

A SECURE AND SCALABLE FILE SYSTEM INFRASTRUCTURE

Brian White, Michael Walker, Marty Humphrey, and Andrew Grimshaw

Computer Science Department

University of Virginia

Charlottesville, VA 22903

ABSTRACT: *The domain of file system usage spans a wide range of geographic environments, usage scenarios, and security requirements. Existing file systems generally have been designed for a particular set of scenarios; however, no one system works well across all environments simultaneously. LegionFS is a file system infrastructure that allows multiple policies and implementations to co-exist in order to meet these diverse requirements. Key features of LegionFS include an extensible object model, strong and configurable security at its core, a peer-to-peer design in both its directory service and its file service, and a rich set of metadata to facilitate adaptive behavior. LegionFS can be accessed directly or through Infsd, a user-level daemon designed to exploit UNIX file system calls and provide an interface between NFS and LegionFS. Scalability tests show that LegionFS and Infsd experience a linear increase in aggregate throughput in accordance with the linear growth of the network, yielding an aggregate read bandwidth of 193.80 MB/sec on a switched 100MB/s Ethernet backplane with 50 simultaneous readers.*

1 Introduction

Existing file systems have generally been designed and optimized for a particular type of environment or usage pattern. “General purpose file systems” are often implemented based on design decisions pertinent to the system’s intended scale, be it a wide-area environment, [Ale97] [Vah98], a local-area environment [San85] [How88] [Sat90] [Kaz90], a cluster [Ji00] [And95], or a single host [McK84]. Additionally, file systems may target a particular domain such as real-time continuous media access [Mar96] or particular interfaces such as those provided by parallel file systems [Cor93] [Nie96]. There are two main reasons for this design rationale. First, the design decisions made for a particular environment can negatively impact functionality and performance in other environments, and it is too difficult or time-consuming to design and implement optimizations for each of the different environments. Second, there has not been a need for a file system that spans the different types of environments.

However, emerging wide-area collaborations are rapidly causing the manner and mechanisms by which files, and more generally data, are stored, retrieved, and accessed to be re-evaluated. New, inexpensive storage technology is making terabyte and petabyte weather data stores feasible, desired to be accessed both physically close to the place of data origin and by clients around the world. Companies are seeking better mechanisms by which to share information and data without compromising the proprietary information of any of the involved sites. Increasingly, clients desire the file system to dynamically adapt to access patterns of varying connectivity, security, and latency requirements. Existing file systems can satisfy the performance requirements of a particular set of usage scenarios; however, no one file system works well across all environments simultaneously.

The Legion File System, *LegionFS*, provides a flexible framework that can span the range of geographic environments, usage scenarios, and security requirements. A key to the design and implementation has been the ability to utilize Legion, an object-based, user-level infrastructure for local-area and wide-area heterogeneous computation [Lew96] [Gri99]. Legion is a *meta*-operating system that utilizes host operating system services to create the illusion of a single virtual machine. In the design of LegionFS, we take the view that the underlying UNIX file systems upon which Legion executes are largely competent in the actual storage of data; the challenge is to provide a cross-architecture, cross-

organization, and scalable file system that both exploits and expands the mechanisms provided by the heterogeneous, isolated UNIX file systems.

There have been a number of key design decisions addressed in the construction of LegionFS. We consider these objectives to be fundamental to any infrastructure hoping to support any present or future data access requirements. An intentionally minimalist list is outlined: the goal is to provide a framework that allows multiple policies and implementations to co-exist, not to mandate excess functionality that may be inappropriate or cumbersome for some environments.

- **Naming:** A three-level naming system is used, consisting of [1] user-readable strings, [2] location-independent intermediary names called Legion Object Identifiers (or LOIDs), and [3] low-level communication endpoints called Object addresses.
- **Security:** Each component of the file system may exist independently, 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:** Individual files within a directory sub-tree can be distributed throughout the storage resources in an organization and I/O operations on the files can be conducted in a peer-to-peer manner, which is a natural consequence of the LegionFS object-based system. This holds also for the directory service and eliminates centralized components that can be performance bottlenecks.
- **Extensibility:** Every object publishes an interface, which may be inherited, extended, and specialized to provide an object supporting additional semantics, alternate policies, or a novel implementation.
- **Adaptability:** LegionFS maintains a rich set of system-wide metadata that may be used by objects to tailor their behavior to environmental changes.

This paper presents the design and implementation of LegionFS, focusing on the extensibility and scalability of the infrastructure rather than a performance evaluation of low-level primitives or collective operations. The performance evaluation contained in this paper is not meant as an optimal implementation for a particular environment. Rather, the results highlight the scalability of LegionFS and show the unique configurability and potential for adaptability in a file system that can span geographic and application boundaries. LegionFS provides only basic functionality, intended to be extended and tailored to meet the performance requirements of specific domains.

