LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications

Brian S. White Michael Walker Marty Humphrey Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{bsw9d,mpw7t,humphrey,grimshaw}@cs.virginia.edu

ABSTRACT

Realizing that current file systems can not cope with the diverse requirements of wide-area collaborations, researchers have developed data access facilities to meet their needs. Recent work has focused on comprehensive data access architectures. In order to fulfill the evolving requirements in this environment, we suggest a more fully-integrated architecture built upon the fundamental tenets of naming, security, scalability, extensibility, and adaptability. These form the underpinning of the Legion File System (LegionFS). This paper motivates the need for these requirements and presents benchmarks that highlight the scalability of LegionFS. LegionFS aggregate throughput follows the linear growth of the network, yielding an aggregate read bandwidth of 193.8 MB/s on a 100 Mbps Ethernet backplane with 50 simultaneous readers. The serverless architecture of LegionFS is shown to benefit important scientific applications, such as those accessing the Protein Data Bank, within both local- and wide-area environments.

1. INTRODUCTION

Emerging wide-area collaborations are rapidly causing the manner and mechanisms in which files are stored, retrieved, and accessed to be re-evaluated. New, inexpensive storage technology is making terabyte and petabyte weather data stores feasible. Such data should be accessible physically close to the place of origin and by clients around the world. Companies are seeking mechanisms to share data without compromising the proprietary information of any involved site. Increasingly, clients desire the file system to dynamically adapt to varying connectivity, security, and latency requirements.

Accommodating the varied and continually evolving requirements of applications existing in these domains precludes the use of file systems that impose static interfaces or fixed access semantics. Common access patterns include whole-file access to large, immutable files and strided access to numerical, scientific datasets. The latter benefits from a non-traditional interface, though neither require typical file system precautions such as consistency guarantees. However, a file system catering only to these file access characteristics would be short-sighted, ignoring a possible requirement for additional policy, such as a stricter form of consistency. To service an environment which continues to evolve, a file system should be flexible and extensible.

Wide-area environments are fraught with insecurity and resource failure. Providing abstractions which mask such nuances is a *requirement*. The success of Grid and wide-area environments will be determined in no small part by its initial and primary users, domain scientists and engineers, who have little expertise in coping which the vagaries of misbehaved systems. Corporations may wish to publish large datasets via mechanisms such as TerraVision [27], while limiting access to the data. This is appropriate for collaborations that are mutually beneficial to organizations, which are, nevertheless, mutually distrusting. Such varied and dynamic security requirements are most easily captured by a security mechanism that transcends object interactions.

As resources are incorporated into wide-area environments, the likelihood of failure increases. A file system should relieve the user of coping with such failures. Approaches which require a user to explicitly name data resources in a location-dependent manner require that a user first locate the resource and later deal with any potential faults or migrations of that resource.

To address these concerns, we advocate a *fully-integrated* file system infrastructure. We have implemented the Legion [17] File System (LegionFS), an architecture supporting the following five tenets, which we consider fundamental to any system hoping to meet the goals delineated above:

- Location-Independent Naming: LegionFS utilizes a three-level naming scheme that shields users from low-level resource discovery and is employed to seamlessly handle faults and object migrations.
- Security: Each component of the file system is represented as an object. Each object is its own security domain, controlled by fine-grained Access Control Lists (ACLs). The security mechanisms can be easily configured on a per-client basis to meet the dynamic requirements of the request.
- Scalability: Files can be distributed to any host participating in the system. This yields superior performance to monolithic solutions and addresses the goals of fault tolerance and availability.
- Extensibility: Every object publishes an interface, which may be inherited, extended, and specialized to provide alternate policies or a novel implementation.
- Adaptability: LegionFS maintains a rich set of system-wide

SC2001 November 2001, Denver ©2001 ACM 1-58113-293-X/01/0011 $5.00

metadata that is useful in tailoring an object's behavior to environmental changes.

Previous work targeted at wide-area, collaborative environments has successfully constructed infrastructures composed of existent, deployed internet resources. Such approaches are laudable in that they leverage valuable, legacy data repositories. However, they fail to seamlessly federate such distributed resources to achieve a unified and resilient environment. A fully-integrated architecture adopts basic mechanisms (such as naming and security) upon which new services are built or within which existent services are wrapped. This obviates the need for application writers and service providers to focus on tedious support structure and allows them to concentrate on realizing policies within the flexible framework provided by the mechanisms.

LegionFS provides only basic functionality and is intended to be extended to meet the performance requirements of specific environments. The core of LegionFS functionality is provided at the user-level by Legion's distributed object-based system. The file and directory abstractions of LegionFS may be accessed independently of any kernel file system implementation through libraries encapsulating Legion communication primitives. This approach provides flexibility as interfaces are not required to conform to standard UNIX system calls. A modified user-level NFS daemon, lnfsd, interposes an NFS kernel client and the objects constituting LegionFS. This implementation provides legacy applications with seamless access to LegionFS.

This paper is organized as follows: Section 2 contains a description of the design of LegionFS, including a brief overview of the Legion wide-area operating system and an in-depth discussion of naming, security, scalability, extensibility, and adaptability. Section 3 contains a performance evaluation highlighting the advantages afforded by a scalable design. Section 4 presents an overview of related work and Section 5 concludes.

2. LEGIONFS DESIGN

Legion [17] is middleware that provides the illusion of a single virtual machine and the security to cope with its untrusted, distributed realization. From its inception, Legion was designed to deal with tens of thousands of hosts and millions of objects - a capability lacking in other object-based distributed systems. This section discusses the key areas of Legion as they apply to the design of LegionFS.

2.1 Object Model

Legion is an object-based system comprised of independent, logically address space-disjoint, active objects that communicate via remote procedure calls (RPCs). Objects represent coarse-grained resources and entities such as users, hosts, schedulers, files, and directories. Each Legion object belongs to a class, which is itself a Legion object. Much of what is usually considered system-level responsibility is delegated to user-level class objects. For instance, Legion classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies.

Legion objects may be active or inactive, and store their internal state on disk. Objects may be migrated simply by transferring this internal state to another host. The object's class then spawns a process which is instantiated with the migrated internal state.

