decentralized security, distributed file systems, third party transfer, active storage, access control.

1 Introduction

Traditional storage systems place security concerns high in the complex software stack that separates end users from physical storage. This design decision comes from architectures carried over from conventional filesystem design. For example, a traditional operating system kernel identifies the calling user at the system call interface, and then asserts that identity without question at every layer below, including the virtual filesystem switch, the filesystem code, and the device driver. In a similar fashion, a modern network storage appliance identifies the calling user as requests come in from the network and then asserts that identity into an internal RAID or SAN. Even large distributed storage systems often assume trust between all of the internal nodes of a system. Essentially, the security concerns are solved in a centralized manner, even if the back end storage may be parallel or distributed.



(A) Centralized Security **(B) Decentralized Security**

Figure 1. Centralized vs. Decentralized
A decentralized security system moves the primary security mechanism closer to each storage device. Although this places more power in the hands of each device owner, it introduces new challenges for the clients of the system.

However, as storage devices become more and more distributed, shared, and dynamic, it becomes necessary to move the security mechanisms deeper and deeper into the software stack. When users may harness multiple storage devices, each perhaps owned by different people, each device must be responsible for enforcing the requirements of its owner. When many users collaborate in a large distributed or grid computing system, each device must be responsible for enforcing the requirements of the hosting institution. When multiple applications with varying privilege levels operate within a single domain, each device must independently enforce policy appropriate to each application. In each of these situations, both the devices and the security mechanisms are distributed. Figure 1 compares centralized and decentralized security mechanisms.

In this paper, we explore how the necessities of distributed security mechanisms affect the design and implementation of traditional distributed system abstractions. We

Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)
0-7695-2537-7/05 $20.00 © 2005 IEEE

explore these issues in the context of a 200-node cooperative storage system deployed across several clusters and workstations at the University of Notre Dame. Each node consists of a user-level file server that exports space to remote users, subject to the policy constraints of its owner. Each server is independently responsible for enforcing a local security policy with local mechanisms. On this substrate, we may build up structures such as filesystems and databases, but their design and operation is greatly affected by decentralized security mechanisms and policies.

We explore three problems in detail. First, we consider how decentralized security affects the semantics of a distributed file system spread across multiple nodes. A difficulty arises from that fact that one concept (filesystem permission) is divided into two mechanisms (directory permission and storage permission.) Curiously, this difficulty offers an interesting new opportunity in file system management. Second, we consider how decentralized security affects the problem of third-party transfer, a necessary technique for efficient use of large storage systems. We survey several possible techniques and describe one solution based on temporary access controls. Third, we consider how decentralized security affects the concept of active storage. Essentially, the notion of process ownership must be divorced from the local user database, allowing a system to run programs associated with arbitrary identities. Finally, we consider how to implement distributed user groups.

Our contribution is a discussion of both the challenges and opportunities that arise by pushing security mechanisms deeper into the software storage stack. Relying on existing techniques for authentication and encryption, we consider how raising new barriers affects the design and implementation of distributed storage. Based on this experience, we offer three recommendations for future systems.

2 Overview of Cooperative Storage

A brief overview of a cooperative storage system is needed before proceeding to the main matter on decentralized security. The interested reader may consult an earlier publication for greater detail on the semantics, performance, and applications of cooperative storage [25]. Sections 2 and 4.2 of this paper recap material from the previous paper. In addition, section 5.3 applies material from an earlier work on sandboxing [24]. More detail on the adapter component is available in reference [26].

A cooperative storage system is a large collection of storage devices owned by multiple users, bound into a loosely coupled distributed system. There are many reasons why cooperating users may bind together multiple devices: they may backup data to mitigate the risk of failure; they may construct large repositories that cannot fit on any single disk; they may improve performance by spreading data

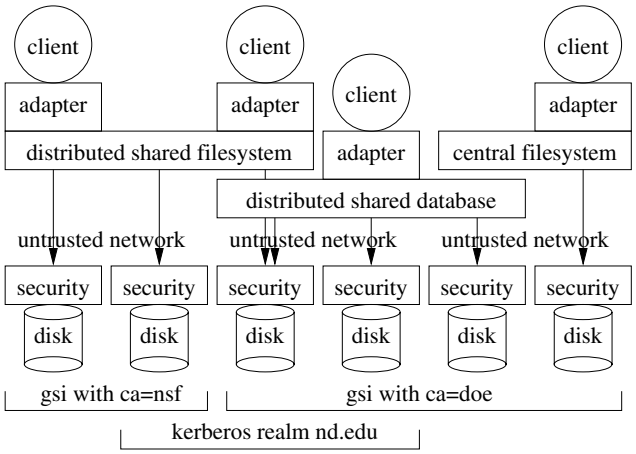


Figure 2. A Cooperative Storage System.
A cooperative storage system consists of multiple independent storage devices, each with a distinct security policy and mechanism. Users create a variety of abstractions on the devices, and connect them to applications with adapters.

across multiple devices; or they may wish to share data and storage space with external collaborators. We assume that users have some external reason to cooperate and so we do not explore issues of fairness or compensation. However, we do assume that resource owners wish to control quite explicitly whom they cooperate with. One user may be willing to share public data with the world at large, while another might only share scratch space with one trusted colleague. Others might share resources with an organization such as a university department or a commercial operation.

Figure 2 shows the structure of a cooperative storage system. At the lowest level, it is composed of an array of file servers that export a Unix-like I/O interface. Each file server may conceivably have a different owner with distinct authentication and authorization policies. In practice, we expect that groups of servers will have a common owner and policy. For example, in our prototype system, there are four distinct owners, while the servers are grouped into six research clusters and five workstations clusters, each with roughly uniform policies limiting what users to admit. As the figure suggests, various groupings of the machines may accept different types of user credentials.