The core of LegionFS functionality is provided at the user-level by Legion's distributed object-based system. As such, the file and directory abstractions of LegionFS may be accessed independently of any kernel file system implementation through libraries that encapsulate Legion communication primitives. This approach provides flexibility as interfaces are not required to conform to standard UNIX system calls. To support existing applications, a modified user-level NFS daemon, *lnfsd*, has been implemented to interpose 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 presents an overview of work related to this project. Section 3 contains a description of the design of LegionFS, including a brief overview of the Legion wide-area operating system. Section 4 contains a performance evaluation and Section 5 concludes.

2 Related Work

LegionFS provides an adaptable and extensible file system infrastructure that supports scalability, security and a global namespace. File system adaptability has been addressed in Coda [Sat96] and Odyssey [Nob97], 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 both collect simple information about certain resources to aid in system monitoring. LegionFS has a flexible adaptability strategy. It collects arbitrary system information (metadata) at any

granularity, allowing file system components to modify behavior according to any chosen characteristic of the system.

Extensibility in file systems has been previously addressed by object-oriented operating systems such as Spring [Nel93] and Choices [Cam87]. File system components are represented as objects that can be replaced or extended through interface inheritance. In this manner, the file system can be configured appropriately for its environment. LegionFS is object-based, and also allows classes of file objects to be derived and substituted as desired. However, the LegionFS infrastructure is designed to support a wide range of domains, whereas other extensible systems support configurability within restricted domains.

Security has been the focus of many specialized file systems. The main idea of the Self-certifying File System (SFS) [Maz99] is the inclusion of a public key in the name of a file, making “self-certifying” pathnames. The management of keys in LegionFS is accomplished in the same manner; however LegionFS achieves scalability (and other goals) through mechanisms other than scalable key management. AFS [How88] [Sat90] and DFS [Kaz90] are each based on Kerberos [Neu94], which is a centralized key service. The centralized key management of Kerberos is very difficult to manage as the number of clients increases significantly. WebFS [Vah98] implements a distributed file system based on HTTP, and as such is not amenable to specialized access patterns or protocols.

The concept of peer-to-peer file systems designed for improved scalability and performance is not entirely new, and can be found in xFS [And95] and JetFile [Gro99]. xFS implements a serverless architecture to provide scalable file service, and provides data redundancy through networked disk striping to increase reliability. xFS is a cluster file system, and assumes trusted kernels within the cluster to enforce security. JetFile achieves scalability by allowing clients to act as servers, and achieves availability by supporting replicated files and large caches. File requests are multicast over the network to provide location transparency and encourage data replication. JetFile does not currently implement security mechanisms. Our project supports heterogeneous wide-area use, where kernels may be untrusted. Thus, fine-grained security is a primary design goal of LegionFS, whereas in xFS and JetFile it is not. The multicast protocol is known to increase network traffic as the number of peers increase, and requires the assumption that multicast packets will not be dropped by routers. LegionFS supports location transparency through its three-tiered namespace, rather than through multicast.

Unified namespaces are well established in distributed file systems such as Sprite [Nel88], NFS[Sun88], and Locus [Wal83], and location-independent naming schemes are used in AFS [How88]. LegionFS provides a single, persistent namespace with location-independent naming to facilitate file migration. The naming scheme is designed to be extensible, so that future requirements in naming can also be accommodated.

3 LegionFS Design

LegionFS is a product of the Legion project [Fer98][Gri99][Gri98][Gri97] at the University of Virginia. Legion is a middleware architecture for designing and building wide-area system services that provide the illusion of a single virtual machine to users that provides secure shared object and shared name spaces. Legion provides Operating System-like abstractions of the underlying hardware and the glue to couple diverse applications. 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. In this section, we discuss the key areas of Legion as they apply to the design of LegionFS.

3.1 Object Model

Legion is an object-based system comprising independent, logically address space disjoint, active objects that communicate with one another via remote procedure calls (RPCs). All system and application components in Legion are objects. Objects represent coarse-grained resources and entities such as users, hosts, storage elements, schedulers, metadata repositories, files, and directories.

Each Legion object belongs to a class, and each class is itself a Legion object. The complete set of signatures for an object describes that object's interface, which is determined by its class. Much of the Legion object model's power comes from the role of Legion classes; 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 (either periodically or during deactivation). 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 Legion file abstraction is a `BasicFileObject`, whose methods closely resemble UNIX system calls such as `read`, `write`, and `seek`. The Legion directory abstraction is called a `ContextObject`, and is used to effect naming, as described below. A more detailed treatment of Legion's various I/O interfaces and implementations is described in [Whi00]. 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`. For security reasons, a container may be associated with each user.