The complete set of method signatures exported by an object defines its interface. The Legion file abstraction is a BasicFileObject, whose methods closely resemble UNIX system calls such as read, write, and seek. ContextObjects manage the Legion name space.

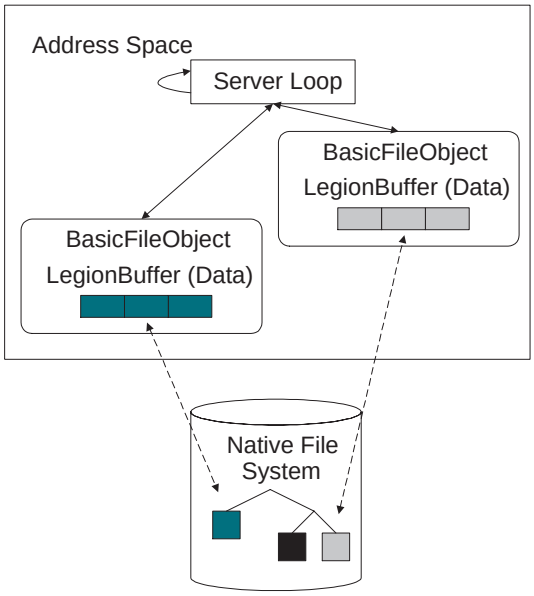

Figure 1: ProxyMultiObject

Due to the resource inefficiency of representing each as a standalone process, files and contexts residing on one host have been aggregated into container processes, called ProxyMultiObjects (Figure 1).

A ProxyMultiObject polls for requests and demultiplexes them to the corresponding contained file or context. Files store data in a LegionBuffer, which achieves persistence through the underlying UNIX file system. ProxyMultiObjects leverage existent file systems for data storage, providing direct access to UNIX files. Unlike traditional file servers, ProxyMultiObjects are lightweight and intended to be distributed throughout the system. They service only a portion of the name space, rather than comprising it in its entirety.

2.2 Naming

User-defined text strings called *context names* identify Legion objects. Context names are mapped by a directory service called *context space* to unique, location-independent binary names called *Legion object identifiers* (LOIDs). For direct object-to-object communication, LOIDs must be bound (via a *binding process*) to low-level *Object Addresses*. An Object Address (OA) represents an arbitrary communication endpoint, such as a TCP socket.

The LOID records the class of an object, its instance number, and a public key to enable encrypted communication. New LOID types can be constructed to contain additional security information (such as an X.509 certificate), location hints, and other information.

Context space is similar to a globally distributed, rooted directory. It is comprised of ContextObjects, which provide mappings from context names to LOIDs in the same fashion that directories map path names to inode numbers. Unlike directories, ContextObjects may contain references to arbitrary objects.

Having translated a context name to a LOID, an object consults a series of distributed caches to bind the LOID to an OA. Each object maintains a local binding cache. A binding cache miss results in a call to a Binding Agent object. Cache misses at the Binding Agent are serviced by the class of the LOID. This operation recurses, if necessary, but is guaranteed to terminate at LegionClass, the root of the Legion object hierarchy.

Legion's location-independent naming facilitates fault tolerance and replication. Because objects are not bound by name to individual hosts, they may be seamlessly migrated. If an object's host fails, but the internal state of an object is still accessible, the object's class may restart it elsewhere.

Classes may act as replication managers by mapping one LOID to multiple OAs, referring to objects on different hosts. A class object is a logical replication manager as its instances would likely employ the same replica consistency policies. By entrusting a class object with more responsibility, the system increases the load on that object. Means of ensuring that individual objects do not become bottlenecks are discussed in Section 2.4.

2.3 Security

Legion's distributed, extensible nature and user-level implementation prevent it from relying on a trusted code base or kernel. Furthermore, there is no concept of a superuser in Legion. Individual objects are responsible for legislating and enforcing their own security policies. The public key embedded in an object's name enables secure communication with other objects. Objects are free to negotiate the per-transaction security level on messages, such as full encryption, digital signatures, or cleartext.

When a user authenticates to Legion, currently via password, she obtains a short-lived, unforgeable credential [12] that uniquely identifies her. Authorization is determined by an Access Control List (ACL) associated with each object; an ACL enumerates the operations on an object and the associated access rights of specific principals (or groups of principals). If the signer of any credentials passed in an invocation is allowed to perform the operation, the operation is permitted.

Per-method access control facilitates finer-grained security than traditional UNIX file systems. No special privilege is necessary to create a group upon which to base access. A client can dynamically modify the level of security employed for communication, for example to use encryption when transacting with a geographically-distant peer, but to communicate in the clear within a cluster. Specialized file objects can be designed to keep audit trails on a per-object or per-user basis (i.e., auditing can be performed by someone other than a system administrator).

2.4 Scalability

LegionFS distributes files and contexts across the available resources in the system. This allows applications to access files without encountering centralized servers and ensures that they can enjoy a larger percentage of the available network bandwidth without contending with other application accesses.

Scheduler objects provide placement decisions upon object creation. Utilizing information on host load, network connectivity, or other system-wide metadata, a scheduler can make intelligent placement decisions. A user may employ existing schedulers, implement an application-tailored scheduler which places files, contexts, and objects according to domain-specific requirements, or may enforce directed placement decisions. Using the latter mechanism, a user might specify that all of his files be created on a local host or within a highly-connected, nearby cluster. This ensures that most file accesses are local, while allowing for wide-area access. It also isolates user file accesses to achieve maximum efficiency. A user may employ the replication techniques described elsewhere in this paper to tolerate failures of local resources. This provides highly-efficient access in the common case, with a measure of insurance in case of host or disk failures.

The fully-distributed design of LegionFS allows the user to remain ignorant of the constraints of physical disk enclosures, available disk space, and file system allocations. Administrators seamlessly incorporate additional storage resources into the file system. By simply adding a storage subsystem to a context of available storage elements, the additional space is advertised to the system and becomes a target for placement decisions.

LegionFS utilizes multiple levels of caching to facilitate efficient file and directory lookups and employs limited forms of replication. Aside from their role in the binding process, Binding Agents cache translations between context names and LOIDs. lnfsd similarly caches translations to avoid excessive RPCs.