Building up from the basic filesystem interface, users may construct a variety of *abstractions* using any subset of the available file servers. The simplest abstraction is the central file system (CFS), by which multiple clients may share a single filesystem remotely. A distributed shared filesystem (DSFS) abstraction spreads file data across multiple servers while exporting a single filesystem interface. A distributed shared database (DSDB) uses a database to index large data files spread across multiple servers. In

Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)
0-7695-2537-7/05 $20.00 © 2005 IEEE

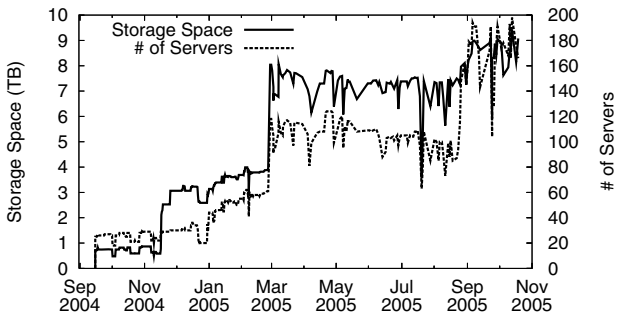


Figure 3. Growth of UND Storage Pool

The size of a storage system over one year, measured at weekly intervals. Although the system grows in several deliberate bursts, the most notable feature is the fluctuation in running servers available at any given time.

each case, an *adapter* is used to connect abstractions to applications. The current adapter interposes on system calls through the `ptrace` interface, allowing unmodified applications to use the abstractions layered on the servers.

By separating storage resources from the abstractions that use them, users gain a great deal of freedom. They may create, destroy, and reconfigure abstractions without any interaction with a system administrator or resource owner. As long as they work within the policies established by each resource owner, users may work as they see fit.

The challenges discussed in this paper have been explored within the context of one cooperative storage system deployed at the University of Notre Dame and currently employed by several user communities in high energy physics and molecular dynamics. Figure 3 shows the growth of this system over the course of the last year, measured at weekly intervals. The system was initially deployed on two research clusters in fall 2004, expanded to research and desktop machines on a department scale in early 2005, and then expanded to classroom workstations in fall 2005. However, the more notable feature is a high fluctuation in available servers at each measurement. In a system of any scale, this behavior is to be expected: servers come and go according to the failures, reboots, and policies of each machine owner. This behavior must be understood and accommodated at higher levels of software, as we will discuss below.

3 Challenges of Decentralization

A cooperative storage system has decentralized mechanisms and policies. Each device is responsible for performing both authentication and authorization upon the clients that connect to it. This grants the owner of each file server fine-grained control over how it is used. It also introduces new challenges in the design and implementation of distributed systems. Some of these challenges are:

Unbounded Set of Users. In a decentralized system, there is no static, global list of users. Certificate authorities may mint new identities daily or hourly. Users may wish to express access controls that refer to identities that a given system has not yet encountered. No two systems are likely to know of the same subset of users. For these reasons, we cannot construct a simple local user database or even assign unique integer IDs to users. We must identify users by strings that refer to external authorities. This prevents us from making use of conventional file system technologies such as NFS or CIFS, which rely heavily upon integers.

Multiple Identities per User. A given user may be known by multiple distinct identities, depending upon the ability of both the user and the system to carry out a given authentication technique. A user may choose among multiple valid identities, depending upon the task she wishes to carry out. For the same reasons as above, we cannot establish a canonical list of identities for a single user.

New Decision Points. A complex operation in a cooperative storage system may involve several servers, each of which may have a different obligation to perform authentication and authorization. What might be an unchecked activity in traditional system (i.e. allocating a block on disk) might now be limited by a new access control. Such an action might succeed at one access control, but fail at the next. Users must be aware of the multiple decision points when deploying or debugging a system.

Unexpected Policy Coupling. When multiple servers must participate in an activity, their policies may interact in unexpected ways. As we will show in an example below, a user that deploys a filesystem across multiple nodes may accidentally discover that the file placement policy interacts with the possible access controls that may be placed upon those files. A higher-level system that wishes to retain its own freedom of policy must vet the policies of the components that may employ before attempting to use them.

To explore these challenges in detail, we must first describe the basic security mechanisms supported by the file servers. Higher-level abstractions that use these basic mechanisms will encounter these challenges.

4 Basic Security Mechanisms

Each file server is responsible for enforcing a local security policy by performing authentication on each incoming client and then authorizing each operation that it attempts. We may consider both phases of security independently.

4.1 Authentication

Each file server provides several authentication methods. The simple `hostname` method allows a client to be identified as the domain name of the connecting host. The `unix` method relies on a challenge and response within the local filesystem: the server challenges the client to touch a file in `/tmp` and then infers the client's identity from the response. (Of course, the `unix` method can only be used to authenticate a user on the same machine, and is typically only employed to identify the server's owner.) The `globus` method allows a client to authenticate via the Globus Grid Security Infrastructure (GSI) [7]. A file server may hold either user or host GSI credentials. The `kerberos` method uses the Kerberos [22] private key authentication system.

Of course, GSI and Kerberos are each logically centralized systems. Each user of GSI has a certificate that must be signed by a central authority. If a client and server do not share a common authority, they cannot communicate. Likewise, a Kerberos client and server must share the same key server. However, the system is still decentralized in the sense that there may be multiple disjoint roots of trust. Figure 2 shows four hosts using GSI, trusting the DOE certificate authority, two using GSI trusting NSF, and three using Kerberos in the Notre Dame realm.