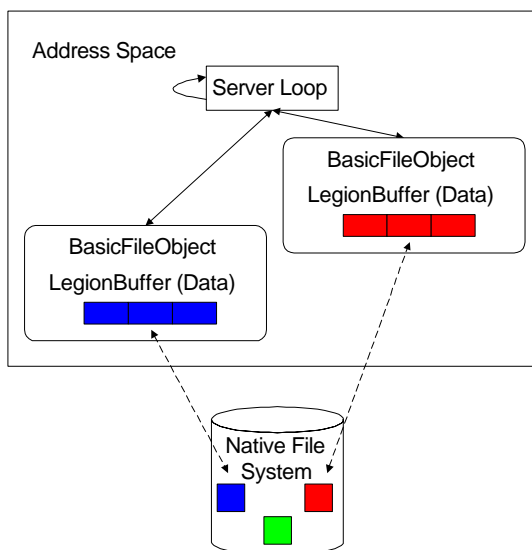


Figure 1: ProxyMultiObject

A `ProxyMultiObject` behaves much like a traditional file server. It polls for requests and demultiplexes them to the corresponding contained file or context. Contexts provide a mapping from user-readable names to object identifiers, as described in Section 3.2, via a hash table. Files store data in a `LegionBuffer`, which provides architecture neutrality for typed data. `LegionBuffers` achieve persistence through the underlying UNIX file system, as shown in Figure 1. Thus, `ProxyMultiObjects` leverage existent file systems for data storage, providing direct access to UNIX files

Unlike traditional file servers, `ProxyMultiObjects` are relatively lightweight and are intended to be distributed through the system. They service only a portion of the distributed file name space, rather than comprising it in its entirety.

3.2 Naming

Legion objects are identified using a three-level naming hierarchy. At the highest level, objects are identified by user-defined text strings called *context names*. These user-level context names are mapped by a directory service called *context space* to system-level, unique, location-independent binary

names called *Legion object identifiers (LOIDs)*. For direct object-to-object communication, LOIDs must be bound to low-level addresses that are meaningful within the context of the transport protocol that will be used for message passing. These low-level addresses are called *object addresses* and the process by which LOIDs are mapped to object addresses is called the Legion *binding process*.

The basic LOID is an extensible data structure consisting of a sequence of variable length binary string fields. Four of these fields are reserved by the system. The first three reserved fields play a key role in the LOID-to-object address binding mechanism. Field 0 is the *domain identifier*, which is used in the dynamic connection of separate Legion systems. Field 1 is a *class identifier*, bits uniquely identifying the named object's class. Field 2 is an *instance number* that distinguishes the named object from other instances of its class within the same Legion domain. Field 3 is a security field containing a public key for encrypted communication with the named object. New LOID types can be constructed to contain additional security information (such as X.509 certificate), location hints, and other information in the additional available fields.

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 such as hosts.

An Object Address (OA) is a list of *object address elements* and an *address semantic* field, which describes how to use the list. An OA element contains two parts, a 32-bit *address type* field that indicates the type of address that is contained in the address, and the address itself, whose size and format vary depending on the address type. The address semantic field is intended to encapsulate various forms of multicast and replicated communication.

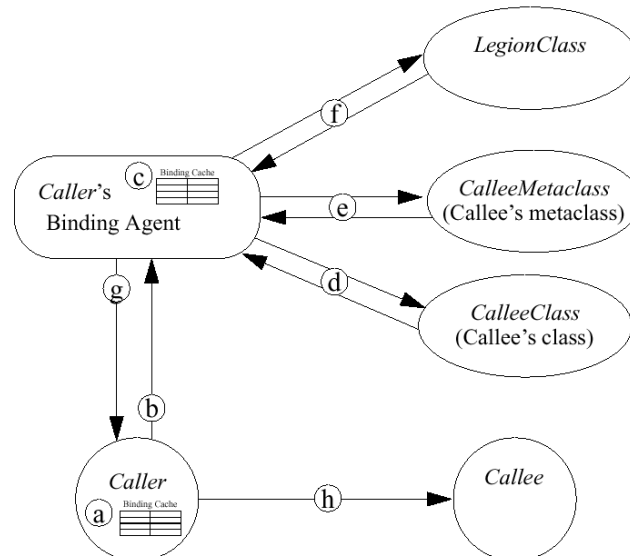


Figure 2: The Legion Binding Mechanism

A binding is implemented as a three-tuple, consisting of a LOID, an OA, and a field that specifies the time at which the binding becomes invalid (including never). Bindings are first-class entities that can be passed around the system and cached within objects. A binding may still be used after the timeout appears to expire at a client—the binding may simply no longer be valid, leading to a communication timeout and rebinding. On the other hand, a client could use the timeout information to schedule rebinding in advance in order to avoid communication delays. Thus, the fact that there is no globally accurate notion of time does not affect correctness, just performance.

Figure 2 depicts a Caller binding the LOID of Callee to an OA. The Caller may already have a cached binding for Callee (a), or it may need to consult a binding agent (b). The binding agent may have a cached binding for Callee (c), or may need to consult Callee’s class, *CalleeClass*, for the binding (d). In order to communicate with *CalleeClass*, the binding agent needs a binding for *CalleeClass*. If the binding agent does not have *CalleeClass*’s binding, it may need to consult *CalleeClass*’s metaclass (e). If the binding agent does not know the binding for this metaclass, the process repeats itself. The recursion is guaranteed to terminate at the root of the binding tree, *LegionClass* (f). Eventually, the binding agent returns Callee’s binding (g) and the Caller can send messages directly to Callee (h).

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 by their classes. If the host on which an object resides fails, but the internal state of an object is still accessible, a class’s object may restart it on another host.

