The PUNCH Virtual File System: Seamless Access to Decentralized Storage Services in a Computational Grid

Renato J. Figueiredo
figueire@purdue.edu

Nirav H. Kapadia
kapadia@purdue.edu

José A. B. Fortes
fortes@purdue.edu

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285, USA

Abstract

This paper describes a virtual file system *that allows data to be transferred on demand between storage and compute servers for the duration of a computing session. The solution works with unmodified applications (even commercial ones) running on standard operating systems and hardware. The virtual file system employs software proxies to broker transactions between standard NFS clients and servers; the proxies are dynamically configured and controlled by computational grid middleware. The approach has been implemented and extensively exercised in the context of the Purdue University Network Computing Hubs, an operational computing portal that has more than 1,500 users across 24 countries. Results show that the virtual file system performs well.*

1. Introduction

This paper describes a *virtual file system* that allows data to be transferred on demand between storage and compute servers for the duration of a computing session. The solution works with unmodified applications (even commercial ones) running on standard operating systems and hardware. The virtual file system has been implemented and extensively exercised in the context of the Purdue University Network Computing Hubs — PUNCH [8, 10]. PUNCH is a platform for Internet computing that turns the World Wide Web into a distributed computing portal. It has been operational for five years, is currently used by about 1,500 users from 24 countries, and offers access to about 70 engineering applications.

The described virtual file system is built on top of an existing, de-facto standard that works across heterogeneous platforms — the Network File System (NFS [14]). The system works by way of software proxies that broker transactions between *standard* NFS clients and servers; the proxies are dynamically configured and controlled by computational grid middleware. Results show that the virtual file system performs well: average overheads of 1% and 18% were measured for two different computing environments.

This mechanism differs from related work in file-staging techniques (e.g., Globus [4] and PBS [6, 3]) in that it supports user-transparent, on-demand transfer of data. It differs from related on-demand data-access solutions for grid computing (e.g., Condor [12] and Legion [5, 15]) in that it does not require modifications to applications and it does not rely on non-native file system servers. Thus, the described virtual file system is unique in its ability to

provide on demand access to data for unmodified applications through native NFS clients and servers of standard operating systems.

The rest of this paper is organized as follows. Section 2 explains the concepts behind logical user accounts and motivates the design of a virtual file system. Section 3 discusses the ways in which a virtual file system can be implemented. Section 4 describes the PUNCH Virtual File System (PVFS). Section 5 discusses security and scalability issues in the context of PVFS, and presents a quantitative performance analysis. Section 6 outlines related work. Section 7 presents concluding remarks.

2. Logical user accounts

Today's computing systems tightly couple users, data, and applications to the underlying hardware and administrative domain. For example, users are tied to individual machines by way of user accounts, while data and applications are typically tied to a given administrative domain by way of a local file system. This causes several problems in the context of large computational grids, as outlined in [9]. (*Details will be provided in the final paper.*)

In order to deliver computing as a service in a scalable manner, it is necessary to effect a fundamental change in the manner in which users, data, and applications are associated with computing systems and administrative domains. This change can be brought about by introducing a layer of abstraction between the physical computing infrastructure and the *virtual computational grid* perceived by users. The abstraction layer can be formed by way of two key components: 1) logical user accounts, and 2) a virtual file system. A network operating system, in conjunction with an appropriate resource management system, can then use these components to build systems of systems at run-time [9].

This abstraction converts compute servers into *interchangeable parts*, thus allowing a computational grid to broker resources among entities such as end-users, application service providers, storage warehouses, and CPU farms. The described approach has been deployed successfully in PUNCH, which employs logical user accounts, a virtual file system, a network operating system, and a resource management service that can manage computing resources spread across administrative domains.

Logical user accounts are made up of two components: *shadow accounts* and *file accounts*. Shadow accounts consist of `uid`s on compute servers that are dynamically allocated to users when they attempt to initiate a run (or session) and reclaimed by the system after the run (or session) is complete [9]. File accounts are accounts on file servers that are used to store user files. A given file account typically stores files for more than one user, and the network computing system may move files across file accounts as necessary. Access to the files is brokered by the virtual file system; users *never* directly login to a file account [9]. (*Details will be provided in the final paper.*)

3. Virtual file system

A *virtual file system* establishes a dynamic mapping between a user's data residing in a file account and the shadow account that has been allocated for that user. It also guarantees that any given user will only be able to access files that he/she is authorized to access.