To negotiate an acceptable authentication method, the client proposes a method, which the server may accept or decline. If the server accepts, both sides attempt to authenticate. If the server declines or the attempt fails, then the client may propose another method. By default, client software attempts all methods, but the user may override the order if needed. Once authenticated, the agreed-upon identity is returned to the client for verification, and the client is free to attempt file accesses. Each access is then controlled by the authorization controls described below.

Many readers initially express some reservations about two aspects of this mechanism. First, it may seem unusual that the client controls the selection of the authentication method. Shouldn't the client present all of its methods, and the server choose among them? (i.e. "I want to see a passport, but failing that a driver's license, and failing that, a birth certificate.") A counterexample demonstrates that this is not enforceable. A devious client could reverse the order by only claiming the ability to perform one method, and then disconnect and re-connect if the method is declined. Thus, we allow the client to control the negotiation by choosing one method at a time, observing that it does not force the server to accept a method it does not respect.

Second, it may seem restrictive that the user can only employ one identity at a time. Many users employing GSI have multiple credentials issued by multiple certificate authorities. All users could be identified by hostname in addition to any other method. We argue that, sooner or later, users must choose one credential with which to perform a task. When creating a new object, it must be assigned an owner for accounting purposes. With multiple simultaneous identities, users would be forced to choose one identity to be the owner. When performing debugging or administration, users would need additional controls to temporarily drop identities they do not wish to use. Thus, we force the user to make a choice once when authenticating and the result simplifies later steps. A user with multiple credentials must re-connect in order to act with a different identity.

Once authenticated, the client is assigned a string identity that is used for all later authorization decisions. The name is composed of the method name, followed by a name decided by the authentication mechanism. For example, the following are all valid names for one author of this paper, each employing a different authentication mechanism:

```
unix:dthain
hostname:hedwig.cse.nd.edu
kerberos:dthain@nd.edu
globus:/O=NotreDame/CN=DouglasThain
```

Note that the many ways of identifying oneself can easily lead to confusion. A common problem in our deployed system is the expiration of credentials. By default, GSI and Kerberos credentials expire after several hours after which the negotiation mechanism automatically falls back to `hostname`. Often, users are puzzled because they are able to login, yet are denied access to their files. Two mechanisms have been added to prevent confusion. First, the server supports a `whoami` RPC that sends back the client the server's perception of its identity. This is displayed prominently in client-side tools. Second, users can manually choose to employ one and only one authentication method, so that the expiration of credentials leads to failed connections and a more explicit error. This problem arises again in third-party transfer, which we explore below.

Note also that each user of a system is given a security context that corresponds to the user-level adapter. Two simultaneous users on a given client may bear different credentials and will authenticate to servers on different TCP connections. Security contexts are not mixed together on a single connection, as in filesystems derived from NFS.

4.2 Authorization

Each file server stores ordinary directories and files as in a Unix filesystem. However, instead of employing the usual Unix mode bits for protection, each directory is protected by an ACL resembling those used in AFS [11] and similar systems. Each entry in the ACL lists a free-form text subject and a list of access rights granted to that subject. A subject may include wildcards if needed. For example, the following ACL allows for all hosts in a domain to read, write, and

list, but also allow a given Kerberos user also modify the ACL.

```
hostname:*.cse.nd.edu     RWL
kerberos:dthain@nd.edu    RWLA
```

Of course, it makes no sense for multiple users to share the same directory with all of their files mixed together, each readable and writable by the other. So, the file servers support a *reservation* (V) right which allows an authenticated user to create a new directory with fresh permissions. Using the V right, a more sensible top-level ACL would be:

```
hostname:*.cse.nd.edu     V(RWL)
kerberos:dthain@nd.edu    V(RWLA)
```

This would allow the named users to perform only a `mkdir` in the top level directory. The newly created directory would obtain the rights following the V for only the named user. Reservation is similar to the concept of *amplification* [12]. For example, if `hedwig.cse.nd.edu` were to perform `mkdir` at the top level, the newly created directory would have permissions:

```
hostname:hedwig.cse.nd.edu     RWL
```

Note that reservation allows the storage owner to control whether the accessing user can delegate access rights to external users. This is a variation on mandatory access control (MAC.) In the example above, users authenticating via `hostname` may use the storage but not grant rights to others. However, the `kerberos` user gains the A right via reservation and would be able to grant access to others.

These combined techniques of negotiated authentication and access control lists with reservation give each individual storage device a high degree of flexibility in security mechanism and policy.

5 Consequences of Decentralization

Decentralization of security brings both new challenges and opportunities. Some formerly simple interactions between components are made more complex, but new modes of interaction between users become possible. We consider how decentralization affects shared filesystems, third party transfers, active storage, and distributed access control.

5.1 Distributed Shared Filesystems

Figure 4 shows the structure of a distributed shared filesystem (DSFS), which allows users to bind multiple file server together into one large filesystem. Much like the Google file system [8], one file server is used to store the directory tree, while the remaining file servers are used to store the actual file data. On the directory server, directories are stored as ordinary directories, while the files are *stub files* that contain pointers to other file servers. The abstraction layer at the client side is responsible for interpreting the stubs and transparently serving the requested data.

A DSFS has a more complex permissions model than a traditional file system. In order to access a file, a client must satisfy the access controls on both the directory server and the relevant file server. Likewise, in response to a request to change the access controls on a file, the DSFS must change the access controls on both the directory server as well as the corresponding file server. This is not a problem for the user that creates and configures the filesystem, but it may be a significant problem for others that gain access.

For example, consider a user that establishes a DSFS using a directory server that he owns and using file servers located on workstations owned by his department. Suppose that he can establish arbitrary access controls on his directory server, but the department workstations will only allow access to department members, and do not grant the A right to those that allocate space. He creates a large filesystem spread across these devices. Now, he wishes to grant access to a certain file to an external collaborator. He is able to change the access controls on the directory server, but the policy of the workstations prevents him from granting access to the external user, so the change of permissions fails!

