Cacheable Decentralized Groups for Grid Resource Access Control

Jeffrey Hemmes and Douglas Thain
Technical Report 2006-06
Department of Computer Science and Engineering, University of Notre Dame

Abstract

Sharing data among collaborators in widely distributed systems remains a challenge due to limitations with existing methods for defining groups across administrative domain boundaries with various file systems. Groups in traditional systems are bound to particular domains or file systems using centralized storage locations either beyond ordinary users' ability to manage, inaccessible outside a closed system, or both. We present a method for users to independently create and manage groups on any networked workstation using global user identities and to control access to shared data and storage resources based on group membership, regardless of domain boundaries or underlying file systems. Decentralized groups are decoupled from shared user databases and centralized authentication servers through the use of a virtual user name space. We describe how owners of shared resources can define security policies through the use of caching, and demonstrate how each caching policy represents tradeoffs between performance, scalability, and consistency.

1. Introduction

Today, grid users have access to a wide array of computing resources, with volumes of disk storage space and network performance allowing vast possibilities for sharing data and local computing resources. Unfortunately, despite widespread availability of hardware resources, the potential for large-scale, dynamic collaborations is limited due to insufficient support for defining access controls on shared data. It is still not easy for users to share specific data or storage space with large groups of collaborators without making it world-readable or world-writable.

A number of existing methods permit large-scale collaborations with users from distinct, geographically separated administrative domains. Virtual organizations [8] are commonly used to organize collaborative efforts among grid users. Other methods such as Grid3 [10] allow group sharing by mapping global identities to local Unix accounts or groups. However, these approaches require an administrator to set up and maintain. If none is available when a new user joins or a new group is required, work is delayed. Additionally, remote users typically do not have visibility into locally created groups, making access controls at remote sites for these groups difficult.

To efficiently share local computing resources in large-scale collaborations, two issues must be addressed. First, users should have have the ability to create and manage their own groups and provide security policies for the resources they share based upon such groups. Doing so permits short-term or informal collaborations perhaps less suitable for traditional approaches. Security policies can be set at the resource level via access control lists (ACL), but referencing remote groups in widely distributed systems is a nontrivial task. To be useful in such a system, user-defined groups must be accessible across administrative domains and unique in the system's global namespace.

Part of the challenge lies with the authentication process itself. Traditional systems tightly couple group membership with authentication. This approach requires a centralized authentication server that establishes group identity once when validating user credentials. Groups are commonly implemented using a centralized storage location to record membership lists. For instance, Unix groups are defined in the file `/etc/group`. Even in distributed file systems such as AFS [13] that allow user-managed groups, group definitions are stored in a protection database inaccessible outside the system. Grids are often dynamic and loosely organized, spanning file systems and domain boundaries. Assuming the existence of a centralized file or database server for groups may not be appropriate in all cases.

This paper presents a method for decentralized authorization that allows every node in the system to host user-managed groups using identities drawn from a global namespace without requiring administrator intervention. Such decentralized groups, created and maintained by end users, can be referenced from ACLs on any shared resource, empowering users to independently define security policies without elevated privileges. Furthermore, group membership is decoupled from the authentication process so membership determinations can occur only as needed. We described this approach in an earlier workshop paper [21] but have only outlined the fundamental ideas behind this work and shown the performance of an early implementation. The purpose of this paper is to present a more detailed description of the system architecture and caching model.

Two underlying philosophies provide the basic framework for our approach. First, *resource owners should determine who may access **any** data under their control on their workstations*, to include group membership information. Data belongs to an individual in the same way physical property does. Second, *each user should have the ability to exercise such control autonomously*; changes to local security policies pertaining to collaborations should not require tasks only an administrator can accomplish. Arguments in favor of end-user control of local security policies are certainly nothing new [5].