3.1. Explicit file transfers

A simple virtual file system could copy all of a user's files to a shadow account just before initiating a run (or session) and then copy the files back once the run (or session) is complete. This approach has two obvious disadvantages: it is likely to result in large amounts of unnecessary data transfer, and it would require a complex coherency protocol to be developed in order to support multiple, simultaneous runs (or sessions). A variation on this theme is to allow (i.e., require) users to explicitly specify the files that are to be transferred. This approach is commonly referred to as *file staging*. File staging works around some of the issues of redundant data transfer and coherency problems (both of which must now be manually resolved by the user), but is not suitable for 1) situations in which the user does not know which files will be required *a priori* (e.g., this is true for many CAD and other session-based applications), or 2) applications that tend to read/write relatively small portions of very large files (e.g., most database-type applications).

3.2. Implicit file transfers

Another possibility is to transfer data on demand. An approach previously deployed on PUNCH relies on system-call tracing mechanisms such as those found in the context of Ufo [1, 2]. Entire files still need to be transferred, but the process is automated. (The transfer is a side effect of an application attempting to open a file.) The disadvantages of this approach are that it is highly O/S-dependent, and it demands extensive programming effort in the development of system-call tracers and customized file system clients.

3.3. Implicit block transfers

The third option is to reuse existing file system capabilities by building on a standard and widely-used file system protocol such as NFS. There are three ways to accomplish this goal. One is to enhance the NFS client and/or server code to work in a computational grid environment. This would require kernel-level changes to each version of every operating system on any platform within the grid. The second approach is to use standard NFS clients in conjunction with custom, user-level NFS servers. This approach is viable, but involves significant software development. The third possibility is to use *NFS call forwarding* by way of middle-tier proxies. This approach is very attractive for two reasons: it works with standard NFS clients and servers; and proxies are relatively simple to implement — they only need to receive, modify, and forward standard remote procedure calls (RPC).

