A User-level Secure Grid File System

Ming Zhao Renato J. Figueiredo
Advanced Computing and Information Systems Laboratory (ACIS)
Electrical and Computer Engineering, University of Florida
{ming, renato}@acis.ufl.edu

ABSTRACT

A grid-wide distributed file system provides convenient data access interfaces that facilitate fine-grained cross-domain data sharing and collaboration. However, existing widely-adopted distributed file systems do not meet the security requirements for grid systems. This paper presents a Secure Grid File System (SGFS) which supports GSI-based authentication and access control, end-to-end message privacy, and integrity. It employs user-level virtualization of NFS to provide transparent grid data access leveraging existing, unmodified clients and servers. It supports user and application-tailored security customization per SGFS session, and leverages secure management services to control and configure the sessions. The system conforms to the GSI grid security infrastructure and allows for seamless integration with other grid middleware. A SGFS prototype is evaluated with both file system benchmarks and typical applications, which demonstrates that it can achieve strong security with an acceptable overhead, and substantially outperform native NFS in wide-area environments by using disk caching.

1. INTRODUCTION

Distributed "Grid" computing systems have been successfully applied in several domains of science, providing for sharing of resources and data across administrative boundaries. A key challenge arising in such systems is data management - how to seamlessly provide data to applications and users in wide-area environments. In the absence of widely deployed grid-wide distributed file systems (DFSs), existing solutions are often based on explicit file transfer ("staging"), or require users to program applications with specific grid-enabling APIs. Nonetheless, a grid-wide file system can facilitate data access and sharing by exposing familiar interfaces of local area DFSs (such as NFS [41][7][40]) to users. It is also desirable for applications that cannot be modified, require implicit data access, have complex access patterns, operate on large and sparse data sets, or require fine-grained data sharing, because data transfers can be performed on-demand, on a per-block basis.

Security is one of the most important concerns for data management in grid environments, where data are shared across organizations with limited mutual-trust, and stored and transferred on resources with limited security. Providing secure grid-wide data access is a challenging task with existing DFSs. In a grid system, virtual organizations are dynamically established, applications and services are dynamically initiated, and entities and trust are dynamically created. Conventional DFSs are not capable to meet this challenge, because they are designed for general file system usage (typically for LANs), and favor static, homogeneous configurations – rather than the dynamic environments encountered in grid deployments.

Nonetheless, recent work has shown the feasibility of applying user-level techniques to build wide-area file systems on top of existing kernel implementations [34][16]. Examples of systems that use NFS to mount grid data are found in the middleware of Legion [43], PUNCH [26], and In-VIGO [1]. This paper proposes such a user-level solution that addresses the aforementioned challenges with a Secure Grid File System (SGFS). It enables secure network communications based on mature technologies (SSL/TLS [17][12]), and employs widely-accepted security tokens (X.509/GSI certificates [42]) to provide compatible grid authentication and flexible access control. SGFS allows data sessions to be created on a per-user or per-application basis, and such sessions can be customized with respect to the security policies and mechanisms. Furthermore, it leverages service-based middleware with standards-conforming security (WS-Security [54]) to manage and configure the sessions.

Overall, the proposed approach makes the following contributions: 1) it achieves strong security for grid-wide file systems; 2) it leverages user-level techniques that support unmodified applications and operating systems; 3) it supports flexible selection of security configurations for file systems based on user and application needs; 4) it conforms to the grid security infrastructure (GSI) and therefore can be easily integrated with other grid middleware and systems.

The paper evaluates an implementation of SGFS with file system benchmarks (IOzone and PostMark), and applications capturing the behavior of both interactive access to data in a development environment (MAB) and scientific computing that exhibits a mix of CPU and I/O activity (Seismic). Experiments were conducted in a LAN to study the overhead from the user-level techniques, and also in an emulated WAN setup which captures the target environment for SGFS. Results from this analysis demonstrate that the solution achieves strong security with reasonable overhead, and a tradeoff can be made to balance the performance and security strength for the file systems. It also shows that SGFS can effectively hide high network latencies using disk caching and deliver efficient data access in wide-area environments, which substantially outperforms native NFS.

SC07 November 10-16, 2007, Reno, Nevada, USA

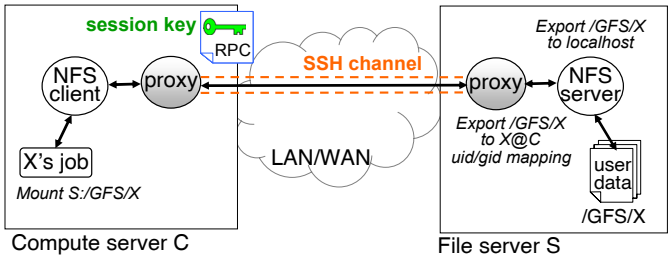

Figure 1 (up): A grid-wide file system can be built upon the virtualization of NFS. SSH tunneling and session-key authentication can be employed to provide security.

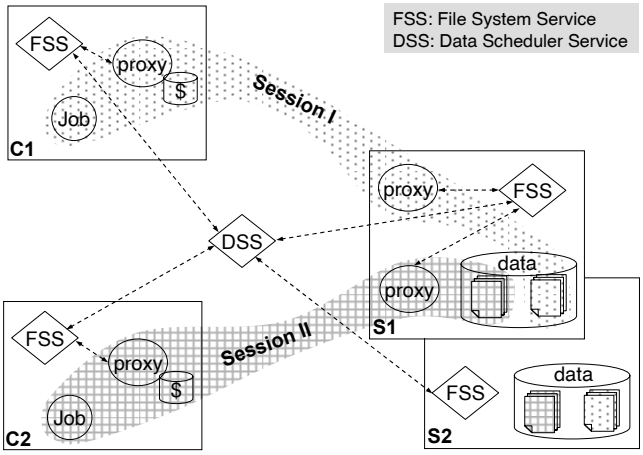

Figure 2 (right): GFS sessions can be dynamically created to provide on-demand grid data access. The management services are leveraged to control and configure the sessions.

The rest of this paper is organized as follows: Section 2 describes the background and related work; Section 3 and 4 explain the design and implementation details; Section 5 discusses the deployment, Section 6 presents the experimental evaluation and Section 7 concludes the paper.

2. Background and Related Work

2.1 File System Virtualization and Enhancements

Traditionally DFSs are designed for general file system usage, implemented in operating systems, and deployed by administrators at the granularity of a collection of users. There are no mechanisms that allow a DFS to be customized to support application- or user-tailored configurations. In contrast, user-level techniques can be leveraged to extend and enhance kernel file system functionality based on loop-back proxies - in essence, virtualizing DFSs by means of intercepting RPCs (Remote Procedure Calls) of protocols such as NFS.