Manager objects such as classes can become hot spots. Fortunately, there is no inherent reason to have one class manager for all instances of a particular class. To mitigate potential bottlenecks, management responsibilities are distributed across 'clones' of a particular class.

2.5 Extensibility

LegionFS differentiates between objects according to their exported interfaces, not their implementations. For example, LegionFS interacts with any object providing the standard BasicFileObject interface as if it were a file. By focusing on the interface without concern for the object's actual class or implementation, LegionFS provides an extensible set of services which can be specialized on an application- or domain-specific basis. An object may provide a value-added service by changing the semantics associated with a method. Thus the same interface can be used to wrap different implementations. Further, an interface may be augmented to provide functionality in the form of additional methods.

A newly-minted object exporting the standard interface may be accessed by existent libraries. If functionality warrants an additional method, it may be implemented, exported by the object, and incorporated into a newly generated library. This allows multiple policies governing a particular design issue to co-exist. A programmer builds upon lower-level functionality, such as the Legion security and communication layers, to construct objects suited for particular domains, adding them to the pool of objects already populating the system.

Legion's event-based protocol stack provides an additional opportunity for extensibility. Remote messages and exceptions are intercepted and announced to higher-level handlers. These handlers are registered according to priority and may handle an event or provide limited processing and announce the event to subsequent handlers. The Legion security layer is implemented as a layer in the protocol stack. Operations that transcend method invocations, such as an auditing facility, may be implemented as additional layers in the stack.

Providing excessive and heavy-weight functionality (such as consistency and replication) in all file and contexts objects is inappropriate as some applications neither require nor want the overhead associated with these mechanisms. Instead LegionFS provides the basic set of functionality described above and the framework to extend semantics where desired. Such functionality need be implemented only in the objects that require it, without impeding objects and applications that do not.

Interface inheritance was useful in implementing ProxyMultiObjects, TwoDFileObjects, and Simple K-Copy Classes (SKCC). TwoDFileObjects are a domain-specific implementation serving the scientific community, but are applicable to a broader audience. A TwoDFileObject implements the BasicFileObject interface such that reads and writes are striped across constituent, underlying BasicFileObjects, arranged as a two-dimensional matrix. A parallel file interface provides convenient access to applications perform-

ing matrix operations. The two-dimensional design degenerates to striping for high-performance I/O.

SKCC wrap standard classes to provide fault tolerance, by replicating an object's internal state (but not the object itself) across a number of user-specified storage elements. The state of an active class object may be synchronized across the replicas at convenient stable points of execution, such as during object deactivation. This approach provides a good measure of fault tolerance with a minimum of performance degradation.

Some environments need more full-featured replication and consistency guarantees than those provided by LegionFS. It is possible to extend ContextObjects to perform replication management: instead of a one-to-one mapping of context names to LOIDs, ContextObjects could provide a one-to-many context name-to-LOID translation. The ContextObject could perform replica selection based on availability or network connectivity constraints.

File data consistency is not addressed by basic Legion mechanisms, because no current Legion object caches file data. The initial implementation of lnfsd, which serves as an access point to LegionFS, provides NFS-like consistency semantics; it caches data for a configurable amount of time before re-validating file metadata via a stat call. There are important classes of domains where consistency guarantees are not appropriate, for example large read-only scientific datasets. For environments where consistency is necessary, it can be handled on a per-file or per-context basis at the object itself, without forcing the semantics on users accessing other data. An object could grant leases [16], which are more scalable than simple callbacks [21], or implement any other consistency scheme.

2.6 Adaptability

A wide-area file system must be adaptable to a diverse set of network, load, and system-wide conditions. LegionFS facilitates adaptation by maintaining system-wide metadata. Each object has an associated, arbitrary set of <key,value> pairs. Typical attributes for a host object include load averages, architecture, and operating system. This list could easily be extended to include other factors which might affect file placement in a wide-area environment such as network interfaces and their associated nominal bandwidths, local file systems, and disk configurations.

Attributes are available directly from the object and are also stored in a metadata repository, called the Collection. The Collection is a hierarchically distributed set of objects which is queried by schedulers to determine object characteristics and state. Objects periodically push their state information to the Collection. More sophisticated monitoring facilities such as the Network Weather Service [44] could also be employed to populate the Collection.

The Collection allows applications to track the dynamics of the system as well as capitalize on its more stable, inherent diversity. A geographically-distributed system is likely to contain a range of heterogeneity in the form of underlying file systems, storage devices, and architectures. If the characteristics of an application are well-known, the application may benefit from placement that matches these needs against the properties of particular resources. As specific examples, XFS [4] provides benefits to streaming applications by allowing them to circumvent standard kernel buffer caches and RAID enclosures may provide more efficient availability than can be provided at higher layers in the system.

Since files and contexts are logically self-contained objects, it is more convenient to specify fine-grained policies than would be possible in a more conventional distributed file system. Objects may act on these policies asynchronously with respect to the user. LegionFS allows a user to explicitly migrate or deactivate an object. More interesting behaviors include the ability to migrate due to network conditions or replicate to accommodate increased load. A file might consider re-negotiating transfer size, changing consistency policy, or varying write-back policy in accordance with network constraints. Many of these issues were explored in the Coda file system [24].

Golding et al. [15] discuss means of exploiting idle periods in computer systems. Assuming fair load distribution, a file object is more likely to experience idleness than a centralized file server. Therefore, a file object has an opportunity to analyze its access patterns in order to prefetch. The file system literature is replete with prefeteching mechanisms [8] [10] [26] [29] [35]. Often efficiency is a concern as the mechanisms must be realized within the limited latency and memory constraints of a kernel-resident file system. Being less memory-constrained, a Legion file may retain more exact data concerning access patterns and prefetch schedules. Having characterized its own usage, a file object could provide access hints [35] to the client to facilitate prefetching across the network. As a further optimization, a file object could recognize long periods of inactivity and move data to a more space-efficient, but less readily-accessible representation, file system, or storage device, as done in the HP AutoRAID system [43].