There are several ways to attack this problem. First, the user might request a change in the access control policy of the file servers. Failing that, he could relocate the desired data to another server with a less restrictive sharing policy. If that proves to be too expensive, he could deploy a proxy server that authenticates the remote user and relays the operations to the restricted file server. (Although this doesn't violate the letter of the file server policy, it likely violates the spirit.) Of course, he user might have avoided this problem in the first place if he had considered whether the target servers would be compatible with likely access controlled policy. Thus, with decentralized security, *data placement policy must be guided by intended access control.*

This in turn, requires examination of the relationships along the sharing-owning continuum. A system that is optimized for ease of sharing (both within-department and with external parties) necessarily has weaker access restrictions. This can result in a lesser "personal ownership" of the disk space and the possibility for users to experience discontent with sharing their space, in general, or with certain other parties. This has to be balanced, however, against those users' gained ability in an easily-shared system to consume many more resources than he has individually available. Additionally, ease of sharing is necessary to implement a DSFS that evenly distributes files across several disks.

Viewing both the distinction between directory and storage access control and the interaction along the sharing-

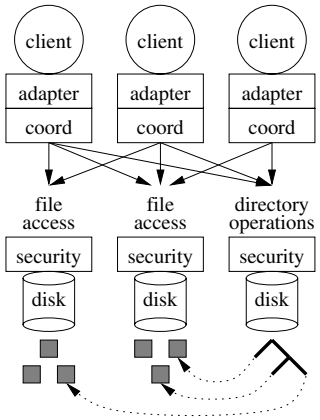


Figure 4. Distributed Shared Filesystem

A user may employ several storage devices in order to build a filesystem larger than can fit on a single device. A large directory structure can be stored on one disk, while the files themselves are scattered across others. Decentralized security complicates access control mechanisms in this system.

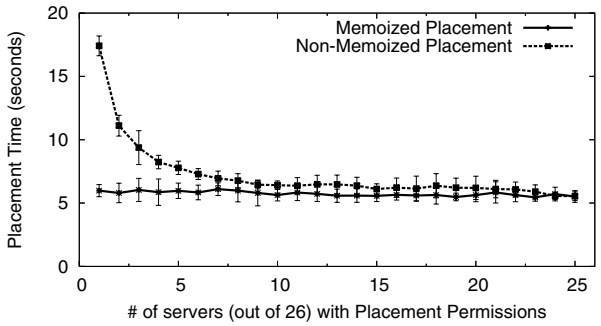


Figure 5. Effectiveness of Memoization in File Placement

A memoized round-robin placement algorithm performs comparably with a non-memoized version of the same algorithm for systems with high node-availability, and outperforms the non-memoized version by larger margins as node-availability decreases.

owning continuum from another perspective can be used to solve a perennial problem in file system design: organizations often wish to share a namespace while avoiding the resource contention that typically comes with it. A common example of this is a shared file server for home directories. Users enjoy the convenience of being able to access their home directory from any machine in an organization. However, the usual approach of putting all home directories on a file server results in users competing for both storage space and file server bandwidth. One aggressive user can prevent others from performing work.

Decentralized security can be used to share a namespace without sharing server resources for data transfers. A DSFS can be established for an organization, giving each user a home directory within the same filesystem namespace. However, each user can be assigned a private file server — perhaps their own workstation — on which only they are allowed to consume space. The entire directory namespace remains shared between users, but both space and load are partitioned for each user. (Of course, load on the directory server is still shared.) Individual users may even provision new space without administrator help by bringing a new file server online and storing new files there.

Not surprisingly, however, the policy for diverse users and hosts that make up an organizational shared namespace DSFS raises performance issues. For example, a problem with giving each user his own file server in the context of the larger namespace is file placement. An administrator for the DSFS may logically conclude that some users will use very little of their space, while others will needlessly overload their allotted disk, so distributing data from heavily burdened disks is a good idea for performance. This is as simple as changing some of the parameters of the DSFS file placement algorithms, yet without a wholesale change in policy to allow every user access on every disk, this policy of attempted distribution can have disastrous performance implications in such a system. If there are a large number of hosts in the namespace, but many users (those without the special permission to spread their data out for the mentioned reasons) have access to only their own disks, applying an even-distribution round-robin placement algorithm to these users' systems will result in unsuccessfully attempting to create/open-for-writing on every machine on the system before rotating back to its one available host.

An effective solution to avoiding this worst-case scenario is an implementation that memoizes on accessible disks. Specifically, the placement algorithm "remembers" whether it had permission on a host before. In this DSFS situation it is unlikely that a user will suddenly gain access on another host to which he previously was denied, memoizing the list of hosts unavailable to him will save fruitless attempts to place a file where he has insufficient permissions. This is a key and subtle characteristic, without which the solution could not only be ineffective, but actually detrimental. If hosts that are currently denying access to a user are likely to grant the user access in the future, performance could be hurt, and impartial distribution among all the available disks prevented, by permanently eliminating temporarily unavailable hosts. Additionally, for the case where all disks are available, memoization of available hosts is unnecessary.

Figure 5 is a performance comparison between a memoized round-robin placement algorithm and a non-memoized

version of the same round-robin algorithm. The data were collected on a DSFS containing 26 identical storage servers, each located approximately the same distance from the directory server. The storage servers were Pentium 4 3.2 GHz PCs running GNU/Linux. The times shown are the mean times to place 260 1KB files into the DSFS. The results indicate that memoization can give significant performance benefits; in our tests, memoizing accessible disks cut the average time to place a file on the pathological case by more than half. On the other extreme, where all but one of the disks are accessible, the memoized algorithm achieved performance measures similar to the non-memoized algorithm.