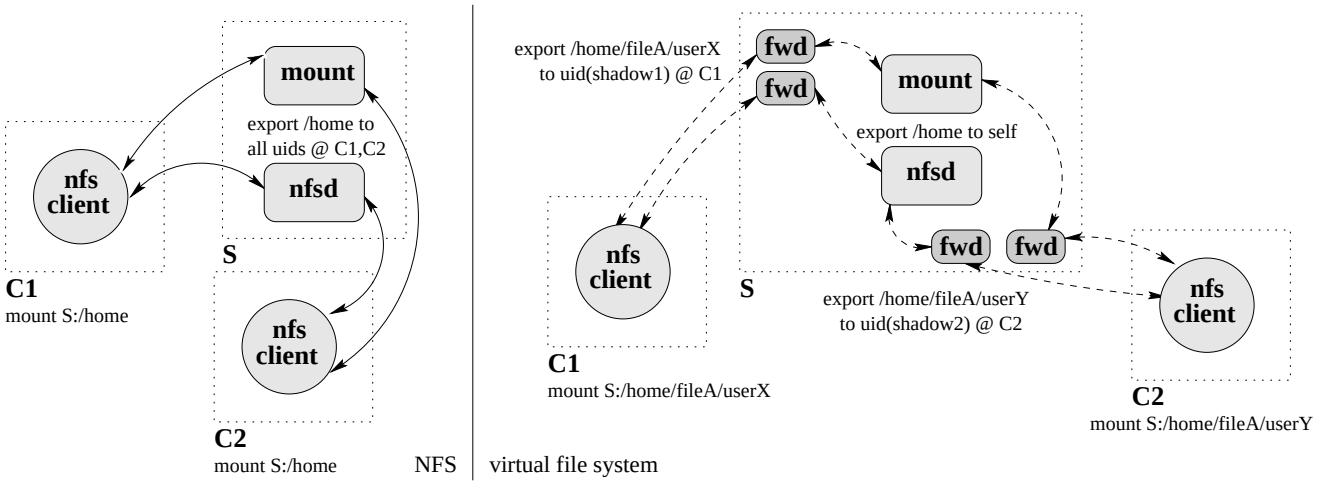

Figure 1. Overview of conventional and virtual file systems. The NFS clients C1, C2 (to the left) have a static mount point for all users under `/home`; the virtual file system clients (to the right) have dynamic mount points for users inside `/home/fileA` that are valid only for the duration of a computing session.

4. The PUNCH Virtual File System

The PUNCH Virtual File System — PVFS — is based on a call-forwarding solution that consists of three components: server-side proxies and file service managers, and client-side mount managers. The proxies control access to data in the various file accounts. The file service managers and mount managers together control the setup and shut down of virtual file system sessions.

Although it is based on a standard protocol, the virtual file system approach differs fundamentally from traditional file systems. For example, with NFS, a file system is established once on behalf of multiple users by system administrators (Figure 1, left). In contrast, the virtual file system creates and terminates *dynamic* client-server sessions that are managed by the grid middleware; each session is only accessible by a given user from a specified client, and that too only for the duration of the computing session (Figure 1, right). The following discussion outlines the sequence of steps involved in the setup of a PVFS session.

When a user attempts to initiate a run (or session), a compute server and a shadow account (on the compute server) are allocated for the user by PUNCH's *active yellow pages* service [11]. Next, the file service manager spawns a proxy daemon in the file account of the server in which the user's files are stored. This daemon is configured to only accept requests from one user (Unix uid of shadow account) on a given machine (IP address of compute server). Once the daemon is configured, the mount manager employs the standard Unix "mount" command to mount the file system (via the proxy) on the compute server.

Once the PVFS session is established, all NFS requests originating from the compute server by a given user (i.e., shadow account) are processed by the proxy. For valid NFS requests, the proxy modifies the user and group

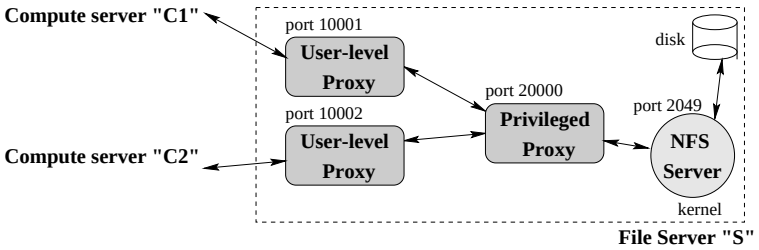

Figure 2. Implementation of proxy-based NFS virtual file system model currently deployed in PUNCH.

identifiers of the shadow account to the identifiers of the file account in the arguments of NFS remote-procedure calls; it then forwards the requests to the native NFS server. If a request does not match the appropriate user-id and IP credentials of the shadow account, the request is denied and is not forwarded to the native server.

Figure 2 depicts an example of the configuration of PVFS in the scenario of Figure 1 (right), where user "X" is allocated shadow account "shadow1" in machine "C1" and user "Y" is allocated shadow account "shadow2" in machine "C2". In the file server "S", two user-level proxy daemons (listening to ports 10001, 10002) authenticate requests from the two clients. They map shadow1/shadow2 to the `uid` of the PUNCH file account "fileA", and forward RPC requests to the kernel-level server via a privileged proxy (which listens to port 20000)[1]. The figure also shows the mount commands issued by the compute servers "C1" and "C2", under the assumption that the PUNCH file account has user accounts laid out as sub-directories of `/home/fileA` in file server "S". The path provided to the mount command ensures that userX cannot access the parent directory of `/home/fileA/userX` (i.e., this user cannot access files from other users).[2]

5. Security, scalability, and performance

The security implications of PVFS are tied to whether or not it spans multiple administrative domains. When the system is deployed within a single administrative domain, it co-exists with the native NFS services. Since the NFS services are not modified or reconfigured in any way, and PVFS implements a more restrictive access control model on top of NFS, existing security is not disturbed.[3] (*Details will be provided in the final paper.*)