3. EVALUATION

This section compares the scalable design of LegionFS to a more traditional volume-oriented approach. The gross disparity in potential parallelism between the two experimental setups is intentional, and serves to validate the move from monolithic servers as employed by NFS to the peer-to-peer architecture advocated by Legion, xFS [4], and others. Previous work [42] examined Legion wide-area I/O performance alongside the Globus [14] I/O facility and FTP, the de facto means of transferring files in a wide-area environment.

Each benchmark utilizes the Centurion cluster [28] at the University of Virginia. These experiments employ 400-Mhz dual-processor Pentium II machines running Linux 2.2.14 with 256 MB of main memory and IDE local disks. These commodity components are directly connected to 100 Mbps Ethernet switches, which are in turn connected via a 1 Gbps switch. A 100 Mbps link provides the cluster with access to the vBNS. During the second experiment, remote hosts at Binghamton University and the University of Minnesota communicate with the Centurion cluster using this connection. The Sparc hosts at Binghamton University run Solaris 5.7, while the dual-processor Intel machines at the University of Minnesota run Linux 2.2.12.

The first micro-benchmark (Figure 2) is designed to show that LegionFS clients accessing independent subtrees achieve a linear increase in aggregate throughput in accordance with the linear growth of the network. lnfsd performance also scales nearly linearly. On the other hand, NFS performance scales poorly with additional clients. Each reader accesses a private 10 MB file via a series of 1 MB transfers. The experiment varies the number of simultaneous readers per run. Each reader and its associated target file are placed on separate nodes, though they share the same switch whenever possible. The experiment employs up to 100 nodes, providing the opportunity to scale the benchmark to 50 readers accessing files on 50 separate nodes. The LegionFS case utilizes the Legion BasicFile library and distributes BasicFileObjects throughout the network. These same BasicFileObjects are accessed in the lnfsd experiment by clients that are co-located with the lnfs interposition agents. The NFS experiment uses a single NFS daemon to service file system requests from 50 readers. In all cases, caching occurs only on the server side.

Single readers attained 4.5 MB/s and 2.1 MB/s under LegionFS

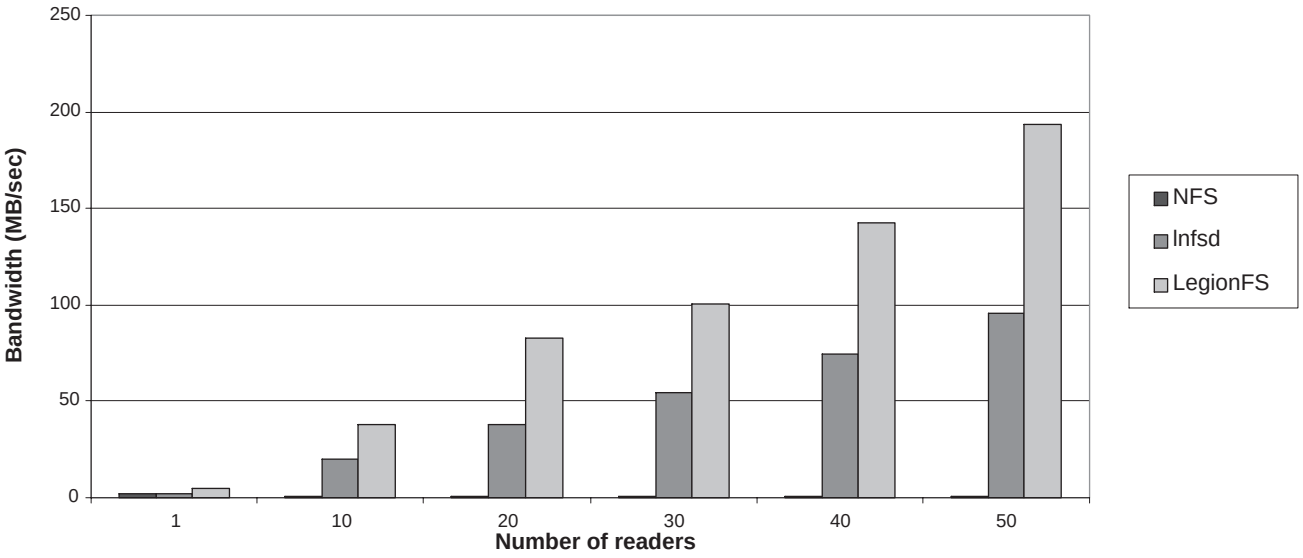

Figure 2: Scalability of read performance in NFS, lnfsd, and LegionFS

and NFS, respectively. NFS is limited to 4K transfers over the network, whereas LegionFS can use arbitrary transfer sizes. lnfsd performs similarly with a bandwidth of 2.2 MB/s. lnfsd performance is degraded by frequent context switches and RPCs between the kernel client and lnfsd. This pure overhead is the expense of supporting legacy applications, and is avoided when using the Legion library interfaces. lnfsd attempts to mitigate the inefficiency of its user-level implementation by performing read-ahead on sequential file access, asynchronous write-behind, and file and metadata caching.

LegionFS and lnfsd each achieved peak performance at 50 readers, yielding aggregate bandwidths of 193.8 MB/s and 95.4 MB/s, respectively. NFS peak performance occurred at 2 readers, yielding aggregate bandwidth of 2.1 MB/s. NFS does not scale well with more than two readers, whereas both lnfsd and LegionFS scale linearly with the number of readers, assuming the file partitioning described above.

To put the above results in the context of a popular domain, the next benchmark examines access to a subset of the Protein Data Bank (PDB). This experiment is intended to simulate the workings of parameter space studies such as Feature [3], which has been used to scan the PDB searching for calcium binding sites. Feature, and similar parameter space studies, employ coarse-grained parallelism to execute large simultaneous runs against different datasets. The Protein Data Bank is typical of large datasets in that it services many applications from various sites worldwide desiring to access it via a high-sustained data rate.