5.2 Third Party Transfer

Many storage systems rely heavily on third-party transfer: the transfer of data from a source to a target under the control of client in a third location. Third-party transfer may not be strictly required — a system can accomplish a third-party transfer by way of two ordinary transfers — but it may result in a dramatic performance improvement by avoiding the transfer to the client. As an example application, a filesystem may rely on third-party transfer in order to replicate files for efficient backup. By using third-party transfer instead of ordinary transfers, it can manage several simultaneous replications at once while taking advantage of the parallelism inherent in switched networks.

Of course, third-party transfer is hardly a new concept. Third-party transfer has been present in FTP [17] for decades. The difficulty arises when third-party transfers must be authenticated. FTP provides no authentication for this mechanism, so it is disabled in most servers. In systems with centralized security, this difficulty is not a problem because the entire transfer is authenticated externally and then carried out between two mutually trusting devices.

The real question is how to perform third-party transfer in a system with decentralized security. The fundamental problem is that the target must have some way of knowing that the source is acting on behalf of the client. There are three solutions to this problem that we are aware of, each with some benefits and drawbacks.

One solution is *delegation of credentials*. This solution is employed by GSI-enabled FTP servers [1]. In this model, the client transmits an attenuated proxy certificate to the source server, which then uses it to authenticate to the target server. The target server then perceives the source to be equivalent to the client, and permits the transfer. This solution is powerful, but it requires all parties to share a common certificate authority and requires the client to trust the source not to steal the proxy and use it to impersonate the client elsewhere. Proxy certificates are typically given short lifetimes to limit potential damage.

Another solution is to use *capabilities*. An example of the capability approach is demonstrated by the Internet Backplane Protocol [16]. In order to perform a third party transfer, the client obtains a write capability from the target server and passes it to the source when requesting a transfer. The source presents the write capability to the target and is permitted to make the transfer. Capabilities are certainly an elegant solution to this problem but have long been a source of some controversy. Most relevant to this discussion are the facts that the secure transmission of capabilities relies on some existing key-based infrastructure such as GSI or Kerberos and that extensive use of capabilities requires careful storage management by the client. It should be noted that capabilities have attracted more attention in recent years and future storage systems may offer extensive support for capabilities.

A third solution is to use *temporary access controls*, shown in Figure 6. The basic file server ACL mechanism is used to give the source permission to write a file in the proper place on the target, which is then reverted after the transfer is completed. The wrinkle is that it is not always obvious to the client precisely how the source will authenticate to the target. If using `hostname` authentication, for instance, the use of domain name aliases or multiple network interfaces might yield an unexpected name which may not map to the permissions that the client will set to facilitate the transfer. If the target holds a GSI or Kerberos host certificate, it may be able to identify itself with its own credentials instead of that of the client.

In order to avoid such confusion, the client asks the source to attempt to authenticate to the target in order to determine the expected name. The source then issues a whoami command to the target to find out the name that the client must use and then feeds this data back to the client. This whole sequence is called a *remote whoami*.

Now that the client knows exactly how the source will authenticate to the target, it can initiate the third-party transfer using temporary access controls. Third-party transfer under this scheme works in this fashion:

1. The client requests a `rwhoami` of the source file server, asking it to `whoami` against the target server.
2. The source file server authenticates to the target server and performs a `whoami` to determine its identity. This name is returned to the client.
3. The client adjusts the access control on the target in order to allow the source to write the necessary file.
4. The client instructs the source to `thirdput` a file to the target server.
5. The source transmits the file to the target.
6. The client reverse the access controls.

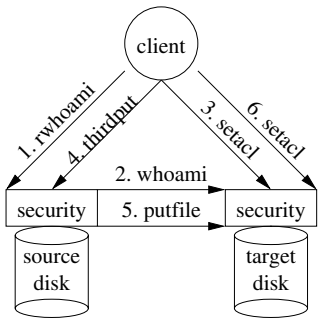


Figure 6. Third Party Transfer

Third-party transfer with decentralized security requires several steps. 1. The client asks the source to test its identity with the target. 2. The source tests its identity. 3. The client changes the ACL on the target to allow the source to write. 4. The client instructs the target to send the file. 5. The source sends the file to the target. 6. The client reverts the ACL changes on the target.

Like the other two methods, temporary access controls are not perfect. If a rogue process on the source machine can employ the same credentials as the file server (i.e. `hostname` authentication), then the target file could be attacked. However, damage would be limited to only that file, and only during an interval controlled by the client. In contrast, stealing delegated credentials would allow impersonation of a user, while stealing a write capability would allow access to the file until the capability was revoked. For clients wary of attacks, the file server provides a remote checksum facility to check for unauthorized change.

Fundamentally, any method for authenticating third party transfer involves some level of risk that must be weighed on a per-application basis. For some applications, the gain in performance and scalability may be worth the risk. For others, it may be simpler and more secure to simple perform two direct transfers.

Figure 7 shows a performance evaluation of third-party transfer using temporary access controls. We transfered variable-length files across 1 Gb/s Ethernet using third-party transfer and the traditional transfer method (get the file, put the file) on three personal computers using 2.8 GHz Pentium 4 processors and running RedHat Enterprise Linux 3. For very small files, the overhead incurred by executing the third-party transfer is greater than the actual file being transfered. For larger files, third-party transfer catches up. Around 6 MB sized files, third-party transfer becomes faster than the traditional method because the overhead incurred by setting up the transfer is smaller than the gain received from avoiding the middle man.