We have taken a *tactical storage system* (TSS) that allows grid users to dynamically share local storage resources without elevated privileges or kernel modifications [19], and added primitives through which any system user can create and manage groups on individual workstations rather than on centralized servers. We handle performance and consistency tradeoffs through caching any combination of group files, membership lookup results, and the caching policies of remote servers, but ultimately what to cache and for how long is determined solely by each resource owner sharing data or storage space. While a TSS provides a convenient platform for proof of concept, we envision possible application of decentralized groups to other grid systems such as GridFTP [2] through an additional software layer.

2. Decentralized Groups in Tactical Storage

2.1. Overview of Tactical Storage

In order to describe distributed group management, we must briefly detour to describe the tactical storage system.

The foundation of a TSS is an array of user-level file servers. The server depends on existing file systems for data storage but exports a subtree of that file system to remote applications via a Unix I/O interface. Any user can deploy file servers on any machine without elevated privileges and may allow or deny access to other users as they see fit. A TSS may be built from personal workstations, cluster nodes, or large servers. Applications connect to the TSS via an adapter that converts file-server-provided system resources to a format more directly accessible to applications, so developers need not write specifically for the resource or collective layers. Our adapter is Parrot [20], an interposition agent that connects applications to file servers without requiring kernel modifications or special privileges.

Authentication is done using existing infrastructure through one of several possible methods negotiated between the server and the connecting client, which must prove its identity using the selected method. Clients can be identified by either the hostname of the machine, the local Unix identity of the user, or through credentials provided via Globus Grid Security Infrastructure (GSI) [7] or Kerberos [18].

Identities are independent of any shared user database. Once authenticated, a *virtual name space* is used in which identities are represented simply as free-form text strings divorced entirely from their underlying implementations. This decoupling enables cross-domain sharing without reliance on mappings to local native accounts employed by systems such as CAS [16].

With a virtual name space, creating groups of global scope is much simpler, as they are simply lists of strings representing group members. An advantage of this approach is heterogeneous membership; a single group can include host names, Globus identifiers, or any other identity the authentication methods support, and can easily accommodate new types of identities. For example, the following are all valid subject names for one author of this paper:

```
globus:/O=NotreDame/CN=Hemmes
kerberos:jhemmes@nd.edu
hostname:bomber.cse.nd.edu
```

2.2. Group Implementation

The following begins the new contribution of this paper.

A TSS allows any machine to serve as a file server. In the same spirit, any machine may serve as a group server, avoiding centralized storage repositories and allowing users to better manage their own sharing policies. While centralized servers can create a performance bottleneck and single point of failure for the system as a whole, such concerns can be handled effectively in a variety of ways such as replication. However, we believe users should not necessarily be compelled to store their data on servers under a possibly unknown individual's control.

Security policy is enforced with directory ACLs similar to those in AFS. Each entry contains a subject name and a list of permissions for the subject within that directory. Entries may be single subjects, subject patterns, or group references. Consider the following ACL:

```
globus:/O=NotreDame/CN=Hemmes          RWA
globus:/O=NotreDame/*                  R
group:bomber.cse.nd.edu/jhemmes/team   RW
```

This ACL gives one user read, write, and administrator access; all users from Notre Dame read access; and all members of a remote group read and write access. Note that a wild card may be used to implement a group, but only if the issuing authority corresponds exactly to the desired membership. More precise control requires explicit groups, which name a host where the membership may be found.