Clients read a subset of files from the PDB stored in Legion context space. To avoid excessively long runs, only the first 100 files from the PDB were accessed. These files have an average size of approximately 171 KBs, with a file size standard deviation of 272 KBs. Such a distribution indicates there are many small files in the database along with a few very large files. A client's execution is termed a job and consists of 100 whole-file reads. Client execution is not synchronized. Each stage of the experiment defines the number of active clients. While the number of active clients is varied from 1 to 32 between stages of the experiment, the number of jobs remains constant at 100.

The test harness iterates through the target hosts in round-robin fashion, assigning readers until the specified parallelism is reached. Upon a client's completion, an additional reader begins execution. Each client records the elapsed time to read the list of files in its entirety and calculates its bandwidth. The averages of these bandwidths are reported on the left-hand side of Figures 3, 4, 5, and 6 as average client bandwidth, along with the associated standard deviations. The test harness responsible for remotely executing the hosts records the elapsed time from instantiation of the first job to completion of the last. This aggregate bandwidth is reported on the right-hand pane of the same figures. The two metrics are intended to capture the performance of individual clients and the throughput of the system under a specified load. During the prelude and epilogue of an experiment, the test is not in a steady state and the number of active clients is below the specified value.

Files hosted on the Centurion cluster store the PDB data. Though only the first 100 files are accessed, 12000 files are stored under a single context. This simulates accessing a relatively small subset of a large data collection. The experiments vary the placement of the clients and the file system distribution to cover local- and wide-area environments and volume-oriented and peer-to-peer designs. The local-area experiments (Figures 3 and 4) execute each of the clients on one of 32 nodes within the Centurion cluster, though they are never co-located with PDB files. The wide-area experiments (Figure 5 and 6) place jobs on a pool of 4 machines at the University of Minnesota and 12 at Binghamton University. The relative dearth of remote machines requires that clients be scheduled on the same host when their active number exceeds 16. While an unfortunate incongruity between the environments, the jobs are I/O-bound and do not suffer unduly by being placed on the same host. The ex-

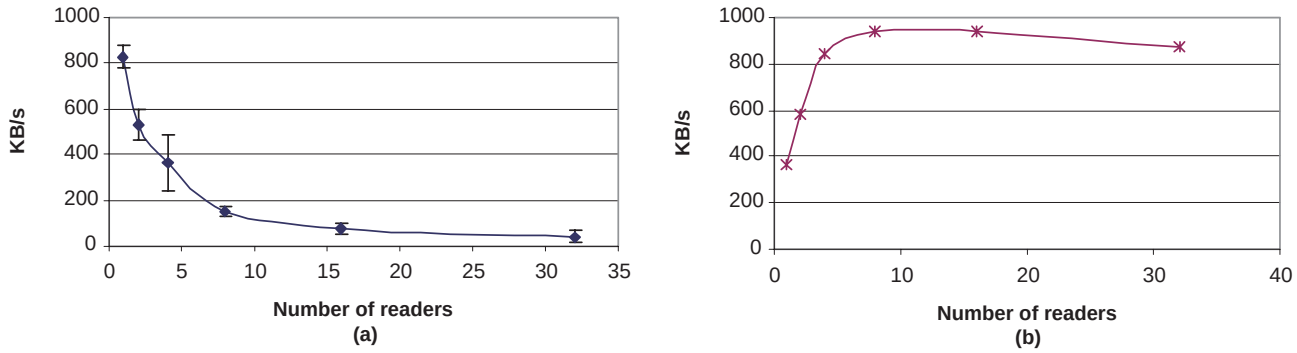

Figure 3: Centurion clients accessing PDB data stored in ProxyMultiObject within Centurion cluster. (a) Average Client Bandwidth (b) Aggregate Bandwidth

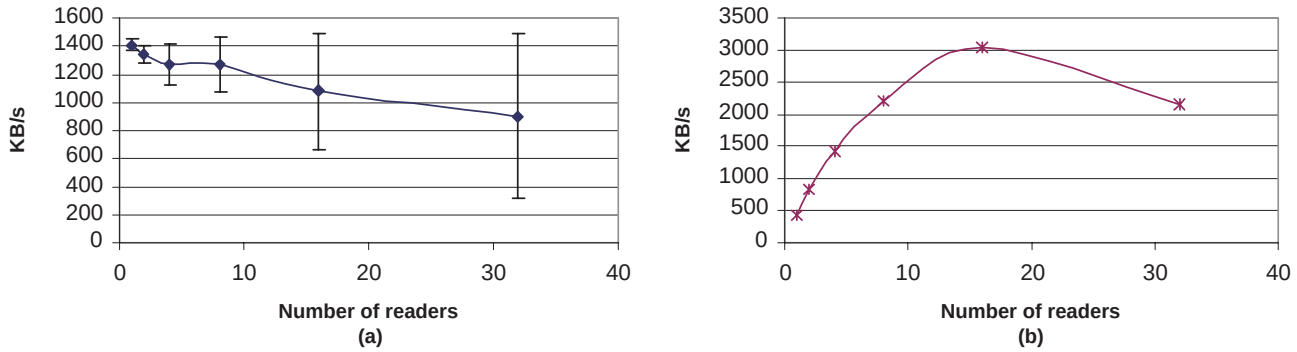

Figure 4: Centurion clients accessing PDB data stored in BasicFileObjects within Centurion cluster. (a) Average Client Bandwidth (b) Aggregate Bandwidth

periments designed to stress the volume-oriented design (Figures 3 and 5) host all files within a single ProxyMultiObject. The peer-to-peer setup (Figures 4 and 6) distributes the BasicFileObjects across 32 Centurion nodes.

As expected, the ProxyMultiObject shows immediate and drastic performance degradation with increasing load. The effect is particularly acute when the clients execute within the cluster (Figure 3). In this case, there is a near 50% reduction in bandwidth with each doubling of the number of active readers. Figure 5 exhibits a similar dramatic trend, though the curve is not as steep. Given its relatively greater distance from the ProxyMultiObject, a client's requests are less densely concentrated than when running within the cluster. This ensures less immediate contention for the ProxyMultiObject and results in a slightly less severe performance impact. Clients achieve peak average bandwidth at 827 KB/s and 312 KB/s within local-area and wide-area environments, respectively. This occurs when a client need not contend with other readers. Average client bandwidth is minimized under each case at 32 readers, dropping to 42 KB/s and 35 KB/s for the local-area and wide-area cases, respectively.