A grid-wide file system (GFS) can be built upon the virtualization of NFS [16][45], and Figure 1 shows such an example. The native NFS server exports the shared filesystem */GFS/X* to the localhost, and users outside the localhost can only mount it via the server-side proxy. The proxy inspects every incoming RPC request, checks the message's user credentials against a GFS exports file, and then forwards the authorized request to the NFS server to complete the data access. Meanwhile, user credential mapping is also performed by the proxy between the account where *X*'s job is running and the one where *X*'s files are stored.

GFS sessions can be dynamically created on a per-application/user basis (Figure 2). Each session can be customized individually according to the data access requirements or characteristics on a variety of aspects, e.g. the use of disk caching and its parameters. Service-base middleware can be used to manage the life-cycles and configurations of the sessions [44].

Related work has studied using user-level techniques to improve a variety of aspects of DFSs: The Sun Automounter [8] mounts file systems when they are referenced; CFS [3] supports a cryptography file system; Alex [10] enables FTP browsing via a file system interface; Kosha [6] implements peer-to-peer routing; Pangaea [38] supports aggressive replication; [46] enables application-tailored cache consistencies; Pond provides an NFS interface to OceanStore [37]; and LegionFS [43] provides a file system interface to Legion [20]. While related to these efforts, this paper focuses on applying user-level security mechanisms to address the limitations of kernel DFSs in order to provide a secure grid-wide file system.

2.2 Secure Distributed File System

Existing DFSs have diverse security designs and strengths. Earlier versions (V2 [41] and V3 [7]) of NFS rely on UNIX-style authentication, with user and group IDs. Although stronger authentication flavors are defined in the specifications, they have not prevailed in deployments. There is also no support for privacy and integrity in these versions, and NFS RPC messages can be easily spoofed, altered and forged. Strong security has not been available until the latest version (V4 [40]), which mandates the support of the RPCSEC_GSS flavor [14]. RPCSEC_GSS provides RPC-layer security based on the GSS-API [30], and a conforming NFS V4 implementation must support two security mechanisms, Kerberos V5 [31] and LIPKEY [13].

All NFS versions use an exports file to specify the hosts that are allowed to access an exported directory. The ACCESS procedure call was introduced in NFS V3 to provide fine-grained access control using POSIX-model ACLs, but again it is not widely used in practice. NFS V4 improves upon this by providing Windows NT-model ACLs which have richer semantics and wider deployments. In addition, NFS V4 represents users and groups with string IDs instead of integers, which facilitates cross-domain identity mapping.

Another important family of DFSs, Andrew File System (AFS [39]) and its successors (OpenAFS [48] and Coda [5]) use Kerberos-based systems to provide strong security. Access control is achieved by associating an ACL with directories that list positive or negative rights for a user or group. Kerberos relies on centralized control and works well within an Intranet. But cross-domain security is difficult to set up because it requires the involved administrations to negotiate a trust relationship.

None of these conventional DFSs has been designed to support grid security requirements. There is also related work on extending DFS security at kernel-level. In particular, the GridNFS [21] project develops a GSI-compatible security in NFS V4. However, such a design requires kernel support that is difficult to

deploy across shared grid environments, and it faces the same limitations as kernel DFSs, that is, it is unable to employ per-user or per-application security configurations. In contrast, a SGFS-style user-level solution can support flexible customization of grid file systems based on individual application and user needs.

User-level techniques can achieve privacy and integrity of NFS through secure tunneling, where SSH or SSL can be leveraged to establish a secure end-to-end connection between the client and server for NFS traffic [4]. A secure tunnel multiplexed by users faces the same limitations as NFS, since RPC-layer mechanism is still required for authentication and authorization within the tunnel. A session-key based inter-proxy authentication can be used along with secure tunneling to provide security for grid file systems [45] (Figure 1). In this model, per-session SSH channels are created to ensure privacy and integrity of each file system session, while the client- and server-side proxies perform authentication and authorization using a session key dynamically created and securely distributed by middleware.

The key advantages of such an approach are in that existing RPC-based clients and servers can be reused without modifications, and it leverages mature security technologies. However, it requires additional middleware to set up tunnels and keys, and its performance also suffers from the overhead of double user-level forwarding. In addition, it is not compatible with grid security standards, which presents a hurdle to the interoperability with other grid middleware. The proposed SGFS inherits the merits of this approach and addresses its limitations by protecting RPC communication directly with SSL, without the addition of tunneling, and uses widely-accepted grid security tokens to provide compatible authentication and authorization.

Self-certifying File System (SFS [34]) also leverages user-level loop-back client and server to enhance DFS security. It addresses the problem of mutual authentication between a file server and users by providing self-certifying pathnames for files. Such a pathname has the server's public key embedded inside, which is used by a client to verify the authenticity of the server, and then create a secure channel to protect the file system traffic. SFS is then extended to provide decentralized access control, in which users are allowed to create file sharing groups with ACLs in the file system [25]. When a user tries to access a file, the authentication server fetches the user's credentials and uses them along with the ACL to authorize the access. Compared to SFS, the proposed SGFS focuses on providing data access that meets the grid security requirements, and supports per-user/application file system customization.

2.3 Grid Security Standards

In [42] several key requirements were studied for a grid security model, including the support for multiple security mechanisms, dynamic creation of services, and dynamic establishment of trust domains. This research resulted in a de facto grid security standard, GSI, which is built upon the Public Key Infrastructure (PKI). In PKI-based grid security, a public key certificate (e.g. X.509 [22]) along with its associated private key uniquely identifies a grid user and is used for authentication. The certificate is often validated by checking the signature of its issuer, a trusted party known as a certificate authority (CA). Then the user identity is checked against certain access control mechanism (e.g. gridmap file in GSI, MayI layer in Legion [15]) for authorization. In addition, public key technologies can also encrypt and digitally sign a message in order to protect its privacy and integrity. Another important grid security requirement is delegation, which allows a service to act on behalf of a user. This can also be supported with extensions to public key certificates, e.g. proxy certificates in GSI and credentials in Legion.

Grid security can be implemented at two different levels. Transport level security [17][12] uses public key certificates to create a secure socket layer connection between two end-points and protect the data exchanges between them. It is a mature technology that has high-performance implementations (e.g. OpenSSL [49]), but it lacks service-level semantics and does not work for multi-hop connections. Message level security is a suite of standards arising from the emerging Web service technologies [54][52][53], which provides security at the layer of SOAP messaging. It is agnostic to transport layer protocols and connections, and supports more service-level functionalities. However, its performance is not comparable to transport level security because XML processing is expensive and it lacks efficient implementations. In this paper, a two-level security architecture that exploits the advantages of both approaches is proposed for the SGFS-based grid data management.