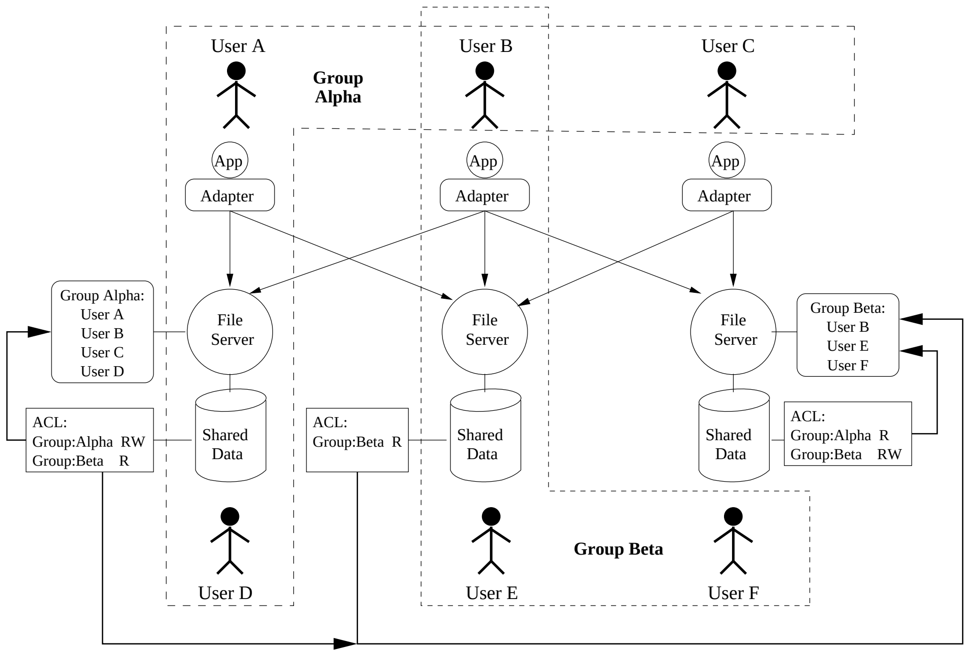

Figure 1. Distributed Access Control in a Grid-Enabled File System

This figure shows the many complex relationships possible with a fully decentralized access control mechanism. Users may access any server from any machine, access controls may refer to groups defined on local or remote machines, and groups may refer to arbitrary users. In this example, Users A-F are placed in overlapping groups Alpha and Beta defined at two distinct machines. Access Control Lists (ACLs) on each server refer (shown via heavy lines) to Alpha, Beta, or both.

Servers resolve the current user's group memberships and record all permissions granted for the directory, rather than only checking for the specific right or set of rights the requested command requires. Specifically, a subject belonging to several groups, each with different associated permissions, would possess the union of all sets of rights for those groups regardless of what is needed to execute the command. The implication is that every ACL entry must be read and every group entry resolved. For security reasons ACL checking should be as algorithmically simple as possible; thus, the complexity of determining whether a user has only a specific set of rights is not warranted.

In contrast to many traditional systems, group files may be stored in any directory in the file system. Group creation requires only write permissions for the group's creator in that directory. Subsequent references to the group must include the path from the root directory of the server much like a web server's URL-path. To ensure uniqueness, group references consist of the fully qualified domain name of the machine hosting the group; the (optional) file server port, and the path to the group file itself.

Groups are implemented with GDBM [9], a commonly available set of lightweight database routines. Membership lookups are accomplished by querying the group definition database for a record corresponding to the user identity. Our concern is with global identities, which are convenient to use as primary keys. To add a user to the group file, create a new key with the user name, which can be any number of identities the system supports, and insert a record. An example would look like:

```
hostname:somehost.nowhere.edu
globus:/O=UnivNowhere/CN=Alice
kerberos:alice@nowhere.edu
globus:/O=UnivNowhere/CN=Bob
kerberos:bob@nowhere.edu
...
```

Regardless of implementation efficiency, on-demand group resolution can eventually become a bottleneck as we expect scalability to group sizes of tens or hundreds of thousands of users. To reduce the overall workload of servers in terms of lookups as well as RPCs required and network bandwidth consumption, we implement caching.