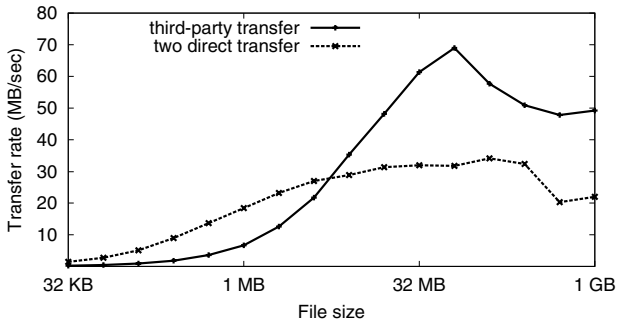


Figure 7. Third-party Transfer Performance

For small files, third party transfer is slightly slower than two direct transfers, due to the overhead of authentication. For larger files, third party transfer achieves twice the bandwidth by avoiding an extra network and disk copy.

5.3 Active Storage

The concept of active storage proposes that storage devices should be equipped with a facility for executing programs directly on the storage device [18, 13]. By doing so, one may avoid large amounts of data transfer over the connection between the main CPU and the disk, whether it be an I/O bus or a long-haul network. Active storage has great potential for parallelizing data-intensive applications, and has been employed to great benefit in databases [18, 19] and filesystems [14, 6, 2].

The cost of performing even very complicated authentication is minimal compared to the potential time savings of executing code remotely rather than pulling the data to the local host and executing locally. The dichotomy between the two approaches becomes even clearer when data is distributed across multiple hosts. In this context, it makes sense to integrate active storage with the DSFS abstraction.

However, a suitable security model for active storage is not yet settled, primarily because it is imagined that active storage programs will execute in devices far below the level of the operating system, where the notion of process identity is very weak. Typically, active storage programs are required to run in a sandbox of some kind, either given only a few capabilities to access streamed data [18], or by restricting programs to trusted codes vetted by the server [2].

We propose that active storage can be achieved for arbitrary applications if the mechanism exists to associate external identities with running processes and files. This allows an active storage server to safely execute a process using only string compare to perform authorization, assuming that a higher level of software has already performed authentication. An ideal implementation of this facility would require modifying the operating system, but it can be approximated

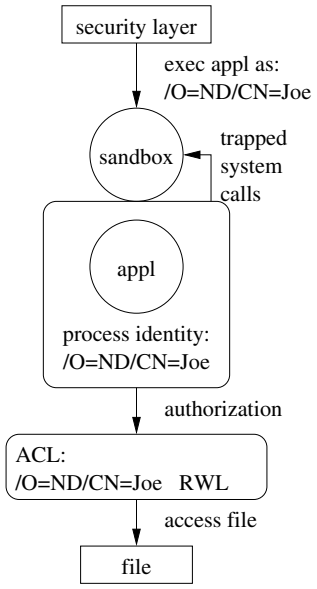


Figure 8. Active Storage via Sandboxing

Active storage is simplified if external identities can be associated with both processes and files. In this model, a sandbox is used to attach an X.509 identity to a process, which then may access files where permitted by ACLs.

through the use of system-call sandboxing technologies.

To demonstrate this concept, we have added an active storage facility to the cooperative storage system. Upon request by a client, a file server can execute an arbitrary application that is already stored on it as a ordinary file. The application is run under the control of a sandbox that traps and interprets its system calls. The application is assumed to be owned by an identity selected by the authentication layer, such as `globus:/O=NotreDame/CN=Joe`, thus we call this technique *identity boxing* [24]. Of course, this identity does not exist in the user database, so there is no equivalent integer user ID. Upon each file access by the application, the sandbox examines ACLs in the file system — the same ACLs enforced by the file server — and determines whether the application may have access. In this way, the execution system can safely run arbitrary code while employing identities established by external mechanisms.

A few remaining details of the authorization system control where and how active storage may be employed. A user must have the X right on a file in order to execute it on a file system. The owner of a file server may provide a fixed list of trusted binaries by providing a directory offering the RX rights to trusted users. Or, an owner might provide a writable directory with the WX rights, allowing a user to stage in and execute arbitrary code.

We illustrate the performance potential of this mechanism with the following experiment. We assume a DSFS containing a large number of files that must be checksummed. In a traditional storage system, this would involve extracting the data from the system and then computing upon it. Using an active storage mechanism, the checksum program is sent to the storage nodes to be executed.

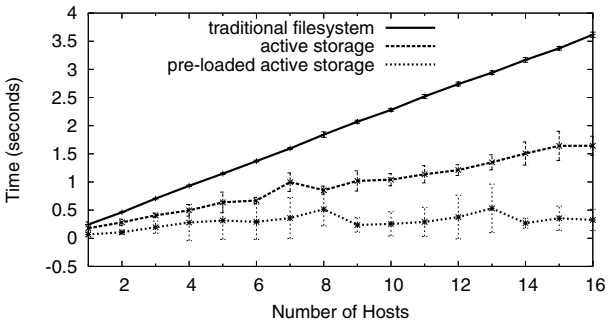


Figure 9. Scalability of Active Storage

Executing tasks in parallel on a traditional file system and an active storage system. Time required grows linearly for traditional and active storage systems, since in both, data or an executable must traverse the network. Time required stays constant when the executable is already in place.

We created a DSFS populated with 2 MB data files to be analyzed with a 512 KB executable. For the traditional storage case, we measured the total time needed to download the data files to the local machine and run the local copy of `md5sum` on them. For the active storage case, we measured the total time needed to upload the analysis executable and calculate remotely the `md5sum` of the data file. We also measured the time needed to only calculate the MD5 checksum, representing the case when the analysis executables have already been distributed. For each case, each task consisting of an upload (or download) and checksum was run in parallel, and the time measured was from the beginning of the first upload (or download) to the completion of the last calculation.