In the related data management solutions, GSI-based GridFTP [2] provides API for programming grid data access, and RFT is a web service for reliable file transfer using GridFTP; Legion [20] is an object-based grid system, which employs a modified NFS server to provide access to file objects, and it also integrates GSI in Legion-G [24]; the Condor system [32] uses system call interception or application relinking to support remote I/O, and it also supports GSI in Condor-G [18]. This paper proposes a grid-wide file system with compatible security mechanisms with these efforts. It differentiates from and also complements them in that SGFS-based data sessions allow unmodified application binaries to access grid data using existing kernel clients and servers, and support application-tailored per-session customizations.

3. Design

This paper proposes a two-level security architecture for SGFS-based grid data management (Figure 3). It leverages transport level security to protect the file system traffic of SGFS, and employs message level security to secure the interactions with the management services. Both layers utilize widely-accepted security tokens (X.509/GSI certificates) to support grid user authentication and file access control. The rest of this section presents this architectural design, followed by the implementation details in the next section.

3.1 Secure Data Access

Secure data access in SGFS is provided by transport level security mechanisms, which enable an efficient secure end-to-end connection between client- and server-side proxies to protect RPC communications. In order to create a SGFS session for a grid user to access a file server, public key based user and server certificates are used to establish the mutual authentication between the proxies. (A user certificate can be the user's grid identity certificate, or a proxy certificate issued by the user that supports delegation.) After a successful authentication, a shared key is negotiated between the two parties and is used to encrypt

the SGFS traffic, while the integrity can also be provided using digital signatures or Message Authentication Code (MAC).

An authenticated user's certificate is used by the server-side proxy to make authorization decisions, i.e. whether to grant the user's access to the exported files. This is achieved using a grid-style ACL mechanism which associates file system access permissions with the grid user's identity in the certificate. Such an access control is provided with different granularities allowing for flexible selection per session needs. For an authorized data access request, necessary identity mapping is also performed by the server-side proxy so that the request can be successfully executed on the file server.

The choice of security mechanisms and policies is flexible and customizable per SGFS session, in order to satisfy different security requirements from users and applications. This is important because such configurations have implications on both security and performance. For example, if the data transferred by SGFS is not confidential, encryption can be avoided to improve the data access performance, while digital signatures can still be employed to protect its integrity. In contrast, for a SGFS session created for highly sensitive data, encryption must be enabled with strong ciphers which require substantial CPU cycles to compute.

3.2 Secure Management Services

SGFS can be directly used by grid users to enable flexible data sharing. In a large grid system, where many SGFS sessions are dynamically created, the management services offer better capability of controlling the sessions and coordinating the data access. As illustrated in Figure 2, the File System Service (FSS) runs on every client and server and controls the local proxies to establish and configure SGFS sessions; and the Data Scheduler Service (DSS) provides the scheduling and customization of sessions through the interactions with client- and server-side FSSs.

To create a SGFS session through the services, a grid user or a service that acts on behalf of the user needs to authenticate with DSS using the user's certificate. Authorization is also performed by checking the certificate against an ACL, or consulting a dedicated authorization service. An authorized user can then proceed to delegate the management services the right to create a SGFS session on behalf the user: the DSS uses the user's certificate to interact with the client- and server-side FSSs, which in turn configure the proxies to use this certificate to establish a secure file system session.

Security for service interactions is enabled at the message level (Figure 3). Despite the performance inefficiency, message level security offers great functionality at the service level, which is necessary for the management services to use and interact with other high-level services. These services are not in the critical path of grid data access; they are only involved infrequently when control is needed on a SGFS session, specifically, when creating, configuring and destroying a session. Therefore, the use of more expensive security mechanisms does not hurt an established SGFS session's I/O performance, and has negligible overhead compared to a session's lifecycle.

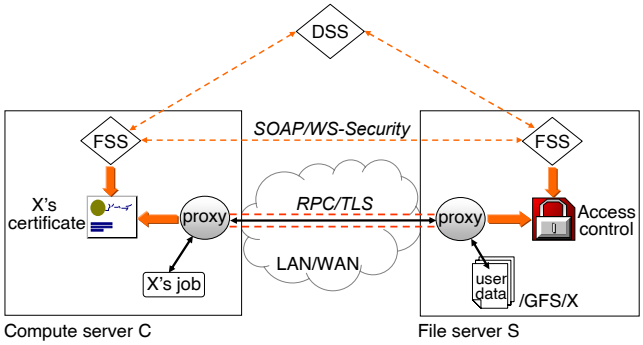

Figure 3: The security architecture of SGFS-based data management system. Transport level security is leveraged to protect the data access on SGFS, while message level security is employed to secure the interactions with the management services. Grid user certificates and ACLs are used for authentication and access control.

4. Implementation

4.1 SSL-enabled Secure RPC

Secure communication for NFS RPC is achieved using transport layer security protocols (SSL/TLS [17][12] - referred to generally as SSL). Although secure RPC can be realized at the RPC-layer itself (RPCSEC_GSS [14]), several factors have motivated the use of SSL: it has very mature and efficient implementations, which have been successfully employed by many important applications; it supports a wide range of algorithms, which can be leveraged to support flexible security configurations; SGFS sessions are established on per-user/application basis, and thus can use SSL to provide full-featured security without using any RPC-layer mechanisms.

A SSL-enabled secure RPC library has been developed for SGFS based on two key packages, TI-RPC [51] and OpenSSL [49]. TI-RPC (Transport Independent RPC) is the replacement for the original transport-specific RPC. It allows distributed applications to transparently support RPC over connectionless and connection-oriented transports for both IPv4 and v6. OpenSSL is an excellent implementation of SSL, and has recently included the support for datagram protocols (DTLS). Therefore, these tools can be effectively utilized to build a secure RPC library that supports both TCP and UDP.

In this library secure RPC APIs are provided in a way that closely resembles the regular RPC APIs. For example, *clnt_tli_ssl_create* and *svc_tli_ssl_create* are two expert-level APIs for creating a RPC client and server, respectively, using a secure transport for communications. These APIs take the same parameters as their regular counterparts with an additional one for the security configuration structure. The use of authentication, encryption and MAC as well as their specific algorithms can be specified through this structure and passed on to the library to create secure transports for RPC with the desired security mechanisms.

This secure RPC library is generic to support all RPC-based applications. The fact that both TI-RPC and OpenSSL are standalone packages helps its use by ordinary users without the need to change any system-level configurations. The current

implementation is based on Linux; support for other platforms is also conceivable.

4.2 GSI-based SGFS Proxy

The SGFS proxies are developed based on our previous work of virtual file system proxies [16][45]. They are enhanced to use the SSL-enabled secure RPC library for communications, and are also extended with the capability of parsing and validating GSI-based certificates. Using these proxies to establish a grid-wide file system, the privacy and integrity of data access are protected in the secure RPCs, while grid authentication and authorization are performed based on the user and server certificates.