3 Caching and Consistency

Caching is intended to improve performance, but with some consistency cost. Our goal is providing resource own-

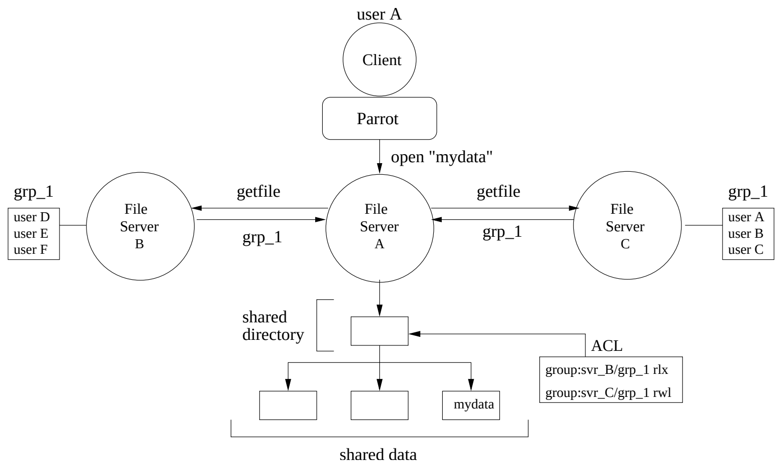

Figure 2. Overview of Group File Caching

This figure depicts the interaction between servers with file caching enabled. Group files are cached upon reading ACLs referencing remotely defined groups. In this example, file server A retrieves group files hosted on servers B and C, referenced by an ACL in server A's shared directory structure.

ers the means to decide for themselves on acceptable trade-offs and implement a security policy that represents the result of such decisions. We do so with three types of caching: group files, membership lookup results, and caching policies of remote servers. Resource owners specify which of these items in any combination remote servers can cache and the duration caches are valid. However, members may be added to or deleted from the original file during the cache duration. Doing so compromises consistency as the hosting server maintains no explicit control over the cached files.

Cache control is handled by a server-specified expiration similar to that of HTTP/1.1 and set or modified by the resource owner. During that duration, caches may be used by requesting servers to resolve remote group lookups, and in general they will unless a cached file is corrupted or deleted. Servers do not track lookup resolution requests, and so are unable to enforce a minimum time interval between remote lookups.

Upon revocation, the cached copies retain the deleted member until expiration, at which time they are brought into a consistent state with the original data. While changes to a group file are not immediately reflected in the caches, the duration specified in the caching policy places an upper bound on the lifetime of stale data. Such caching policies apply individually to each group hosted on a particular server; policies can be adjusted based on the preferences of each resource owner, to include the system default policy of no caching.

Poor consistency can be remedied by adjusting the cache lifetimes appropriately, or in the case of highly volatile group memberships, perhaps by disallowing caching altogether. Adjusting the expiration date is a common approach for dealing with X.509 certificate revocation [11] and our approach is similar in many respects.

3.1 Caching Group Files

In accordance with the caching policy of a remote group, a server performing an ACL check caches the entire group file for subsequent lookups. Such caching policies are mandatory; any references to remote groups on a server with caching enabled will result in caching, with the duration established by any user with write permissions in the directory containing the remote group. Expired caches must be revalidated upon access and refreshed if inconsistent with the original data.

Upon reading an ACL entry referencing a remote group, the requesting server first checks the remote group's caching policy (which may itself be cached). It then checks for a valid file cache by searching an index of all remote group files cached locally. If a cache does not exist, the remote server is contacted and the file is then saved in a designated cache directory and an entry added to the index. The name of the file is randomized to prevent namespace collisions within the cache directory, but metadata about the file and the remote group is saved in the index. Expired caches are revalidated by comparing the timestamp of the cache to the last modification time of the original via RPC.