When PVFS is deployed across administrative domains, it becomes necessary to preserve the *limited-trust* relationship between the nodes of the computational grid. For example, when a user belonging to administrative domain 'A1' is allocated a compute server 'C2' in a different domain 'A2', PVFS will map the user's data from the file server(s) in 'A1' to 'C2'. At this point, 'C2' has access to the specific user's data for the duration of the computing session. To the extent that 'C2' has access to user data, and the user has access to 'C2', there is a *trust*

[1] A privileged proxy is employed to allow a PVFS server configuration that co-exists with conventional LAN setups. Details about this configuration will be provided in the final version.

[2] Tighter access control can be introduced on the server side by using a proxy for the mount protocol. This proxy would negotiate mount requests in the same manner that the NFS proxy negotiates file system transactions.

[3] The implications associated with employing logical accounts are discussed in [9].

relationship between 'A1' and 'A2'.[4] However, this trust between 'A1' and 'A2' is *limited* in the sense that 'A2' cannot gain access to files outside the user's account or to computing resources in 'A1' via PVFS — even if `root` on 'C2/A2' is compromised. This is a consequence of the fact that 'A1' controls the mount point *and* all NFS transactions via the server-side mount and NFS proxies.[5] (*Details will be provided in the final paper.*)

The scalability of PVFS can be evaluated in terms of its ability to support a growing number of users (or sessions), clients (i.e., compute servers), and file servers. Since PVFS only establishes point-to-point connections over TCP/IP, and because there is no communication between sessions at the PVFS level, the system scales simply by replicating the different components of PVFS appropriately. (*Details will be provided in the final paper.*)

The performance of PVFS can be measured in terms of its impact on the transactions/second with respect to native NFS. PVFS introduces a fixed amount of overhead for each file system transaction. This overhead is primarily a function of RPC handling and context switching; the actual operations performed by the proxy are very simple and independent of the type of NFS transaction that is forwarded.

The following performance analysis is based on the execution of the Andrew file system benchmark (AB [7]) on directories mounted through PVFS. This benchmark consists of a sequence of Unix commands of different types (directory creation, file copying, file searching, and compilation) that models a workload typical of a software development environment.

Machine	#CPUs	CPU type	Memory	L2 cache	Network	O/S
S	2	400MHz UltraSparc	1GB	2MB	100Mb/s	Solaris 2.7
C1	4	400MHz UltraSparc	2GB	4MB	100Mb/s	Solaris 2.7
C2	4	480MHz UltraSparc	4GB	8MB	100Mb/s	Solaris 2.7

Table 1. Configuration of server (S) and clients (C1, C2) machines used in the performance evaluation. All nodes are connected by a local-area switched ethernet network; C1 and C2 are in the same sub-network, while S sits on a different sub-network.

The analysis considers the execution of up to 4 independent, simultaneous instances of AB on each client machine 'C1' and 'C2' described in Table 1; in the largest experiment, 8 instances of AB are executed concurrently —- i.e., one instance in every CPU of the two 4-way multiprocessor clients. The AB benchmark performs both I/O and computation; thus its execution time is dependent on the client's performance. Since this experiment considers clients with different configurations, performance results are reported separately for each machine.

The file server 'S' is a dual-processor machine that is currently used by the Purdue-based PUNCH portal; the experiments are performed in a "live" environment where the file server is accessed from both regular PUNCH users and the benchmark. For each combination of client, server, and file system, 200 samples of AB executions were collected at 30-minute intervals over a period of four days.

[4] It is relatively simple to adapt PVFS to export user-specified directory sub-trees so as to limit the amount of data exposed to 'C2'.

[5] Mount proxies are not currently deployed in PUNCH because the setup is contained within a single administrative domain.