A SGFS proxy is configured by a user or service through a configuration file, which is useful for customizing several important aspects of a SGFS session (e.g. the use of disk caching and its parameters). This configuration mechanism is augmented to include the security configurations, including the algorithms for authentication, encryption and MAC, and the paths to user, host and trusted CA certificates. In this way, both the client- and server-side proxies can be properly configured to use the grid user's and server's certificates to authenticate with each other, and set up a data session with the desired security mechanisms and policies.

A SGFS session's security customization can also be reconfigured by signaling the proxies to reload the configuration files. Such a dynamic reconfiguration is very useful in many important scenarios. For example, it can force a proxy to reload the certificate when the original one is expired or believed to be breached. It can reset a session's security setup when the desired configuration is changed. It can also be used to force a SSL-renegotiation and refresh the session key for a long-lived session. In fact, a proxy can be configured with a timeout value to enable periodical automatic renegotiation.

4.3 Grid File Access Control

After a successful mutual authentication, the grid user's certificate presented by the client-side proxy is cached by the server-side proxy and used for authorization of the data requests received from this session. The user credentials (UNIX user and group ID) in each NFS RPC message are from the client-side account allocated for a grid user or job. They do not represent the grid user's identity, and cannot be used for the purpose of authorization, but they are still necessary for the identity mapping. For each authorized RPC request, these credentials are mapped to a local user account's credentials, which are then used by the kernel NFS server to grant access to files. This authorization and mapping are determined by the grid file access control policies.

With GSI-enabled proxies, a variety of ACL mechanisms can be employed to enforce access control for SGFS sessions. The basic mechanism is based on a gridmap file similarly to GSI's, which provides access control per exported filesystem. This file describes the mapping between a user's grid identity (distinguished name) in the certificate and a local account's name. If a mapping exists for a user in the gridmap file, the user gains the same access rights to the exported filesystem as the corresponding local user. Otherwise, the user is mapped to an anonymous user, or denied access completely, depending on the session's configuration. In SGFS, the gridmap file can be set up on a per-session basis to enable flexible sharing. For example, if a user wants to share her files with another user, she only needs to add the mapping between that user's distinguished name and her local account name in the session's gridmap file.

Fine-grained access control is realized by leveraging the ACCESS procedure call available in NFS V3. Each file or directory can have an ACL file associated with it (under the same path and named in the style of *.filename.acl*). A user or service can grant or deny a user's access to a file or directory by putting the user's distinguished name inside the corresponding ACL file along with a bit mask encoding the access permissions. (Only the NFS V3 style ACL is supported in the current implementation.) Upon receiving an ACCESS request, the server-side proxy checks the user's grid identity against the requested file/directory's ACL, and returns the corresponding bit mask if the user is found in the ACL, or a zero which disables all access permissions.

A file or directory automatically inherits its parent's ACL if it does not have a dedicated ACL file. This inheritance mechanism can reduce the management complexity of ACLs. For the reason of performance, the ACLs are cached in memory by the server-side proxy once they are read from disk. The ACL files are protected by the server-side proxy from remote access, and can only be modified by the local owner of the files manually, or through an authorized service as explained later. Note that in the SGFS security model, the NFS server delegates the access control of the exported filesystems completely to the proxies. So the ACL mechanisms in kernel (except the kernel exports file) are no longer useful and should be disabled to avoid overhead.

4.4 GSI-based Management Services

The SGFS management services are implemented using WSRF::Lite [55], a Perl-based implementation of WSRF (Web Services Resource Framework [19]). This tool supports signing and verifying of SOAP messages using X.509 certificates according to the WS-Security standard, which is utilized to enable grid security at the service level. The resulting SGFS services can securely communicate with each other, use the grid user and server certificates to perform authentication and authorization, and then control the SGFS proxies to use these certificates to establish a secure file system session.

As mentioned earlier, the management services are responsible for creating and customizing SGFS-based data sessions on behalf of grid users or services. These operations are provided through web service interfaces, which conform to the WSRF standard; while the security of the service-level interactions also follows web service security standards and is compatible to GSI. This is important to provide interoperability with other grid services, e.g. to serve a job scheduler which needs to prepare data access for the jobs submitted to a grid resource.

The SGFS services support flexible grid file access control using the aforementioned mechanisms. Per-filesystem based ACLs are stored in the DSS database, and used to automatically create gridmap files that are used by the server-side proxies for the corresponding SGFS sessions. For fine-grained access control, the services manage the per-file/directory ACLs through the server-side proxies that are responsible for exporting the filesystems. In a large grid system, the access control to grid resources is often dedicated to the virtual organization's security service (e.g. a Community Authorization Service [36]). In this case, the SGFS services can consult this security service for file access control decisions based on individual grid users, or groups of users.

5. Deployment

SGFS can be conveniently deployed on grid resources because it does not require any modifications to either applications or kernels. It also obeys the least privilege principle in that the proxies and services work completely at user level and use unprivileged network ports, and they can be managed using a single regular user account (e.g. user *gfs*) on each host. On the server-side, the only privilege required is the configuration of a host-wide exports file used by the kernel NFS server. This can be restricted to a single entry in the exports file by organizing all the grid-accessible filesystems under a single path (e.g. */GFS*), which need only be exported to the localhost. On the client-side, the use of file system mount and un-mount is necessary, and it can also be minimized by giving only the local SGFS management account the permission to use sudo or a setuid program to mount and un-mount SGFS sessions to a restricted path (e.g. */GFS*).

To use SGFS (with or without the services), it is not necessary for a grid user to have a personal account on the client or server. The services and proxies create a secure file system session on behalf of the user between the account where her job is running and the account where her files are stored. These job and file accounts are often provided by mapping a grid user to a local user [42], or allocated on-demand for dynamically submitted jobs [16].

6. Performance

6.1 Setup

A prototype of the proposed SGFS is evaluated in this section with experiments. File system benchmarks (IOzone and Postmark) are used to investigate the overhead of achieving strong security under intensive I/O load. Benchmark applications modeling workloads in software development and scientific computing are also employed to study its performance with typical file system usages.

Both LAN and WAN environments are considered in the experiments. LAN-based runs study the overhead from the user-level techniques, while tests in an emulated WAN reveal its performance for the targeted grid environments. NIST Net [9] is used to emulate different wide-area network latencies. The file system client and server as well as the NIST Net router are set up on VMware-based virtual machines. They are hosted on separated physical servers connected via Gigabit Ethernet. Each physical server has dual 3.2GHz hyperthreaded Xeon processors and 4GB of memory. The client and server VMs are both configured with 1 CPU, and have 256MB and 768MB memory respectively.

The use of a network emulator and virtual machines facilitates the quick deployment of a controllable and replicable experimental setup. However, the timekeeping within a virtual machine may be inaccurate so the system clock on a physical server is used to measure time, which suffices the granularity required by this evaluation. All the experiments were conducted on virtual machines running on dedicated physical servers without interference from other workloads.