Figure 9 shows that the cost of executing multiple, parallel operations on remote data remains constant versus the number of remote hosts accessed, when the analysis executable has been pre-distributed. This is because in this ideal case, no bulk data traverses the network. In the other cases, the cost of executing said tasks grows linearly with the number of hosts accessed, because either the data or the executable must be transferred. When the executable is of size E and the data files are each of size D, active storage gives a speedup proportional to D/E. Thus, active storage is best for small executables and large data sizes.

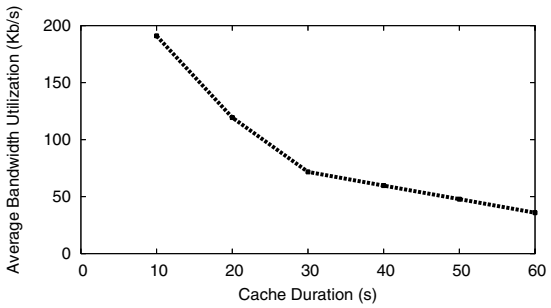


Figure 10. Network Bandwidth Utilization

The impact of caching policies on network throughput. Caching group files, in this case a group of 200,000 users, results in lower average bandwidth consumed as the duration of the cache increases.

Command	Exec Time (ms) (Uncached)	Exec Time (ms) (Cached Decision)
ls	143	47
get	162	12
put	116	1
rm	98	9
mv	151	3
stat	102	2

Table 1. File Command Performance

The impact of decision caches on file commands in Chirp. By eliminating redundant, expensive group lookups, caching individual decisions significantly improves performance of commands that require one or more ACL checks to complete. Group size in all cases is 300,000 users.

5.4 Access Controls for User Groups

As described earlier, each directory in a cooperative storage system is protected by an ACL listing a free-form text subject and a list of access rights granted to that subject. However, a resource owner may wish to grant directory access to a number of users large enough that explicitly listing them in each ACL would be impractical. Not only would each ACL in every shared directory need to be modified separately, but ACLs themselves can grow very large in size as well. To reduce the burden of doing such, we are implementing a distributed group access control mechanism into the cooperative system in which groups of users can be defined by any resource owner at any location within the system and referenced within ACLs. For example, a group entry in a directory ACL would appear as:

```
group:hedwig.cse.nd.edu/dthain/team  RWL
```

which not only specifies the group name and owner, but the host on which the group is defined. Upon encountering an ACL entry such as this, a server would then determine if the current user is a member of the group `team` owned by user `dthain` defined on host `hedwig.cse.nd.edu` and grant or deny permissions accordingly.

Because file servers in a cooperative system may have many concurrent users accessing directories for which ACLs grant permissions on a groupwise basis (and there may be a large number of groups), and in turn each group may potentially contain thousands or even millions of users, performance is a key consideration. To avoid overloading file servers with expensive group lookup operations, we are implementing the means for resource owners to set their own flexible security policies. Resource owners may allow external servers to cache group definition files for a period of time they specify so lookups can be performed locally by those requesting servers as needed. We are also studying the effects of caching individual lookup decisions for a period of time specified by the resource owner.

Such mechanisms allow resource owners to define their own dynamic security policies on a per-server basis within a cooperative system. As is typical in a distributed system, each policy potentially involves tradeoffs between consistency and performance. However, the group access control mechanism presented in this paper allows system users, rather than system administrators, to decide for themselves exactly what and how much to trade off. For instance, caching policies improve lookup performance and overall server performance possibly at some cost of consistency when members are added or removed from group files, or when the groups themselves are deleted, while at the same time providing a degree of fault tolerance in the event a server hosting a group file fails.

Deciding what to cache and for how long may have a cost in terms of either performance (in terms of network bandwidth utilization and CPU time) or consistency. Figure 10 shows the impact of caching policies on network bandwidth utilization. As the duration of the cache increases, average bandwidth utilization decreases possibly at the expense of consistency. Caching the remote file and performing a lookup is expensive, so providing a mechanism for caching individual lookup decisions can improve overall performance in terms of both network utilization and lookup time. Table 1 shows the performance effects of caching lookup decisions on a number of common file system commands.

6 Recommendations

Based on our experience with the above areas, we offer three pieces of advice for the design and implementation of

future systems with decentralized security. We believe these recommendations flow easily from our experience above, and although easily stated, are not obvious, nor necessarily easy to carry out in practice.

Servers must employ meaningful identities deep in the software stack. Traditional operating systems and file systems rely heavily on integers to represent user identity. Integers are used at the system call interface (i.e. `getuid`), in the data structures of filesystems (i.e. inode owners), and in network protocols (i.e. the owner subfield of the `stat` structure). However, the use of integers to represent identity assumes that all participants share a common user database with a consistent mapping of names to numbers. This is not the case in a system with decentralized security. There is no global user database, nor can we expect each participating system to add a new entry to the local user database every time it interacts with a new client.

Instead, meaningful string identities must be used throughout the system, from server processes, to the operating system, down to filesystem storage. We were able to meet this requirement by employing a user-level file server, storing permissions in auxiliary files, and using a custom I/O protocol. If we were to use standard protocols and storage systems, we would simply not be able to store the necessary identity strings. Thus, we recommend that systems move toward a model where identity is stored as a free-form string to be interpreted by higher levels of software. This requires some adjustments to traditional OS structures.