Classes may act as replication managers by mapping one LOID to a number of OAs, likely referring to objects residing on different hosts. In many respects, the class object is a logical replication manager as its instances would likely employ the same replica consistency policies. However, by entrusting the class object with more responsibility, the system will increase the load on class objects. Means of ensuring that individual objects do not become bottlenecks are discussed in Section 3.4. Another approach is to push the replication mechanism up one level in the naming hierarchy, to the *ContextObjects*. This more distributed approach is discussed in Section 3.5.

3.3 Security

Unlike traditional operating systems, 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.

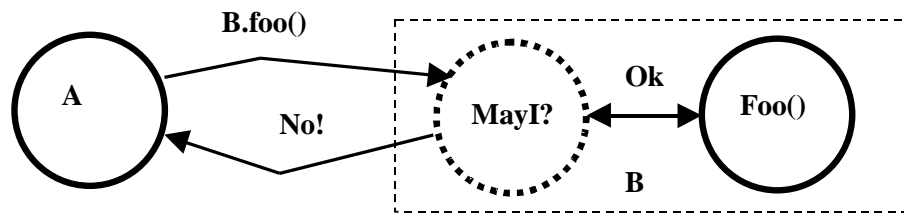


Figure 3: Legion MayI

As described in Section 3.2, the name of a Legion object includes a public key. The public key enables a sending object to optionally encrypt data in a message to a recipient object with assurances that only the recipient object can decode the data. Similarly, the public key of a sending object can be used by the receiving object to verify digitally-signed transmissions. A fundamental component of the security infrastructure is that two objects are free to negotiate the per-transaction security level on messages, such as full encryption, digital signatures, or simple cleartext.

Legion *credentials* [Fer98] are used to authenticate a user and base access control decisions. When a user authenticates to Legion—currently via password—the user obtains a short-lived, unforgeable credential uniquely identifying the person. A person may possess multiple credentials,

signifying multiple roles. When a user invokes an operation of another object, the user's credentials are encrypted and transmitted to the target object. On the recipient side, the security layer unmarshalls the credentials and passes them through the `MayI` method as shown in Figure 3. For each credential passed, `MayI` determines whether the following criteria are satisfied: the credential has not expired, the invoked object is a member of a class listed in the class restrict list, the intended method is enumerated in the methods list, and the credential is signed. Valid credentials (satisfying these criteria) are pooled together. By default, authorization is determined by an Access Control List (ACL) associated with each object; an ACL enumerates the operations on an object that are accessible to specific principals (or groups of principals). If the signer of any of these credentials is allowed to perform the operation, the access is permitted.

The flexibility provided by the Legion security infrastructure is crucial to the design of LegionFS. Per-method access control facilitates a much finer granularity than traditional UNIX file systems. The access control policy for a file object can be set from any location in LegionFS. Although not currently exploited, the access control language can be extended on a per-object basis to further constrain access beyond simply “allow” or “disallow”, such as to support time-of-day restrictions. No special privilege is necessary to create a group of users upon which to base data access (any user can create a group). A consumer of data held in a file has the flexibility to dynamically modify the security of the information transmitted from the file to the consumer—for example, if the consumer is on the same cluster as the file itself, message confidentiality may not be necessary for the transmission because the data never actually leaves the cluster. However, if the client were to move to a geographically distant location, it might request encryption in a minimum number of bits. 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 `sysadmin`).

3.4 Scalability

The object-based architecture of LegionFS frees it from limitations of centralized services by providing a fully distributed, peer-to-peer infrastructure. This has obvious consequences for performance, scalability, fault tolerance, and availability. We have described the opportunity for fault tolerance and availability above and revisit the issue in the form of replication in Section 3.5. Our focus in this section is the scalability afforded by the object model.

Unlike traditional volume-oriented file and storage systems, LegionFS distributes files and contexts (and therefore load) across the available resources in the system. This allows applications to access files without encountering centralized server hot spots 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, or files bound for a particular context, are created by default 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 in general. It also isolates user file accesses to achieve maximum efficiency. A user may employ the replication techniques described elsewhere in this paper to tolerant 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.

Users who wish to remain oblivious to scheduling decisions at any level still benefit from the distributed placement of file and context objects, as LegionFS makes use of a default scheduling object. The scalability of LegionFS allows the user to remain ignorant of the constraints of physical disk enclosures, available disk space or file system allocations, host architectures, security domains, or geographic regions. Additional storage resources are seamlessly incorporated into the file system by administrators. By simply adding a storage subsystem to a context of available storage elements, the

additional space is advertised to the system and becomes the target for placement decisions. The underlying characteristics of the storage system (i.e. disk configuration or file system) is masked from the user. By utilizing multiple, distributed storage systems, LegionFS effectively provides disk parallelism. Because we expect load to be distributed across these disks, the performance of individual disks or storage arrays is less of a concern and LegionFS thrives on cost-effective solutions such as IDE devices. While in general, Legion seeks to provide transparencies sheltering users from the heterogeneity of the distributed system, we describe scenarios below in which sophisticated users may wish to exploit these peculiarities.