Different (secure) DFS setups are experimented, including:

Nfs-v3 and *nfs-v4*: The native kernel-level NFS V3 and V4 provide the baseline performance for comparison. Although not evaluated here, kernel-level secure NFS solutions (e.g. Kerberos-enabled NFS, GridNFS) can be expected to have worse performance than their results. Kernel NFS implementations use only memory for caching and revalidate the cached data when the file is reopened or its attributes have timed out. NFS-V4 can also provide delegation, which allows a client to aggressively cache data, but this feature is not yet widely supported.

Gfs and *gfs-ssh*: The basic Grid File System [16] without any security enhancements, and the SSH-enabled secure GFS [45]. Their results demonstrate the overhead from the user-level RPC processing and SSH tunneling.

Sfs: The related work of Self-certifying File System [34] - another NFS-based user-level secure file system. The overhead from the user-level techniques can also be observed from its performance. SFS aggressively caches attributes and access permissions in memory, which can improve metadata operations.

Sgfs: The approach proposed in this paper. By comparing to the above systems, the experiments examine the performance of the SSL-enabled strong authentication, privacy and integrity. Aggressive disk caching of attributes, access permissions and data are used in the WAN-based tests, so those results also reflect the potential performance improvement from that.

In all of the above setups, the server exports the filesystem with write delay and synchronous update, and the client accesses the server using TCP and 32KB read and write block sizes.

The experiments only consider file systems dedicated to a single user or job. For scenarios where multiple users/jobs share the data concurrently, different application-tailored cache consistency protocols can be applied in SGFS, which overlay upon the native NFS consistency mechanisms and provide improvements to both performance and consistency. A detailed discussion and evaluation can be found in [46].

All the experiment results are reported with the average and standard deviation values from multiple runs. Every run was started with cold client-side caches by unmouting the file system and flushing the disk cache.

6.2 Filesystem Benchmarks

6.2.1 IOzone

IOzone [35] analyzes a file system's performance by performing read and write operations on a large file with a variety of access patterns. In this experiment, it is executed on the client in read/reread mode, which sequentially reads a 512MB file twice from the server. Since the client has only 256MB of memory, the buffer cache doe not help with its LRU-based replacement for the benchmark's sequential reads. In fact, the client needs to read a total of 1GB data from the server during the execution. On the server side, the file is preloaded to the memory before each run, so there is no actual disk I/O involved in the tests. This "extreme" intensive setup reveals the worst-case overhead from SGFS' user-level virtualization and security enhancements.

The experiments evaluate various SGFS configurations that have different security strengths, as follows:

Sgfs-aes uses AES (Rijndael [11]) in CBC mode with 256bit key, a very strong cipher, to encrypt RPC traffic, and ensures integrity with SHA1-based HMAC [47].

Sgfs-rc uses RC4 (ARCFOUR [28]) with 128bit key, a "medium"-strength cipher for encryption, and also enables SHA1-HMAC for integrity.

Sgfs-sha does not use any encryption but still provides integrity using SHA1-HMAC.

To compare with SGFS, in *gfs-ssh* the SSH tunnels are configured to use 256bit AES-CBC and SHA1-HMAC, which is similar to the *sgfs-aes* configuration; SFS provides privacy and integrity using a customized RC4 and SHA1-HMAC, which is close to the *sgfs-rc* setup.

Figure 4 illustrates the runtimes of IOzone on the above DFS setups in LAN. The user-level file systems all show a slow down of more than two-fold compared to the kernel NFS implementations. However, such intensive workload is very rare in practice, and the user-level processing latency can often be overlapped with application "thinking" time or diminished by disk I/O latency. More importantly, in a WAN environment, the network latency becomes the dominant factor and renders the user-level latency negligible. User-level caching techniques can further hide the latency and improve the file system's performance. These discussions will be validated with the experiment in Section 6.2.2.

Comparing the different secure GFS configurations, *sgfs-sha* has the lowest overhead from the security enhancements (9% w.r.t. *gfs*), because it only calculates HMAC but does not perform any encryption/decryption on the file system traffic. With the use of encryption, the overhead is increased to 15% in *sgfs-rc*, and 50% in *sgfs-aes*. *Gfs-ssh* has a much higher overhead than the other ones (more than six-fold slowdown w.r.t. *gfs*). This can be at least partially attributed to the penalties from the double user-level forwarding: for every RPC message, two network stack traversals and kernel-user space switches are required by GFS and SSH to process it. As discussed earlier, such an overhead is magnified by this intensive experiment setup.

The proposed new SGFS approach removes this extra penalty, and thus improves the performance substantially. Notice that *sgfs-rc* is about 15% slower than *sfs*, which is due to our less efficient prototype implementation. In contrast to SFS' asynchronous RPCs, the use of blocking RPCs in SGFS prevents it from handling multiple outstanding requests simultaneously. A multithreaded implementation of SGFS is currently under development.

The experiment also measures the overhead of the user-level file systems in terms of CPU utilization. The user time percentages for GFS/SGFS proxies and SFS daemons were collected every 5 seconds throughout the benchmark's execution. The client- and server-side results are plotted in Figure 5 and 6 respectively. On the client, the basic GFS' CPU usage is very low, averaging 0.6% and under 1% for all the time. For SGFS, the utilization goes up to 5% with SHA1-HMAC, and further increased to about 8% when encryption/decryption is also used (256bit-AES consumes slightly more CPU than 128bit-RC4). On the server, the CPU usage is even less for *gfs*, *sgfs-sha* and *sgfs-rc*, averaging 0.3%, 1.5% and 3.6% respectively. All the SGFS configurations need less CPU than SFS which has more than 30% utilization on both sides.

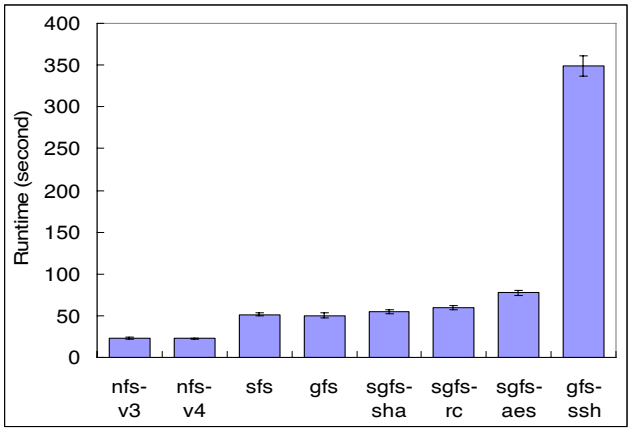

Figure 4: IOzone runtime on the different file system setups in LAN.