Each entry in the cache index contains the group name, the remote hostname and port, and the local file name and cache expiration time. A list entry looks like this:

```
host:bomber.cse.nd.edu
```

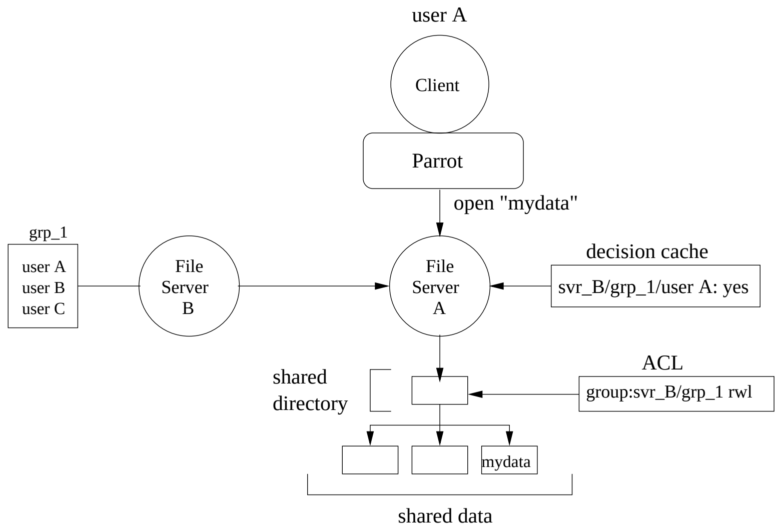

Figure 3. Lookup Decision Caching

In accordance with the caching policy set by the data's owner, lookup results can be cached for subsequent ACL checks. Lookups can be cached whether they were performed remotely, on a cached file, or on a locally hosted group. In this figure, file server A determines user A's membership in remote group `grp_1`*, defined on server B, and caches the result. Use of the decision cache takes precedence over other group resolution methods for as long as the cache is valid.*

```
port:9094
grp_name:/jhemmes/team
file_name:tmpJtcY19
expiration:1135793686
```

While caching files reduces interserver RPCs, lookups must still be performed locally, possibly imposing a performance penalty for operations requiring multiple ACL checks or ACLs with many group entries. To remedy this, policies can allow caching of individual lookup decisions.

3.2 Caching Lookup Decisions

Even with a locally cached group, lookups may still impose a performance penalty, particularly for large numbers of group references in ACLs and large groups. Caching lookup decisions is an effective way to avoid redundant lookups. It is particularly important for groups to which a user does *not* belong, since performing multiple lookups is even more of a waste of time than redundant confirmations.

Each server maintains a list of recently checked user/group lookup results, but only for those groups whose policy allows decision caching. If allowed, the list is checked for the appropriate group/user pair and if the entry is valid (determined by comparing the expiration time of the cache against the current system time) the stored decision is used. If the decision cache has expired, the entry is removed and the stored result ignored, and a lookup is performed either on a locally cached group file or remotely on the hosting server, depending on the caching policy and the validity of the cached group file, if it exists. The result of that lookup is then placed in the list and a new expiration time set. Figure 3 illustrates the decision cache lookup. Upon reading an ACL and detecting a group subject name, the lookup index is checked and the cached result used.

Introducing a second level of caching poses additional consistency challenges, but as with group files, cache controls limit persistence of stale data. When a decision cache expires, a lookup must be re-accomplished either on the cached group file or remotely, depending on policy. If policy permits all caching, use of cached decisions take precedence over cached files.

Such hierarchical caching mechanisms can create three levels of inconsistency. As described above, a cached group file can be inconsistent with the original data if the original changes before the cached copy expires. However, consider the case of two concurrent users who are members of the same remote group accessing the same server. User A opens a directory protected with an ACL referencing a remote group. File caching is allowed, so the file is fetched and a lookup performed on the cache. Since decisions are also cacheable, the result is saved for some duration separate from that of the cached file. Later, user A performs another directory access requiring a lookup just before the cached file expires.