The ProxyMultiObject aggregate bandwidth curves bear close resemblance to one another. Aggregate bandwidth grows steadily until a maximum is reached at 8 clients, and then flattens. The ProxyMultiObject is best utilized by a small number of clients, but can not continue to scale with increased load. The peak bandwidths of 944 KB/s within the cluster and 717 KB/s of the remote clients may seem surprisingly small in comparison to the achieved average client bandwidths. This occurs because the aggregate bandwidth measures the total elapsed execution time of all 100 jobs, including the time required to start the remote jobs, transfer an input file, and reap the results. While this additional overhead comprises a non-trivial percentage of the total job turnaround time, it is illustrative of actual execution. The average client bandwidths report performance once the job has begun execution; the aggregate bandwidth is indicative of system throughput.

Distributing the load amongst the BasicFileObjects leads to more graceful performance degradation in Figures 4 and 6. The system does not scale linearly, however. Unlike the raw throughput experiment above, clients in this setup access a shared portion of the PDB, rather than dedicated per-client files. While average client bandwidth remains fairly steady with a few additional clients in both graphs, large numbers of active clients increase the likelihood that one or more will access the same data, leading to contention at the BasicFileObject. At 1406 KB/s, peak client bandwidth ac-

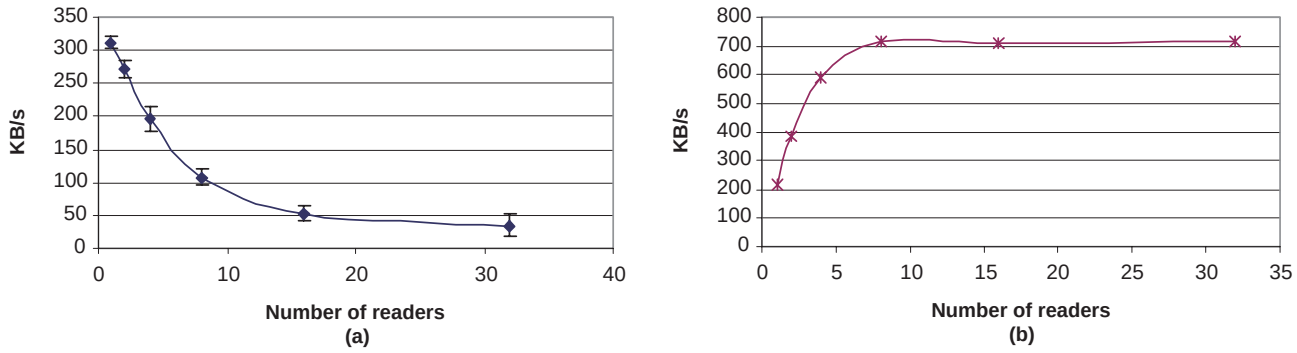

Figure 5: Remote clients accessing PDB data stored in ProxyMultiObject within Centurion cluster. (a) Average Client Bandwidth (b) Aggregate Bandwidth

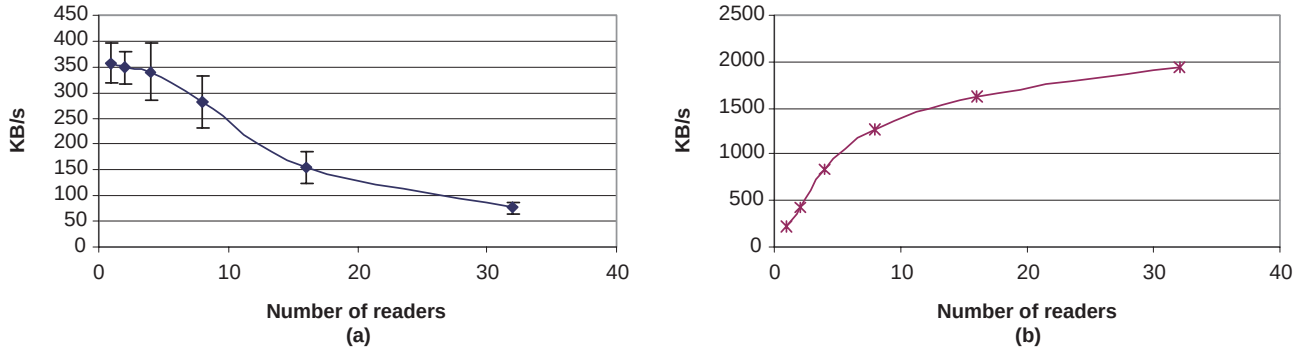

Figure 6: Remote clients accessing PDB data stored in BasicFileObjects within Centurion cluster. (a) Average Client Bandwidth (b) Aggregate Bandwidth

cessing BasicFileObjects within the cluster is significantly higher than the corresponding ProxyMultiObject case. This suggests additional benefits of BasicFileObjects. ProxyMultiObjects must maintain state for each constituent object, leading to overhead when demultiplexing a request to the target. Further, BasicFileObjects may greater exploit the local file system cache since they serve a much smaller portion of the name space and are less likely to suffer capacity cache misses.

The increasingly large standard deviations of Figure 4 result from the contention described above. Since all clients iterate through files in the same order, contention is more likely at the onset of the experiment. During the ramp up stage, clients perform more poorly than during steady-state execution. This may seem counter-intuitive as the test has not reached its full complement of active clients. Nevertheless, clients caravan behind one another until adequate spacing is achieved. As a job completes, a new job begins execution and inherits the spacing won by the finished job. This effect is not present in the wide-area case of Figure 6, where temporal distance between jobs is achieved by the relatively longer time required to start remote execution.

The caravan effect is pronounced in the aggregate bandwidths of Figure 4(b), where performance dips under the load of 32 clients. Unlike previous cases, the retarded progress of the 32 initial clients is significant amongst 100 jobs. Aggregate bandwidth reaches its height of 3044 KB/s at 16 clients. Unburdened by temporal proximity, the remote clients accessing the BasicFileObjects contribute to increased aggregate bandwidth up to 32 clients at 1938 KB/s.