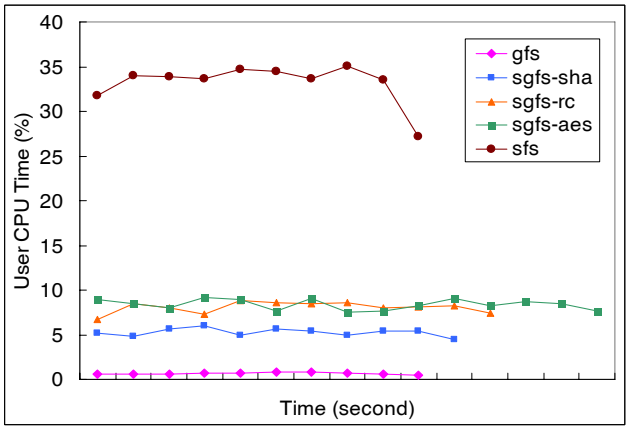

Figure 5: IOzone client-side CPU utilization of the user-level file system proxy/daemon.

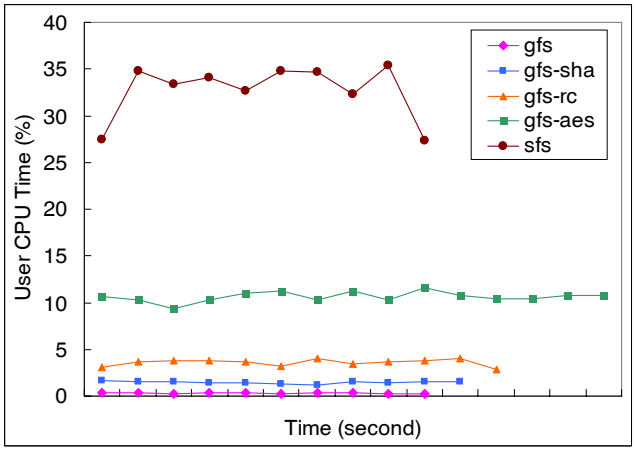

Figure 6: IOzone server-side CPU utilization of the user-level file system proxy/daemon.

6.2.2 PostMark

PostMark [27] is a more realistic file system benchmark that simulates the workloads from emails, news and web commence applications. It starts with the creation of a pool of directories and files (*creation* phase), then issues a number of transactions, including create, delete, read and append, on the initial pool (*transaction* phase), and finally removes all the directories and files (*deletion* phase). In contrast to the uniform, sequential data accesses used in the IOzone experiment, the file system is randomly accessed with a variety of data and metadata operations from PostMark.

In this experiment, the initial number of directories and files are 100 and 500 respectively, and the number of transactions is 1000. The transactions are equally distributed between create and delete, and between read and append. The file sizes range from 512B to 16KB, and thus the benchmark excises mostly on metadata operations and small writes.

Figure 7 shows the runtimes of each PostMark phase for the aforementioned DFS setups. The strong SGFS configuration *sgfs-aes* is used for the rest of this section, and is denoted as *sgfs* from here on. For the *creation* and *deletion* phases, the runtimes of the secure file systems are all very close to the native NFS' (*gfs-ssh* is marginally worse than the others). However, for the more intensive *transaction* phase, where a large number of small data and metadata updates are involved, only *sgfs* shows a close performance to NFS (V3), and it is better than *sfs* and *gfs-ssh* by 17% and 14% respectively.

The above experiment was conducted in a LAN environment, where the network round-trip time (RTT) between the file system client and server is about 0.3ms. Then it was repeated in the emulated WAN with different network latencies. Figure 8 compares the total runtimes of PostMark on *nfs-v3* and *sgfs*. Benefited from the use of disk caching, SGFS shows a very slow decrease in performance as the network latency grows. It is also significantly more efficient than native NFS in wide-area environments, and the speedup is about two-fold when the RTT is 80ms. These results prove our earlier discussions that a user-level secure file system can be very efficient for grid-scale systems.

Since no performance advantage has been observed in the version of NFS-V4 used in the experiments, only the results from NFS-V3 are reported here, as well as in the following experiments. Note that even though improved delegation support in V4 implementations may lead to better performance, SGFS provides the unique feature of application-tailored per-session customization.

6.3 Application Benchmarks

6.3.1 Software Development

This experiment models the typical software development process using a modified Andrew benchmark (MAB). It consists of four phases that exercise different aspects of a file system. The first phase (*copy*) makes a copy of a software source tree, which transfers a large number of small files within the file system. The second phase (*stat*) recursively examines the status of every file, and thus exercises metadata lookups intensively. The third phase (*search*) reads every file thoroughly to search for a keyword. The last phase (*compile*) compiles the entire source tree, which

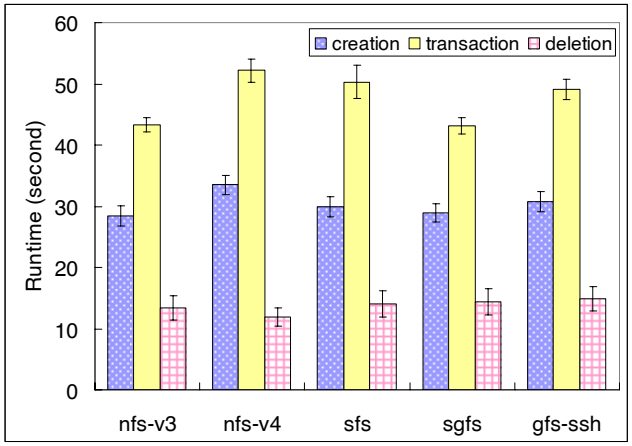

Figure 7: Runtime of each PostMark phase on the different DFS setups in LAN.

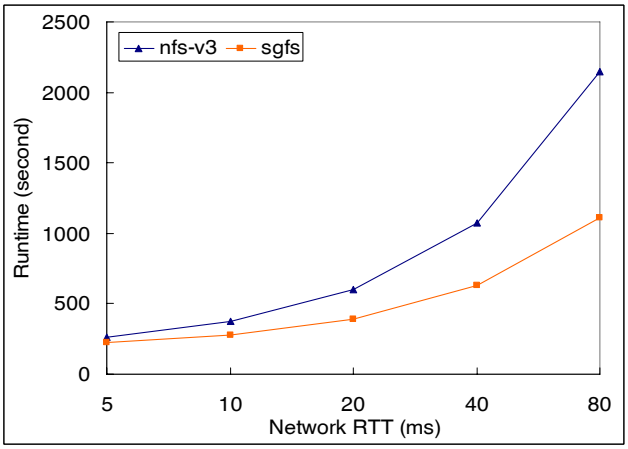

Figure 8: Total runtime of Postmark on NFS-V3 and SGFS in the different network setups.

generates a large number of data and metadata operations. Because the original Andrew benchmark [23] uses a workload that is too light for today's file systems, the source tree is replaced with the package of an OpenSSH client (*openssh-4.6p1*). It is a 3-level source tree with 13 directories and 449 files, and the entire compilation generates 194 binaries and object files.