Clients must be prepared for a wide array of failures. Any coordinated activity in a cooperative storage system must be prepared for one part of the transaction to succeed and another to fail. For example, a distributed file system must allow for the possibility that the insertion of a directory entry succeeds, but the creation of the corresponding file object is rejected. (This is not a possibility in conventional file systems.) A third party transfer must allow for the possibility that the ACL change may be accepted, but the file transfer may fail. In each case, the client must record the activities attempted at each site, and then roll back the incomplete changes. Of course, the rollback may fail as well, so the client must choose a sequence of operations that do not pass the system through a dangerous state. For example, the directory insertion must occur before the file creation so that a failure yields a (safe) dangling pointer instead of unreferenced garbage.

Users need tools for debugging security mechanisms. In a decentralized security system, an attempt to access an object may be denied for an enormous number of reasons in both authentication and authorization. Consider how many ways the third party transfer in Figure 6 may fail. The client may fail to negotiate an authentication method with the target. Authentication with the target may fail, if perhaps a certificate has expired. The user may not have permission to change the ACL. A similar array of problems occurs between the client/source and between the source/target. When presented with a failed third party transfer, users are left scratching their heads over what may have failed.

To attack this problem, we require tools for debugging security mechanisms. The `whoami` and `rwhoami` RPCs are examples of very simple debugging tools. To go farther, we might include with each failed operation some details of the reason for the failure: for example, which necessary right was not present in the ACL? Or, client side tools might take a more aggressive approach and probe a server repeatedly, trying similar operations in different combinations to flesh out the scope of a problem. Of course, revealing the reason for a denied access may not be suitable for very high security systems: this may reveal exploitable weakness. For less critical systems, debugging tools would be a great help.

7 Related Work

Although there exist many distributed storage systems with flexible and powerful security models, nearly all have a centralized point for authentication and then perform operations within a trusted perimeter. GPFS [21], PVFS [3], Lustre [4], and the Google file system [8] all follow this model. Riedel et al. [20] offer a framework for evaluating the security mechanisms of systems following this general model. while our focus is more on the unexpected results of new security mechanisms.

The cooperative storage system model has some of the flavor of object-based and network attached storage as described by Gibson et al. [9, 15] Gobioff [10] describes a security model for network attached storage. The primary distinction is that cooperative storage must deal with authentication at a higher semantic level because there is no global mapping of text identities to local tokens.

A variety of systems perform a high-level authentication step at a centralized point, which yields a lower-level credential that can be employed at many distinct sites. For example, Kerberos [22] requires the client to authenticate with the key server and then a ticket granting service, which then issues credentials to interact with individual services. In effect, this is centralized authentication, but decentralized authorization. In a related manner, work in the grid computing area has proposed a centralized authentication and authorization service that issues tickets to access individual objects [5]. This has some of the flavor of capabilities.

Capabilities have long been an object of study in computer architectures [12], as well as distributed computing [23], distributed storage [16], and active storage [19]. This mechanism has always been quite controversial, as capabilities clearly admit a wide range of security policies, but also present new challenges in management, storage, and transmission. In a cooperative storage system, it is nec-

essary to have an ACL-based mechanism, because storage owners wish to have fairly detailed knowledge of and control over remote users.

8 Conclusion

Decentralization of security mechanisms introduces new challenges into the design and implementation of traditional storage systems. We have presented four challenges in shared filesystems, third-party transfer, active storage, and group management. From this experience, we offer three recommendations for future system designers: (1) Systems should store meaningful identities deep in the software stack. (2) Clients must be prepared for a wide array of failures. (3) Users need tools for debugging security mechanisms. The general philosophy of a cooperative storage system is that users should be given many of the powers currently reserved for system administrators. In order to exercise these powers effectively, users require tools that assist them with the natural complexity of such an environment.

References

[1] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*, pages 161–163, 2000.

[2] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Lerna: An active storage framework for flexible data access and management. In *High Performance Distributed Computing*, 2005.

[3] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for linux clusters. In *Annual Linux Showcase and Conference*, 2000.

[4] Cluster File Systems. Lustre: A scalable, high performance file system. white paper, November 2002.

[5] D. Feichtinger and A. Peters. Authorization of data access in distributed systems. In *Workshop on Grid Computing*, 2005.

[6] E. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active storage processing in a parallel file system. In *LCI Conference on Linux Clusters: The HPC Revolution*, 2005.

[7] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[8] S. Ghemawat, H. Gobioff, and S. Leung. The Google filesystem. In *ACM Symposium on Operating Systems Principles*, 2003.

[9] G. A. Gibson, D. 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 *Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1997.

[10] H. Gobioff, D. Nagle, and G. Gibson. Integrity and performance in network attached storage. In *Proceedings of International Symposium on High Performance Computing*, 1999.

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

[12] A. K. Jones and W. A. Wulf. Towards the design of secure systems. *Software - Practice and Experience*, 5(4):321–336, 1975.

[13] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 1998.

[14] X. Ma and A. L. N. Reddy. MVSS: an active storage architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(9), September 2003.

[15] M. Mesnier, G. Ganger, and E. Riedel. Object based storage. *IEEE Communications*, 41(8), August 2003.

[16] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium*, 1999.

[17] J. Postel. FTP: File transfer protocol specification. Internet Engineering Task Force Request for Comments (RFC) 765, June 1980.

[18] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, Carnegie-Mellon University, 1997.

[19] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large scale data mining and multimedia. In *Very Large Databases (VLDB)*, 1998.

[20] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Conference on File and Storage Technology (FAST)*, 2002.

[21] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *USENIX Conference on File and Storage Technologies (FAST)*, Jan 2002.

[22] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Winter Technical Conference*, pages 191–200, 1988.

[23] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *International Conference on Distributed Computing Systems*, 1986.

[24] D. Thain. Identity boxing: A new technique for consistent global identity. In *International Conference for High Performance Computing and Communications (Supercomputing)*, November 2005.

[25] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *International Conference for High Performance Computing and Communications (Supercomputing)*, November 2005.

[26] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.