4. RELATED WORK

The continued and increasing interest in wide-scale distributed computing, driven by high-bandwidth, long-haul networks and the economies of scale of commodity hardware, has lead to the design of file systems and data access facilities engineered specifically for such an environment [2] [6] [5] [9] [13]. Such file systems were motivated by concerns inherent in wide-area environments, unlike their predecessors which were originally intended for campus- or local-area networks and were retrofitted to fill expanding roles [21] [38] [37].

Recognizing the diverse and evolving nature of wide-area environments, researchers have followed the approach taken in LegionFS of developing layered architectures consisting of a potentially-expansive set of services integrated via lower-level protocols [5] [9]. SRB [5] is middleware that provides access to data stored on heterogeneous resources residing within a distributed sys-

tem. SRB Agents contact the MCAT metadata service in order to locate and transact with local storage facilities, such as file systems, databases, and hierarchical storage systems.

In the context of the Globus Grid Toolkit [14], Chervenak et al. [9] posit a framework that stresses the importance of employing standard protocols to achieve interoperability. This work leverages previous work on Globus data access [6], deployed internet infrastructure and protocols such as HTTP and LDAP, and protocol extensions such as GridFTP [1]. File replication and selection via Condor ClassAds [36] have been successfully implemented using these mechanisms [41]. While Globus benefits from existent protocols and internet services, it is also constrained by their mandates. To ensure interoperability, entities must communicate using the standard protocol. A perceived need or feature in the service may require amending that standard. Not held captive to prescribed interfaces, Legion objects may simply export new methods. Because internet protocols evolved independently, they do not necessarily share commonalities along important dimensions such as naming, authentication, and authorization. Thus features such as authorization, that might be expected to pervade the system, must be implemented anew for each service, either as a mapping to each specific protocol or outside the service proper. By exposing uniform and integrated mechanisms to distributed objects, LegionFS ensures file abstractions are secured in a consistent manner without this burden.

WebFS [40] and Ufo [2] also provide access to internet services. WebFS is a kernel-resident file system that provides access to the global HTTP name space. It supports three cache coherency policies deemed appropriate for HTTP access: last writer wins, append only, and multicast updates. Ufo employs the UNIX tracing facility to intercept open system calls and transfers whole files from FTP and HTTP servers.

The PUNCH Virtual File System (PVFS) [13] interposes unmodified NFS clients and servers with NFS-forwarding proxies. PVFS allows a client executing on a compute server to access files stored within another security domain. During the course of a session, clients are allocated a temporary shadow account on the compute server. Requests are directed to the proxy, co-located with the target NFS server. The proxy maps the shadow account id of the request to the user's corresponding id on the target host and forwards the request to the NFS server.

File system adaptability has been addressed in Coda [24] and Odyssey [34], which support application-transparent and application-aware adaptation, respectively. Both adaptation strategies are designed to provide resilience in the presence of varying network performance and collect simple information about certain resources to aid in system monitoring.

The Hurricane File System (HFS) [25] employs building blocks to encapsulate file system policies, such as prefetching and distribution. These building blocks may be composed according to their interfaces to achieve per-file and per-open file instance specialization.

While building blocks are relatively coarse-grained and focus on policies that span the entire file system, stacking allows individual file system calls to be interposed. Higher layers in a stacked file system may provide additional processing or modify arguments before invoking the same operation on the subsequent, symmetric layer. Stacking vnodes [39] create a chain of traditional vnodes to support interposing. Ficus [19] is a replicated file system that allows kernel- or user-level file system modules exporting the vnode interface to be stacked. Later work [20] abandoned the rigid vnode interface in favor of the UCLA interface which is formed at kernel initialization and is the union of interfaces exported by each layer. A directory subtree constitutes a layer and may be mounted atop another layer to form a stack.

The Spring [32] object-oriented operating system is composed of cooperating servers running on a micro-kernel. File objects inherit from Spring interfaces charged with handling operations such as paging, authentication, consistency, and I/O [33]. A new file system is allocated by contacting its corresponding creator object. This file system may be stacked on an existing file system by means of a stackon method [23]. Subsequent work on the Solaris MC File System [30] replaces the vnode interface with a new interface defined in CORBA IDL.

The FiST language [45] is a high-level language for describing stackable file systems. By providing a standard interface to mask operating system peculiarities, FiST allows for portable file system implementations. File systems may interpose specific operations or a set of operations and may choose to insert code before, after, or in lieu of the operation. The FiST description of the file system extensions is input to fistgen, a parser and code generator, which outputs kernel C sources.

Legion's goal of acceptance amongst diverse organizations requires both that it provide secure means of cross-domain access and that administrative overhead be minimized. Centralized key services, such as Kerberos, have been successfully employed by AFS [21] [38] and DFS [22], but do not meet these requirements. The centralized key management in Kerberos becomes increasingly difficult as the system scales. The Self-certifying File System (SFS) [31] embeds a public key in the name of a file, making "self-certifying" pathnames. LegionFS leverages a similar, distributed key management system.

The notion of serverless or peer-to-peer file systems was popularized by xFS [4], and has spurred a rash of related projects [7] [11] [18]. xFS implements a serverless architecture to provide scalable file service, and provides data redundancy through networked disk striping to increase reliability. JetFile [18] relies on multicast to locate files distributed throughout the network. This location-independent naming scheme encourages data replication. Unfortunately, multicast is problematic in wide-area environments as it floods networks and relies on router support.

5. CONCLUSION

This paper has examined a small sample of the usage scenarios and requirements of file access in Computational Grids as they exist today. With this knowledge, we advocate an architecture integrated by basic, but powerful, facilities such as location-independent naming and pervasive authentication, authorization, and confidentiality mechanisms. A scalable, peer-to-peer design ensures that the Grid can benefit fully from its consituent resources, rather than be bound by the performance limitations of centralized services. Finally, wide-area applications can exploit the dynamics of the system through adaption and continue to evolve with our understanding of the Grid's potential through a framework promoting extensibility.