The benchmark is executed on *nfs-v3* and *sgfs* in both LAN and emulated WAN (with 40ms RTT), and the results are shown in Figure 9. *Sgfs* performs as well as *nfs-v3* for the first three phases in LAN, and in the intensive *compile* phase, it shows a relatively small overhead of 14%. In WAN, SGFS caching effectively hides the network latency, and the total runtime of MAB is slowed down by only 2.5 times. Compared to *nfs-v3*, it is more than four times faster, and the speedup is approximately nine-fold, five-fold and eight-fold for the *stat*, *search* and *compile* phases respectively. Although not shown here, the performance of *sgfs* in LAN can also be improved if disk caching is used, in which the *compile* phase is only 2% slower than *nfs-v3*.

6.3.2 Scientific Computing

The second application benchmark uses a scientific tool, Seismic, which implements algorithms used by seismologists to locate

resources of oil. It is taken from the SPEC HPC96 suite [50], and its sequential version is used in the experiment. The execution consists of four phases: *data generation* (1), *data stacking* (2), *time migration* (3) and *depth migration* (4). Phase 1 prepares a large initial data file, and each of the following phases performs certain computation based on its previous phase's output file and then generates its own results on disk. In the end, the intermediate outputs are removed and only the results from the last two phases are preserved. This benchmark models a grid application that is both I/O and computation intensive.

This experiment was also conducted in both LAN and emulated WAN (with 40ms RTT). Based on the results shown in Figure 10, similar observations can be made as the previous experiment: in LAN, the performance of *sgfs* is very close to *nfs-v3*; in WAN, *sgfs* still delivers a good performance and is substantially better than *nfs-v3*. In phase 1, *sgfs* stores the large output entirely in cache with the use of write-back; in phase 2, a large number of reads can be satisfied from the data cached in disk, which are not available in memory; and at the end, *sgfs* also saves considerable time from writing back only the final results but not the temporary data to the server. Consequently, *sgfs* shows no slow down in WAN, and phase 2 in fact runs faster because disk caching is not enabled in LAN. Compared to *nfs-v3,* it is more than five times faster in the total runtime, and the speedup is about two-fold, forty-fold and four-fold for the first three phases respectively.

7. Conclusions

A grid-wide file system can provide users and applications with transparent access to grid data, but it must support strong security in order to be used in untrusted, cross-domain grid environments. Meanwhile, flexible, customizable security configurations are also desirable to meet different user and application needs. This paper presents a user-level secure grid file system (SGFS) that addresses these challenges. It employs SSL-enabled strong security, conforms to the GSI standard, supports flexible access control, and allows for customization of security mechanisms and policies. In addition, it leverages secure middleware services for the management of dynamic data sessions.

This paper shows that secure grid data management can be effectively achieved using user-level file system techniques. The experiments based on a SGFS prototype also demonstrate that strong security can be provided with an acceptable overhead. The tradeoff observed between security strength and performance overhead proves that an application-tailored security configuration is very important. The results also show that the use of disk caching can help SGFS to hide high network latencies and deliver efficient secure data access in the targeted wide-area environments.

This paper focuses on the security issues of network data access in grid systems; however, the data storage need also be protected from untrusted servers and administrators. Therefore, our future work will consider building user-level cryptographic functions into SGFS to ensure the privacy and integrity of data stored on the servers and provide a complete end-to-end grid data security.

The secure RPC library and SGFS prototype developed in this paper, as well as the Modified Andrew Benchmark used in the experiments, are available at http://www.acis.ufl.edu/~ming /software.

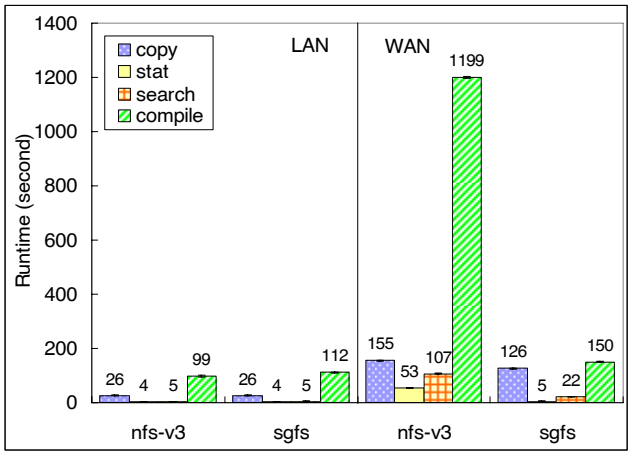

Figure 9: Runtime of each MAB phase on NFS-V3 and SGFS in both LAN and WAN. The time needed to write back data at the end of execution is 51.2s in average with a standard deviation of 1.3s.

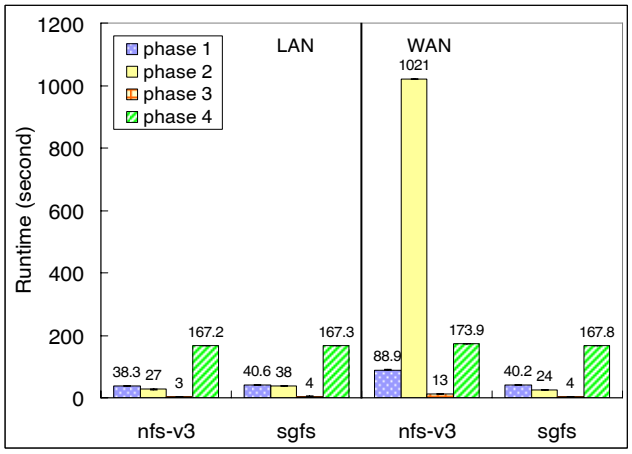

Figure 10: Runtimes of each Seismic phase on NFS-V3 and SGFS in both LAN and WAN. The time needed to write back data at the end of execution is 14.2s in average with a standard deviation of 1.3s.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This research is sponsored by NSF under grants EIA-0224442, EEC-0228390, ACI-0219925, ANI-0301108 and SCI-0438246. The authors also acknowledge the use of resources from the IBM SUR program and the Defense University Research Instrumentation Program (DURIP). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] S. Adabala et al, "From Virtualized Resources to Virtual Computing Grids: The In-VIGO System", Future Generation Computing Systems, Vol 21/6, 2005.

[2] B. Allcock et al, “Data Management and Transfer in High Performance Computational Grid Environments”, Parallel Comp Journal, Volume 28, Issue 5, May 2002.

[3] M. Blaze, “A Cryptographic File System for Unix”, 1st ACM Conference on Computer and Communications Security, November 1993.

[4] J. C. Bowman, “Secure NFS via SSH Tunnel”, URL: www.math.ualberta.ca/imaging/snfs/

[5] P. J. Braam, “The Coda Distributed File System”, Linux Journal, #50, June 1998.

[6] A. Butt et al, “Kosha: A Peer-to-Peer Enhancement for the Network File System”, Supercomputing, 2004.