It is important to note that LegionFS achieves its scalability through the Legion object model and distributed naming. While some designs rely on extreme forms of metadata replication to achieve availability and scalability in clusters, we believe that this approach is not appropriate in general. Nevertheless, LegionFS utilizes multiple levels of caching to facilitate efficient file and directory lookups and employs limited forms of replication. The distributed binding process and associated caches were described above. Aside from their role in the binding process, binding agents cache translations between context names and LOIDs. Infsd similarly caches translations to avoid excessive RPCs. Despite its scalable design, 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 hot spots, management responsibilities are distributed across ‘clones’ of a particular class. Such clones govern over the a shared pool of instances and replicated upon creation but may diverge as they manage independent instances.

3.5 Extensibility

Objects export interfaces and are characterized and classified according to that interface. LegionFS does not differentiate between objects according to their implementation, but rather their exported interface. For example, LegionFS treats any object providing the standard BasicFileObject interface as 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 abstraction can be used to wrap different implementations. Further, an interface may be augmented to provide functionality in the form of additional methods. So long as an object supports the bare minimum interface requirements of a particular class, it may be used as an instance of that class.

We believe such an ability is vital to a file system intended to span a multitude of requirements. Providing excessive and heavy-weight functionality (such as consistency and replication) in all file and contexts objects is inappropriate as some applications do not 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 can then be implemented only in the objects that require it, without impeding objects and applications that do not.

The Legion design team has taken advantage of this extensibility in a number of object implementations. Those most germane to LegionFS are the ProxyMultiObject, the TwoDFileObject, and Simple K-Copy Classes (SKCC). ProxyMultiObjects are container objects exporting the interfaces of BasicFileObjects, ContextObjects, and their associated classes. ProxyMultiObjects were introduced above.

TwoDFileObjects are an example of a domain-specific implementation intended to serve the scientific community, but applicable on a broader scale. A TwoDFileObject is actually the front-end to a number of BasicFileObjects. It overrides the BasicFileObject interface so that reads and writes are striped across the constituent, underlying BasicFileObjects, arranged as a two-dimensional matrix. A parallel file interface provides a convenient access to applications performing matrix operations. Further, the two-dimensional design degenerates to striping that may be appealing to applications seeking high-performance I/O.

SKCC implement a wrapper around standard classes to provide fault tolerance. SKCC replicate a class'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 occurs during object deactivation. This simple approach provides a good measure of fault tolerance with a minimum of performance degradation.

We recognize the need for more full-featured replication and consistency guarantees in some environments. Here we propose the means of extending LegionFS files and contexts to provide this functionality. In Section 3.2 we considered using specialized replication LOIDs to effect replication via classes. It is also possible to extend ContextObjects to perform replication management. The replication mechanism itself is simple: instead of mapping context names to LOIDs one-to-one, ContextObjects could provide a one-to-many context name to LOID translation. As with the class object replication manager, the ContextObject could perform replica selection based on availability and/or network connectivity constraints.

File data consistency is not addressed by the basic Legion mechanisms, because currently no Legion object caches file data. The initial implementation of Infsd, which serves as the access point to LegionFS, provides NFS-like consistency semantics. That is, it caches data for a configurable amount of time before revalidating file metadata via a stat call. Many may consider consistency guarantees fundamental to a file systems design. However, there are important classes of domains and applications where consistency guarantees are not appropriate, for example large read-only scientific data sets. 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. In such a case, we propose that an object grant leases [Gra89] as they are more scalable than simple callbacks [How88]. However, any scheme could be implemented.

3.6 Adaptability

A file system should not only accommodate of a range of application requirements, but also adaptable to a diverse set of network, load, and system-wide conditions. LegionFS facilitates adaptation via the maintenance of system-wide metadata. Each object has an associated, arbitrary set of <key,value> pairs. Attributes tend to vary according to class. Typical attributes for a host object include load averages, number of active objects, architecture, operating system, and number of processors. This list could easily be extended to include other factors which might effect 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 push their state information to the Collection. More sophisticated monitoring facilities such as the Network Weather Service [Wo199] could also be employed.

The Collection allows applications to track the dynamics of the system as well as capitalize on its more stable, inherent diversity. For example, a geographically-distributed system is likely to contain a range of heterogeneity in the form of underlying file systems, storage devices, and architectures that would not be seen in a smaller-scale environment. If the characteristics of an application (and the data it accesses) are well-known, possibly through explicit user enumeration, the application may achieve better performance through placement that matches these needs against the properties of particular resources. As specific examples, XFS [And95] provides benefits to streaming applications by allowing them to circumvent standard kernel buffer caches and RAID enclosures may provide more efficient availability to concerned applications than could 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 might include the ability to migrate due to network conditions or replicate due to 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 [Sat96].

Being an active entity, a file object may characterize its access pattern in an attempt to prefetch. The file system literature is replete with prefetching mechanisms [Cao95, Cur93, Kro96, Mad97, Pat95]. Often efficiency is a concern as the mechanisms must be realized within the limited time and constraints of a single file system. Assuming fair load distribution, a file object or ProxyMultiObject is more likely to experience idleness than a centralized file server. Further, it is less memory-constrained and may thus retain more exact data concerning access patterns and prefetch schedules than an implementation facing data infidelity due to compression. A file object could provide access hints [Pat95] to LegionFS so that the file system can prefetch data across the network. Of course, a file object would also utilize this information to prefetch from the local disk.