		Client C1				Client C2			
#AB	FS	Mean	Stdev	Min	Max	Mean	Stdev	Min	Max
1	NFS	18.2s	1.6s	16s	28s	14.1s	0.5s	13s	17s
1	PVFS	18.4s	1.9s	17s	27s	16.7s	1.4s	15s	29s
2	NFS	24.0s	2.1s	20s	30s	18.7s	1.4s	14s	24s
2	PVFS	22.8s	3.8s	20s	42s	21.4s	4.2s	17s	43s
4	NFS	29.8s	2.4s	24s	36s	25.6s	2.3s	21s	34s
4	PVFS	35.5s	10.3s	24s	85s	35.0s	8.3s	22s	72s
8	NFS	43.4s	4.7s	31s	72s	38.4s	4.4s	28s	53s
8	PVFS	55.4s	12.5s	29s	94s	56.5s	12.1s	29s	116s
8	PVFS-8	42.5s	4.3s	32s	57s	38.8s	4.8s	30s	65s

Table 2. Mean, standard deviation, minimum and maximum execution times (in seconds) of 200 samples of Andrew benchmark runs for native and virtual file systems (NFS, PVFS) measured at clients C1 and C2. The table shows data for 1, 2, 4, and 8 concurrent executions of AB (#AB). The average load of the 4-processor client, measured prior to the execution of the benchmark, is at most 0.04.

The results from the AB experiments are summarized in Table 2; NFS refers to the native network file system setup, and PVFS refers to the virtual file system setup with multiple server-side user-level proxies and a single privileged proxy. Both NFS and PVFS use version 3 of the protocol and 32KByte read/write buffers.

The single-client results show that the overhead introduced by the PVFS proxy is small: the average difference in execution times are 1% (C1) and 18% (C2). The overhead introduced by PVFS is only on the CPU-bound RPC-processing phase of an NFS transaction. With a fast server processor, this overhead is small with respect to the network and disk I/O components of NFS [9].

The multiple-client results show trends that indicate a degradation in the performance of PVFS: the average relative overhead increases with the number of clients (up to 47%). Furthermore, the PVFS standard deviation becomes larger relative to the average (up to 21%), and the ratio between PVFS and NFS maximum execution times increase to up to 2.19.

This performance degradation is due to the fact that the current implementation of the PVFS proxy is not multi-threaded, causing RPC requests to be serialized by the (single) privileged proxy. This was verified by repeating the 8-client experiment with 8 user-level and 8 privileged proxies. With this setup, the performance of PVFS was, on average, within 1% of the performance of the native NFS (PVFS-8 in Table 2).

The performance analysis reported here is conservative in the sense that it is based on a user load that is more demanding of the file system than the observed behavior of PUNCH users: the Andrew benchmark generates 100-

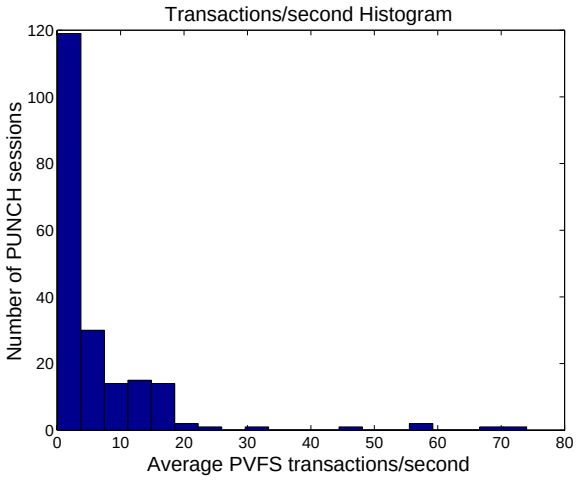

Figure 3. Histogram of file system transactions-per-second collected across 200 PUNCH user sessions. The total user time across these sessions is 78 hours.

130 transactions/second, whereas PUNCH sessions tend to have less than 20 transactions/second (see Figure 3).[6] (*A more detailed report will be provided in the final paper.*)

6. Related Work

Current grid computing solutions typically employ file staging techniques to transfer files between user accounts in the absence of a common file system. Examples of these include Globus [4] and PBS [6, 3]. As indicated earlier, file staging approaches require the user to explicitly specify the files that need to be transferred, and are often not suitable for session-based or database-type applications. Some systems (e.g., Condor [12]) utilize remote I/O mechanisms to allow applications to access remote files. This approach requires applications to be re-linked with special libraries, making it unsuitable for situations where object or source code is not available (e.g., as with commercial applications).