User B also opens a directory with remote group permissions. Since the cached group file is now expired, a new

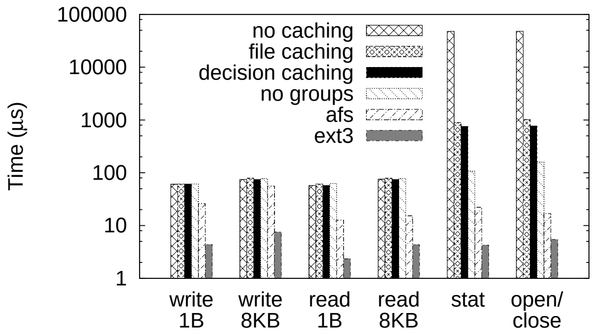

Figure 4. I/O Call Latency

The effect of caching on single I/O calls. Access control is performed only on name lookups, so operations on file descriptors are not affected by the mechanism. Operations on names such as `stat` *and* `open/close` *invoke the mechanism and suffer serious overhead without caching.*

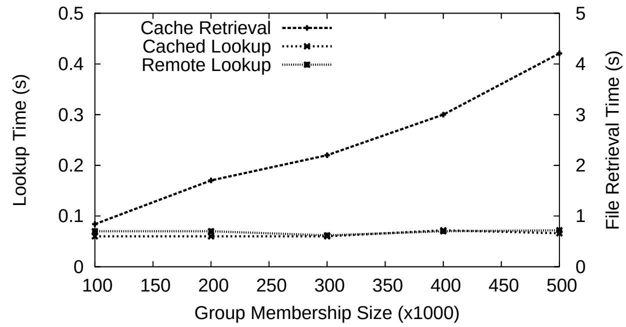

Figure 5. Group File Lookups

Performance of a single remote lookup over a LAN is comparable to that of a cached file, and lookup times do not increase substantially with the size of the group. If file caching is enabled, the time to transfer a file dwarfs the lookup times, and increases linearly with the size of the group.

copy is retrieved if the original had been modified. Any changes to the original group after this retrieval but before user A's cached decision expires would result in two separate caches, the file itself and user A's lookup result, neither of which are necessarily consistent with the original data. Consistency is restored upon expiration of all concurrent users' decision caches and the cached file. Expirations for different caching types are independent from one another and are based on data access rather than wall clock time.

3.3 Caching Policy

Primarily, mechanisms for caching group files and lookup decisions were implemented for scalability and performance. However, security policies may change so requesting servers must verify the group's caching policy with each group lookup. The caching policies of remote groups may be cached as well; doing such prevents bottlenecks on servers similar to those found in early implementations of NFS [17] and improves performance over wide area networks.

Upon initial reference to a remote group in an ACL, servers retrieve the caching policy from that server and inserts it in an index. Subsequent references to that remote groups require verification of policy before any other caches are checked. The cached policy expires when all of the cached components expire. For instance, if a decision cache is expired, and a cached group file cannot be checked because it too has expired, the policy is no longer considered valid and must be reverified. Missing or expired policy data requires an RPC to the appropriate server for retrieval or revalidation. Included in policy cache entries are the durations of either decision or file caches, or both, for each remote group.

4. Evaluation

This section presents an evaluation of the effects of caching on execution time for remote I/O calls and common file system operations to show that a file system with remote groups does not significantly underperform similar file systems without them with caching enabled. Experiments were run using 2.8 GHz Pentium 4 workstations running Red Hat Enterprise Linux connected via 1Gb/s switched Ethernet as the system is currently deployed on a limited scale at the University of Notre Dame, and will be further evaluated as groups are added to the larger storage pool employed over a WAN. Group size is 300,000 users.

Figure 4 shows the effects of caching on network I/O operations using a file server with remote groups. We compare the performance of 1,000 iterations of remote file operations to that of unmodified AFS and ext3. While other file systems have buffer caching enabled and asynchronous writes, neither of which supported by the TSS protocol, the purpose is to compare file systems as commonly configured, even if the comparison is unequal. In most cases, average time is within an order of magnitude of both AFS and ext3 despite user-level implementation. Network latency dominates `stat` and `open/close` calls, but caching improves performance by nearly two orders of magnitude, and within an order of magnitude of the file system without groups.