Golding et al. discuss means of exploiting idle periods in computer systems [Gol95]. Another interesting example can be found in archiving inactive data. Like the HP AutoRAID system [Wil95], 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.

4 Evaluation

A complete evaluation of LegionFS including extensibility, adaptability and ease of use is premature and beyond the scope of this work. Instead, this section presents a more quantitative performance evaluation by considering scalability in LegionFS, and compares LegionFS performance to the volume-oriented file system approach. We present the results of micro-benchmarks designed to stress the overall scalability of (a) LegionFS, (b) a Legion-ready version of NFS called `lnfsd`, and (c) NFS. The testing results support the claim that the LegionFS architecture is more suitable for scalable file system performance. LegionFS and `lnfsd` demonstrate a linear increase in aggregate throughput in accordance with the linear growth of the network, whereas NFS performance does not scale well beyond a few clients.

The Legion NFS daemon, `lnfsd`, is a modified user-level NFS daemon interposed between NFS kernel requests and Legion and is based on a NFS Version 2 NFS server. By using `lnfsd`, a user can invoke normal UNIX routines (such as “`cat`”) to access Legion file objects. To provide a reasonable performance evaluation, `lnfsd` was written with efficiency in mind and avoids unnecessary network usage by performing asynchronous write-behind and prefetching. Blocks are written to a block cache and written back 30 seconds later. When `lnfsd` notices that a file is being accessed sequentially, it will perform readahead on that file. This is done asynchronously if the user requested read is completely satisfied by the cache and merged with the demand read otherwise.

To service a user's request, `lnfsd` must have the same rights as that user. Legion stores a user's credentials in `/tmp` and ensures they are accessible only to that user. The daemon runs as a privileged process in order to read a user's credentials from disk and package them with messages sent on that user's behalf. To alleviate the known security holes of NFS Version 2, `lnfsd` and the NFS client are mandated to be co-located on the same host and `lnfsd` only accepts connections made from a reserved port.

To provide statelessness, early NFS file handles were derived from a well-known and deterministic algorithm. This would allow a malicious, remote user to perform actions against `lnfsd`. To overcome this deficiency, `lnfsd` does not send file handles in the clear, but rather encrypts them with its private key. We have found this user-level implementation convenient for debugging. However, it suffers significant overhead. For example, we found that for a typical NFS write request (which simply dirties a cache block), fully two-thirds of the time spent servicing the request was needed to read the arguments and request from the socket and respond to the kernel. The communication between the NFS client and `lnfsd` is pure overhead. Nothing in our design ties us to this particular prototype or to NFS, so a kernel implementation is certainly possible.

The set of micro-benchmarks examines file system performance for large reads. Each reader in the test accesses a separate 10 MB file and performs a series of 1 MB reads on the file. To test scalability, we vary the number of readers per test that simultaneously perform file reads. The results demonstrate the potential bandwidth of a scalable file system.

The benchmark set runs on a cluster of 400-Mhz dual-processor Pentium II machines running Linux 2.2.14-1.3.0smp with 256 MB of main memory and IDE local disk. These commodity components are part of the Centurion cluster at the University of Virginia, and are networked via 100 Mbit/sec ethernet switches connected to 1 Gbit Myrinet. Each reader and its associated target file are always placed on separate nodes, and they share the same switch whenever possible. This placement policy necessitates network communication for file access, but distributes the network traffic, which reduces stress on the network by avoiding excessive communication.

The LegionFS testing environment consists of a Legion network running on 100 nodes of the Centurion cluster. This network provided opportunity to scale the benchmark to 50 readers accessing files on 50 separate nodes. For the Infsd testing, Infs daemons run on 50 nodes in a Legion net in a peer-to-peer fashion, and 50 separate nodes host the readers. The NFS test environment used a single nfs daemon to service file system requests from 50 separate readers. Caching file data is allowed on the file side only, and this remains consistent over each scenario.