Legion [5, 15] employs a modified NFS daemon to provide a virtual file system.[7] From an implementation standpoint, this approach is less appealing than call forwarding: the NFS server must be modified and extensively tested for compliance and for reliability. Moreover, common user-level NFS servers (including the one employed by Legion) tend to be based on the older version 2 of the NFS protocol, whereas the call forwarding mechanism described in this paper works with version 3 (the current version).[8] Finally, user-level NFS servers generally do not perform as well as the kernel servers that are deployed with the native operating system.

The Self-certifying File System (SFS) [13] is another example of a virtual file system that builds on NFS.

[6] The PUNCH transactions/second represent data collected across 200 user sessions; data is continuously collected by PUNCH, and a larger number of samples will be used for the final paper.

[7] This approach has also been investigated in the context of PUNCH.

[8] The call forwarding mechanism also works with version 2 of NFS.

The primary difference between SFS and the virtual file system described here is that SFS introduces additional parameters in the NFS remote procedure call semantics. This allows SFS to provide encrypted channels, but results in a solution that uses messages that are not fully compliant with the NFS protocol specifications and requires both server-side and client-side proxies.

7. Conclusions

Employing a virtual file system makes it possible for a computing system to manage data independently of constraints imposed by user accounts and administrative domains. PUNCH employs a sophisticated virtual file system that leverages NFS to provide on-demand transfers at the granularity of file segments. This solution is non-intrusive (i.e., it co-exists with an unmodified local area NFS setup) and performs well.

PUNCH has employed a virtual file system since Fall of 1999. The virtual file system described in this paper has been in place since Fall of 2000. These mechanisms have been extensively exercised during normal use of PUNCH by its users. It has been found to perform well, with an average overhead of 18% or less over the native NFS.

Acknowledgements

This work was partially funded by the National Science Foundation under grants EEC-9700762, ECS-9809520, EIA-9872516, and EIA-9975275, and by an academic reinvestment grant from Purdue University. Intel, Purdue, SRC, and HP have provided equipment grants for PUNCH compute-servers.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The Ufo global file system. In *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, California, January 1997.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[3] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable Batch System: External reference specification. Technical report, MRJ Technology Solutions, November 1999.

[4] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Teucke. A resource management architecture for metacomputing systems. In *Proceedings of the Fourth Workshop on Job Scheduling Strategies for Parallel Processing*, 1998. Held in conjunction with the International Parallel and Distributed Processing Symposium.

[5] A. S. Grimshaw, W. A. Wulf, et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[6] R. L. Henderson and D. Tweten. Portable batch system: Requirement specification. Technical report, NAS Systems Division, NASA Ames Research Center, August 1998.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance of a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[8] N. H. Kapadia, R. J. O. Figueiredo, and J. A. B. Fortes. PUNCH: Web portal for running tools. *IEEE Micro*, pages 38–47, May-June 2000.

[9] N. H. Kapadia, R. J. O. Figueiredo, and J. A. B. Fortes. Enhancing the scalability and usability of computational grids via logical user accounts and virtual file systems. In *Proceedings of the Heterogeneous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, California, April 2001.

[10] N. H. Kapadia and J. A. B. Fortes. PUNCH: An architecture for web-enabled wide-area network-computing. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2(2):153–164, September 1999. In special issue on High Performance Distributed Computing.

[11] N. H. Kapadia, J. A. B. Fortes, M. S. Lundstrom, and D. Royo. Punch: A computing portal for the virtual university. *International Journal of Engineering Education*, 7(2), 2001. In special issue on Virtual Universities in Engineering Education.

[12] M. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[13] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, South Carolina, December 1999.

[14] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

[15] B. S. White, A. S. Grimshaw, and A. Nguyen-Tuong. Grid-based file access: The Legion I/O model. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 165–173, Pittsburgh, Pennsylvania, August 2000.