[7] B. Callaghan, B. Pawlowski, P. Staubach, “NFS Version 3 Protocol Specification”, RFC 1813, Network Working Group, June 1995.

[8] B. Callaghan, T. Lyon, “The Automouner”, Winter 1989 USENIX Conference, pp 43-51, 1989.

[9] M. Carson, D. Santay, “NIST Net - A Linux-based Network Emulation Tool”, SIGCOMM Computer Communication Review, Volume 33, Issue 3, July 2003.

[10] V. Cate, “Alex - A Global Filesystem”, USENIX File System Workshop, May 1992.

[11] J. Daemen, V. Rijmen, “AES Proposal: Rijndael”, 1999.

[12] T. Dierks, E. Rescorla, “The Transport Layer Security (TLS) Protocol”, RFC 4346, April 2006.

[13] M. Eisler, “LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM”, RFC 2847, June 2000.

[14] M. Eisler, A. Chiu, L. Ling, “RPCSEC_GSS Protocol Specification”, RFC 2203, September 1997.

[15] A. Ferrari, F. Knabe, M. Humphrey, S. Chapin, A. Grimshaw, “A Flexible Security System for Metacomputing Environments”, 7th International Conference on High-Performance Computing and Networking, April 1999.

[16] R. J. Figueiredo, N. Kapadia, J. A. B. Fortes, “Seamless Access to Decentralized Storage Services in Computational Grids via a Virtual File System”, Cluster Computing, 7(2), April 2004.

[17] A. O. Freier, P. Karlton, P. C. Kocher, “The SSL Protocol Version 3.0”, Internet Draft, 1996.

[18] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, “Condor-G: A Computation Management Agent for Multi-Institutional Grids”, Cluster Computing, Volume 5, Issue 3, July 2002.

[19] I. Foster (ed) et al, “Modeling Stateful Resources using Web Services”, White Paper, March 5, 2004.

[20] A. S. Grimshaw, A. Ferrari, F. Knabe, M. Humphrey, “Wide-Area Computing: Resource Sharing on A Large Scale”, Computer, 32(5):29–37, May 1999.

[21] P. Honeyman, W. A. Adamson, S. McKee, “GridNFS: Global Storage for Global Collaborations”, Local to Global Data Interoperability - Challenges and Technologies, 2005.

[22] R. Housley et al, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile”, RFC 2459, January 1999.

[23] J. Howard et al, “Scale and Performance in a Distributed File System”, ACM Transactions on Computer Systems, Volume 6, Issue 1, 1998.

[24] M. Humphrey, “From Legion To Legion-G To OGSI.NET: Object-Based Computing For Grids”, 17th International Parallel and Distributed Processing Symposium, 2003.

[25] M. Kaminsky et al, “Decentralized User Authentication in a Global File System”, 19th ACM Symposium on Operating Systems Principles, 2003.

[26] N. H. Kapadia, J. A. B. Fortes, “Punch: An Architecture for Web-enabled Wide-area Network-Computing,” Cluster Computing, vol. 2, no. 2, 1999.

[27] J. Katcher, “PostMark: A New File System Benehmark”, Technical Report TR-3022, Network Appliance, 1997.

[28] Kalle Kaukonen, Rodney Thayer, “A Stream Cipher Encryption Algorithm ‘Arcfour’”, Internet Draft, 1999.

[29] J. Kubiatowicz et al., “OceanStore: An Architecture for Global-Scale Persistent Storage”, 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[30] J. Linn, “Generic Security Service Application Program Interface”, RFC 1508, September 1993.

[31] J. Linn, “The Kerberos Version 5 GSS-API Mechanism”, RFC 1964, June 1996.

[32] M. Litzkow, M. Livny, M. W. Mutka, “Condor: a Hunter of Idle Workstations”, 8th International Conference on Distributed Computing Systems, 1988.

[33] D. Mazières, “A Toolkit for User-Level File Systems”, USENIX Technical Conference, June 2001.

[34] D. Mazières, M. Kaminsky, M. F. Kaashoek, E. Witchel, “Separating Key Management from File System Security”, Symposium on Operating Systems Principles, 1999.

[35] W. Norcott, “The IOzone Filesystem Benchmark”, URL: www.iozone.org

[36] L. Pearlman, V. Welch, I. Foster, C. Kesselman, S. Tuecke, “A Community Authorization Service for Group Collaboration”, 3rd International Workshop on Policies for Distributed Systems and Networks, 2002.

[37] S. Rhea et al, “Pond: The Oceanstore Prototype”, 2nd USENIX Conference on File and Storage Technologies, 2003.

[38] Y. Saito et al, “Taming Aggressive Replication in the Pangaea Wide-area File System”, 5th Symposium on Operating Systems Design and Implementation, 2002.

[39] M. Satyanarayanan, “Scalable, Secure, and Highly Available Distributed File Access”, IEEE Computer, 23(5), May 1990.

[40] S. Shepler et al, “Network File System (NFS) Version 4 Protocol”, RFC 3530, April 2003.

[41] Sun Microsystems, Inc., “NFS: Network File System Protocol specification”, RFC 1094, March 1989.

[42] V. Welch et al, “Security for Grid Services”, 12th IEEE International Symposium on High Performance Distributed Computing, 2003.

[43] B. White, M. Walker, M. Humphrey, A. S. Grimshaw, "LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications", Supercomputing, 2001.

[44] M. Zhao, V. Chadha, R. J. Figueiredo, "Supporting Application-Tailored Grid File System Sessions with WSRF-Based Services", 14th IEEE International Symposium on High Performance Distributed Computing, July 2005.

[45] M. Zhao, J. Zhang, R. J. Figueiredo, "Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing", Cluster Computing, Volume 9, Number 1, January 2006.

[46] M. Zhao, R. J. Figueiredo, "Application-Tailored Cache Consistency for Wide-Area File Systems", 26th International Conference on Distributed Computing Systems, July 2006.

[47] "The Keyed Hash Message Authentication Code (HMAC)", FIPS Pub 198, National Institute of Standards and Technology, 2002.

[48] "Open Source Version of AFS", URL: www.openafs.org

[49] "OpenSSL: The Open Source toolkit for SSL/TLS", URL: www.openssl.org

[50] "SPEC HPC96 Benchmark Suite", URL: www.spec.org/hpc96

[51] "TI-PRC", URL: nfsv4.bullopensource.org/doc/tirpc_rpcbind.php

[52] "Web Services Secure Conversation Language", URL:www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf

[53] "Web Services Security Policy Language", URL:www6.software.ibm.com/software/developer/library/ws-secpol.pdf

[54] "WS-Security Specification", URL: www.oasis-open.org/specs/index.php#wssv1.0

[55] "WSRF::Lite", URL: www.sve.man.ac.uk/Research/AtoZ/ILCT