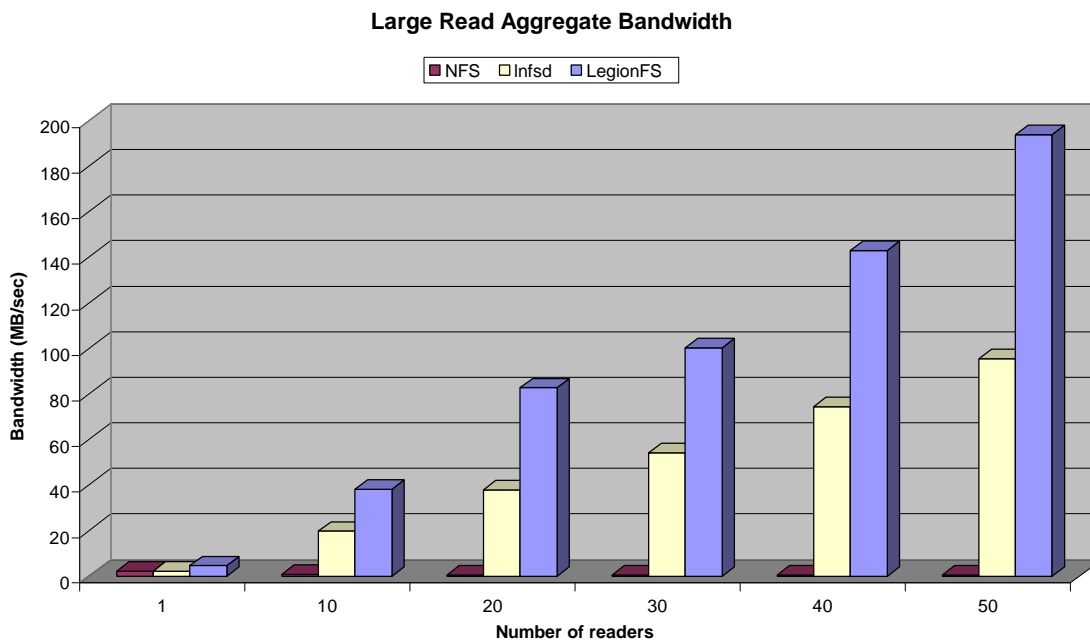


Figure 4: Scalable read performance in NFS, Infsd, and LegionFS. The x axis indicates the number of clients that simultaneously perform 1 MB reads on 10 MB files, and the y axis indicates total read bandwidth. All results are the average of multiple runs.

Figure 4 depicts the aggregate read bandwidth under the three scenarios. For single reader read operations, LegionFS produces a bandwidth of 4.53 MB/sec, which is 2.19 times larger than the 2.07 MB/sec bandwidth of NFS. This difference is due to LegionFS performance tuning for large reads. LegionFS uses a sliding window protocol for message passing in network file transfers that allows multiple in-flight packets, whereas NFS does not. The number of in-flight packets can be adjusted to complement the amount of network traffic. Also, NFS is limited to 4K transfers over the network, whereas LegionFS can use variable transfer size between 8K and 1MB. Larger transfer sizes will increase performance for large reads, since it results in fewer overall transfers. Infsd performs slightly better than

NFS in the single reader case, providing 2.19 MB/sec bandwidth, or 1.06 times the bandwidth of NFS. This can be attributed to the performance tuning in Infsd, which performs read-ahead for sequential file access. The difference between Infsd and LegionFS is due to the extra overhead of interfacing with both NFS and Legion.

LegionFS peak performance was found at 50 readers, yielding an aggregate bandwidth of 193.80 MB/sec. Infsd also experienced peak bandwidth at 50 readers, yielding 95.42 MB/sec aggregate bandwidth. NFS peak performance was found at 2 readers, yielding aggregate bandwidth of 2.1 MB/sec. NFS does not scale well with more than two readers, whereas both Infsd and LegionFS scale linearly with the number of readers. This is largely due to the peer-to-peer nature of LegionFS, and the volume-oriented nature of NFS. The centralized NFS file server experiences saturation at a very low number of readers, creating a file system bottleneck. The distributed nature of the peer-to-peer file systems yield scalability without signs of any bottlenecks.

Server-side caching may improve NFS performance for a small number of readers, but when the number of readers increases, the cache cannot accommodate all associated files. The extra time spent thrashing may actually detract from performance. The file caching in LegionFS aids in performance even when the number of readers increase, because the caches are distributed throughout the system.

Because of the linear nature of the performance results, we would expect LegionFS and Infsd to continue to scale as the number of readers increase. We expect LegionFS to scale similarly in wide area, especially since the sliding window message passing protocol make the file system adaptable to longer-latency communications, which are characteristic of wide area networks.

5 Conclusions

As file system usage patterns continue to mature, a need has arisen for extensibility, scalability, security and performance in a uniform file system infrastructure. LegionFS has been designed and implemented to meet these goals. LegionFS collects system-wide metadata, allowing file system components to modify behavior according to dynamic changes in the system. Its object-based design allows components to be extended and specialized to support novel semantics and implementations as needed. The peer-to-peer architecture eliminates performance bottlenecks by eliminating contention over a single component, yielding a scalable file system. Flexible security mechanisms are part of every file system component in LegionFS, which allows for highly-configurable security policies. The three-level, location-transparent naming system allows for file replication and migration as needed.

The use of LegionFS has been demonstrated with the Legion object-to-object protocol as well as Infsd, a user-level daemon designed to exploit UNIX file system calls and provide an interface between NFS and LegionFS. The scalability of LegionFS has been compared to NFS through micro-benchmark testing, showing that LegionFS and Infsd experience a linear increase in aggregate throughput in accordance with the linear growth of the network. **In the final version of this paper, we will have wide-area tests and a macro-benchmark scalability test.**

References

[Ale97] Alexanandrov, Albert, Maximilian Ibel, Klaus Schauer, and Chris Scheiman. Extending the operating system at the user level: the UFO global file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 77-90, 1997.