Figure 5 shows the time required to perform lookups remotely and the time to search for an entry in a cached file for groups of various sizes. Figures do not reflect latencies over a wide area network, as the current scope of system deployment is limited to a storage pool at Notre Dame. In both cases lookup times are dwarfed by the time to transfer and cache a large group file, which increases linearly with the size of the group. However, the file transfer time is a one-time penalty when caching the entire group. If no changes

Caching Configuration	Exec Time (s) (mean)	Standard Deviation
None	91.912	1.392
File	12.536	0.486
Decision	10.470	0.083
No Groups	8.269	0.030

Table 1. Andrew Benchmark Performance

The effect of caching on system performance for the Andrew benchmark. Execution time is reduced significantly by caching group files and lookup decisions, and is competitive with that of a system without remote groups.

occur, the revalidation at cache expiration is a simple RPC between servers.

Finally, Table 1 shows the effects of different caching policies on system performance for the Andrew benchmark [13]. Five trials were ran for each caching configuration, and the table shows the mean execution time for all five stages. With no caching enabled, each group lookup requires spawning a new process and an interserver RPC, which account for the bulk of the overhead.

5. Related Work

Many systems, both centralized and distributed, have implemented some form of grouping within a closed administrative domain. Most modern operating systems allow the administrator to create arbitrary groups of local users, but do not afford the end-user this facility. The Network Information Service (NIS) allows multiple machines to share the same password and group databases defined by a single administrator. The AFS [13] distributed file system allows end users to create arbitrary groups and ACLs using Kerberos principals in one realm. The NeST [3] storage appliance allows the construction of user-defined groups local to one storage devices. CURE [12] gives end users a graphical interface for constructing and applying user groups. In each of these cases, there is no facility for employing or sharing information *outside* of the closed system. In addition, the tradeoffs between consistency, availability, and performance are fixed by the administrator. For example, the propagation delay of all information in NIS is fixed by the master server, while group memberships in AFS are determined at login and discarded at logout.

Many grid computing systems are designed to live within the constraints of an ordinary Unix filesystem, and thus require external grid identities generated by GSI [7] to be mapped to a single local Unix identity. There are many ways to accomplish this: Globus [6] relies on a "gridmap" file to map grid identities to Unix accounts; Grid3 [10] maps multiple grid identities to group-shared Unix accounts; Legion [14] maps grid identities to fresh anonymous Unix accounts; In each case, a grouping decision is performed once for a session, and then the user is restricted to the (limited) sharing models available in a Unix filesystem.

The grouping decision in many grid systems relies on users (or the user's agent) to present credentials to a distinct authorization service, which then generates the local account or new grid credentials to be used on the resource of interest. Examples of this approach include CAS [16] and VOMS [1]. This notion is roughly equivalent to the ticket-granting service in Kerberos [18]. Our work is compatible with this approach to authorization: one may easily create an ACL that requires the user to present a certificate created by an authorization service. However, the sense of the control flow is different. In the ticket-granting approach, the user must identify and consult the authorization service before proceeding to the resource. In our work, the user proceeds directly to a resource, which then implicitly references one or more remote groups as needed.

Two systems are most closely related to our work. Kaminsky et al., describe additions to the Self-Certifying File System (SFS) that allow for group definitions shared across administrative boundaries [15]. Each organization in a system operates a group server with a local policy for the modification of groups. Once per hour, all servers exchange all data with each other. This system represents one design point whereby system administrators choose the tradeoffs between performance, consistency, and availability. Our work builds upon this by empowering end users with abilities reserved for administrators. Similarly, Grapevine [4] allows for the construction of distributed expansion lists enabling the delivery of mail to many recipients. As in SFS, the system designers chose to emphasize availability over consistency in a fixed manner.