Understanding that many classes of scientific applications can best utilize the Grid without the imposition of costly functionality, LegionFS follows a minimalist approach. However, the means of incorporating application-specific policies is enabled by the set of mechanisms afforded by Legion. This ensures that emerging services and applications can effortlessly utilize existent infrastructure to form a cohesive system, without having to cobble and reconcile mechanisms that were not intended to work in unison. Extensions to core services, such as ProxyMultiObjects and TwoDFileObjects, are a result of this philosophy. We have also described as yet unimplemented opportunities such as replication via class or context

objects and consistency guarantees that capitalize on lower-level Legion facilities.

The heterogeniety, wealth of storage, and abundance of CPU cycles in wide-area environments suggest interesting possibilities for file systems. The ability to schedule processes according to their I/O affinities and leverage idle periods are two avenues for continued research. We expect wide-area file systems to evolve into more than mere extensions of smaller-scale distributed file systems. Rather, they may efficiently bridge local or local-area file systems, gaining advantage from their unique strengths.

While anticipating the future of wide-area file systems, this paper provided a quantitative study of the current state of LegionFS. The utility of LegionFS has been demonstrated with the Legion object-to-object protocol as well as lnfsd, a user-level daemon designed to exploit UNIX file system calls and provide an interface between a UNIX kernel and LegionFS. Benchmarks showed that the scalability of LegionFS compared favorably under load to volume-based file systems, such as NFS. Finally, LegionFS was shown to facilitate efficient data access in an important scientific domain, the Protein Data Bank.

6. ACKNOWLEDGMENTS

This work was partially supported by Logicon (for the DoD HPCMOD/PET program) DAHC 94-96-C-0008, NSF-NGS EIA-9974968, NSF-NPACI ASC-96-10920, and a grant from NASA-IPG. In addition, the authors would like to thank John Karpovich and Mark Morgan for answering questions on the Legion architecture, Norm Beekwilder for his aid in experimental design and administration, Katherine Holcomb for her patience as we taxed the Centurion network, Anand Natrajan for his feedback and guidance with Legion scheduling, and the entire Legion team. The wide-area results would not have been feasible without contributed academic resources. The authors thank Mike Lewis for the use of machines at Binghamton University and Jon Weissman of the University of Minnesota for his support. Finally, we thank our shepherd, Ann Chervenak, and the anonymous referees for adding clarity to this paper's presentation.

7. REFERENCES

[1] Gridftp: Ftp extensions for the grid. Grid Forum Remote Data Access group, October 2000.

[2] 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 *1997 Annual Technical Conference on Unix and Advanced Computing Systems (USENIX '97)*, January 1997.

[3] R. Altman and R. Moore. Knowledge from biological data collections. *enVision*, 16(2), April 2000.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126, Copper Mountain, CO, December 1995. ACM Press.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The sdsc storage resource broker. In *CASCON'98*, Toronto,Canada, November-December 1998.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, May 1999. ACM Press.

[7] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Sigmetrics 2000*, pages 34–43, 2000.

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–196, Ottawa, Ontario, Canada, 1995.

[9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 1999.

[10] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 1993.

[11] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HOTOS VIII*, Schoss Elmau, Germany, May 2001.

[12] A. Ferrari, F. Knabe, M. Humphrey, S. Chapin, and A. Grimshaw. A flexible security system for metacomputing environments. Technical report, University of Virginia, December 1998.

[13] R. J. Figueiredo, N. H. Kapadia, and J. A. B. Fortes. The punch virtual file system: Seamless access to decentralized storage services in a computational grid. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, August 2001.

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

[15] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Technical Conference*, pages 201–212, January 1995.

[16] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210. ACM Press, December 1989.

[17] A. S. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–37, May 1999.

[18] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.

[19] R. G. Guy, J. S. Heidemann, W. Mak, J. Thomas W. Page, G. J. Popek, and D. Rothmeier. Implementation of the ficus replicated file system. In *USENIX Conference Proceedings*, Berkeley, CA, June 1990. USENIX Association.

[20] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[22] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T.

Tu, and E. R. Zayas. Decorum file system architectural overview. In *Proceedings of the 1990 Summer USENIX Conference*, pages 151–163, Anaheim, CA, June 1990. USENIX Association.

[23] Y. A. Khalidi and M. N. Nelson. Extensible file systems in spring. In *Proceedings of the Fourteen ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993. ACM Press.

[24] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 3–25. ACM Press, February 1992.

[25] O. Krieger and M. Stumm. Hfs: A performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, August 1997.

[26] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[27] Y. G. Leclerc, M. Reddy, L. Iverson, and N. Bletter. Terravisionii: An overview. Technical report, SRI International, 2000.

[28] G. Lindahl, S. J. Chapin, N. Beekwilder, and A. Grimshaw. Experiences with legion on the centurion cluster. Technical report, University of Virginia, August 1998.

[29] T. M. Madhyastha and D. A. Reed. Input/output access pattern classification using hidden markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, November 1997.

[30] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris mc file system framework. Technical report, Sun Microsystems Research, 1996.

[31] D. Mazieres, M. Kaminsky, M. F. Kasshoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999. ACM Press.

[32] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the spring system. In *CompCon Conference Proceedings*, 1994.

[33] M. Nelson, Y. Khalidi, and P. Madany. The spring file system. Technical report, Sun Microsystems Research, February 1993.

[34] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, St. Malo, France, October 1997. ACM Press.

[35] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM Press, December 1995.

[36] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[37] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference Proceedings*, Berkeley, CA, Summer 1985. USENIX Association.

[38] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, 23(5):9–22, May 1990.

[39] G. C. Skinner and T. K. Wong. "stacking" vnodes: A progress report. In *USENIX Conference Proceedings*, pages 61–74. USENIX Association, Summer 1993.

[40] A. M. Vahdat, P. C. Eastham, and T. E. Anderson. Webfs: A global cache coherent file system. Technical report, University of California, Berkeley, 1996.

[41] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. In *Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid*, pages 106–113. IEEE Computer Society Press, May 2001.

[42] B. S. White, A. S. Grimshaw, and A. Nguyen-Tuong. Grid-based file access: The legion i/o model. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, August 2000. IEEE Computer Society Press.

[43] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[44] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1998.

[45] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, California, June 2000. USENIX Association.