[And95] Anderson, Thomas E., Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, pp 109-126, Copper Mountain, CO, December 1995.

[Cam87] R. H. Campbell, G. Johnston, K. Kenny, G. Murakami, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). In *Fourth Workshop on Real-Time Operating Systems*, pages 12-18, Cambridge, Mass., July 1987.

- [Cao95] Cao, Pei, Edward W. Felten, Anna R. Karlin, and Kai 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.
- [Cor93] Corbett, Peter, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pp 472-481, November 1993.
- [Cur93] Curewitz, Kenneth M., P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 1993.
- [Fer98] Ferrari, Adam, Frederick Knabe, Marty Humphrey, Steve Chapin, and Andrew Grimshaw. A Flexible Security System for Metacomputing Environments. In *Seventh International Conference on High Performance Computing and Networking Europe (HPCN Europe 99)*, pages 370-380, April 1999.
- [Gol95] Golding, Richard, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of the USENIX Technical Conference*, pp 201-212, New Orleans, LA, 1995.
- [Gra89] Gray, Cary and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth Symposium on Operating System Principles*, pages 202-210, December 1989.
- [Gri99] Grimshaw, Andrew S, Adam Ferrari, Frederick Knabe, and Marty Humphrey. Wide-Area Computing: Resource Sharing on a Large Scale. *Computer*, 32(5):29-37, May 1999.
- [Gri97] Grimshaw, Andrew S. and William A. Wulf. The Legion Vision of a Worldwide Virtual Machine. *Communications of the ACM*, 40(1):39-45, January 1997.
- [Gri98] Grimshaw, Andrew S, Michael Lewis, Adam Ferrari, and John Karpovich. Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems. Department of Computer Science Technical Report CS-98-12, University of Virginia, June 1998.
- [Gro99] Grönvall, Björn, Assar Westerlund, Stephen Pink. The Design of a Multicast-based Distributed File System. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, Louisiana, February 1999.
- [How88] Howard, John H., Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.
- [Ji00] Ji, Minwen, Edward W. Felten, Randolph Wang, and Jaswinder Pal Singh. Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services. In *Proceedings of the Fourth USENIX Windows Systems Symposium*, August 2000.
- [Kaz90] Kazar, Michael, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. WnTHONY Mason, Shu-Ysui Tu, and Edward R. Zayas. Decorum file system architectural overview. In *Proceedings of the Summer 1990 USENIX*, pages 151-163, Anaheim, CA, 1990.
- [Kro96] Kroeger, Thomas M. and Darrell D. E. Long. The case for efficient file access pattern modeling. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.
- [Lew96] Lewis, Michael and Andrew Grimshaw. The Core Legion Object Model. *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, CA, August 1996.
- [Mad97] Madhyastha, Tara M. and Daniel 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.
- [Mar96] Martin, Cliff, P.S. Narayanan, Banu Ozden, Rajeev Rastogi, Avi Silberschatz. The Fellini Multimedia Storage Server. In *Journal of Digital Libraries*, 1998.
- [Maz99] Mazieres, David, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating Key Management from File System Security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, South Carolina, December 1999.

- [McK84] McKusick, Marshall K., William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.
- [Nel93] Nelson, Michael, Yousef Khalidi, Peter Madany. The Spring File System. Sun Microsystems Research, Technical Reports, TR-93-10, February 1993.
- [Nel88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite File System. *ACM Transactions on Computer Systems*, pages 6(1):134--154, February 1988.
- [Neu94] Neuman, B. Clifford and Theodore Ts'o. "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications Magazine*, vol. 32, no. 9, Sept 1994, pp. 33-38.
- [Nie96] Nieuwejaar, Nils and David Kotz. The Galley parallel file system. In *Proceedings of the Tenth ACM International Conference on Supercomputing*, pp 374-381, Philadelphia, PA, May 1996.
- [Nob97] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K. Agile Application-Aware Adaptation for Mobility. *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.
- [Pat95] Patterson, R. Hugo, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 79-95, Copper Mountain, CO, December 1995.
- [San85] Sandberg, Russel, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119-130, Portland, OR, 1985.
- [Sat90] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9-21, May 1990.
- [Sat96] M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, Vol. 3, No. 1, February 1996.
- [Sun88] Sun Microsystems, NFS: Network File System Protocol Specification, Request for Comments 1094, 1988.
- [Vah98] Vahdat, Amin. *Operating System Services for Wide-Area Applications*. PhD thesis, Department of Computer Science, University of California, Berkeley, December 1998.
- [Wal83] Walker, Bruce, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. *The LOCUS Distributed Operating System*. Proceedings of the 9th Symposium on Operating Systems Principles (SOSP), November 1983, 49-70. Published as ACM SIGOPS Operating Systems Review 17, 5.
- [Wil95] Wilkes, John, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*. December 1995.
- [Whi00] White, Brian S., Andrew S. Grimshaw, Anh 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.
- [Wol99] Wolski, Rich, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *The Journal of Future Generation Computing Systems*, 1999.