6. Challenges and Future Work

Currently, nested groups are not fully supported. Each group should be able to contain any number of subgroups, to include groups defined both locally and remotely. This approach requires an efficient lookup mechanism as on-demand remote lookups on an arbitrary number of nested groups can become quite expensive. However, traversing all subgroups is not necessary in all cases. Because all members of a group and its subgroups share the same permissions, once a subject is found, no further searching is necessary. However, there should be a limit to the total time spent on each lookup. Furthermore, the system must carefully handle loops in group references.

Failure semantics must be further defined. How does one determine acceptable wait times when one server queries a remote group, and who should make that decision? The end user may control the application-level timeout via the adapter, but one server may still become stuck waiting for another to respond, even if the calling user gives up. These concerns lead to the possibility of varying strictness in enforcing policy. If a cached group file exists but is expired, we may be able to continue to use it with a "best effort" approach if the original data is unreachable when revalidating. A new expiration time placed on it may be based on the original policy or perhaps a separate policy. However, the resource owner ultimately controls the mechanism and may demand strict policy adherence.

7. Conclusions

In this paper we have presented the mechanisms for implementing security policies for user groups in a distributed system. Policies are determined by resource owners and offer several tiers of data protection. Policies can be enabled to allow maximum data privacy and consistency, albeit with a significant cost of performance and scalability, by only allowing remote group lookups with no caching. Using remote lookups with cached decisions trades off some privacy and consistency and offers good performance, but may not be scalable. Caching group files and lookup decisions costs more in terms of privacy (due to the possibly widespread distribution of group files to many servers) and consistency, but allows for better scalability by reducing the average workload on the servers hosting the group files. This is particularly the case as the number of users connecting to those servers simultaneously grows large.

Implementing groups in a grid-enabled file system relieves resource owners from having to frequently maintain a large number of possibly very large ACLs. More significantly, however, it also provides mechanisms for implementing security policies determined solely by resource owners and may be changed at any time. These security policies facilitate the sharing of storage resources by giving owners the flexibility to efficiently share a precise set of resources to a precise set of collaborators from any location.

References

[1] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, A. Gianoli, K. Lõentey, and F. Spataro. VOMS, an Authorization System for Virtual Organizations. In *1st European Across Grids Conference*, February 2002.

[2] 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.

[3] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[4] A. Birrell, R. Levin, M. Schroeder, and R. Needham. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4), April 1982.

[5] C. Ellis, S. Gibbs, and G. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[6] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[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] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. to appear in *International Journal of Supercomputer Applications*, 2001.

[9] Free Software Foundation, Inc. GNU dbm, Release 1.8, 1999. http://www.gnu.org/software/gdbm/gdbm.html.

[10] R. Gardner and et al. The Grid2003 production grid: Principles and practice. In *IEEE Symposium on High Performance Distributed Computing*, 2004.

[11] C. Gunter and T. Jim. Generalized Certificate Revocation. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–329, New York, NY, USA, 2000. ACM Press.

[12] J. Haake, A. Haake, T. Schümmer, M. Bourimi, and B. Landgraf. End-user Controlled Group Formation and Access Rights Management in a Shared Workspace System. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, pages 554–563, 2004.

[13] 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.

[14] M. Humphrey, F. Knabe, A. Ferrari, and A. Grimshaw. Accountability and control of process creation in metasystems. In *Network and Distributed System Security Symposium*, February 2000.

[15] M. Kaminsky, G. Savvides, D. Mazieres, and M. Kaashoek. Decentralized user authentication in a global file system. In *Symposium on Operating Systems Principles*, pages 60–73, 2003.

[16] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2002.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of USENIX 1985 Summer Conference*, pages 119–130, Portland OR (USA), 1985.

[18] 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.

[19] 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, Networking, and Storage (Supercomputing), November 2005.

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

[21] D. Thain, C. Moretti, P. Madrid, P. Snowberger, and J. Hemmes. The Consequences of Decentralized Security in a Cooperative Storage System. In *Proceedings of the 3rd IEEE Security in Storage Workshop*, San Francisco, CA (USA), December 2005.