Grid-Based File Access: The Legion I/O Model

Brian S. White Andrew S. Grimshaw Anh Nguyen-Tuong
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{bsw9d,grimshaw,an7s}@cs.virginia.edu

Abstract

The unprecedented scale, heterogeneity, and varied usage patterns of grids pose significant technical challenges to any underlying file system that will support them. While grids present a host of new concerns for file access, we focus on two issues: performance and usability. We discuss the Legion I/O model and interface to address the latter area. We compare Legion and Globus I/O against a baseline to validate the efficiency of existent grid-based file access solutions.

1. Introduction

The advent of high-speed networks coupled with a desire to harness more processing power and access immense data stores has lead to the possibility and necessity of federating such resources into computational grids [6, 7, 5]. The unprecedented scale, heterogeneity, and varied usage patterns of such grids pose significant technical challenges to any underlying file system that will support them. Many well-known distributed file system problems, such as security, scalability, performance, and usability, are exacerbated by a grid environment.

Based on our experiences with grids and their users, we highlight two aspects of grid-based file systems: performance and usability. Grid file systems face the daunting task of providing low response times and high throughput in a wide-area environment. Usability is an acute concern because the computational resources of grids are attractive for legacy scientific applications. It may be impossible or exceedingly difficult to re-tool these applications for a grid environment.

Some users will want to be sheltered from the grid environment. Ideally, their method of system interaction (both at the command line and API) *need not* change from whichever system they are accustomed. On the opposite end of the computing spectrum are sophisticated users and application programmers who will prefer a rich interface to take full advantage of the grid's potential. A grid environment should accommodate both types of users.

This paper examines the I/O infrastructure and performance of Legion, comparing it with Globus. Legion [7] is an object-based grid operating system charged with reconciling a collection of heterogeneous resources, dispersed across a wide-area, with a single virtual system image. Legion provides resource management, scheduling, and other system-level tasks, as does any operating system; however, it does so on a much wider scale. Built from the ground up, Legion addresses such issues as scalability, programming ease, fault tolerance, security, and site autonomy.

Legion provides a remote access capability which attempts to address the standards delineated above. We show that Legion achieves 55-65% of ftp's write bandwidth and 70-85% of ftp's read bandwidth for mass transfers. For transfer sizes less than 1 MB, Legion performance suffers owing to its protocol overhead.

Related work is summarized in Section 2. In Section 3, we discuss the Legion I/O model and define terminology. To address the issue of usability, we present the Legion I/O interfaces in Section 4. Next, we present server implementations in Section 5, to motivate a discussion of their performance in Section 6. Finally, we conclude with Section 7.

2. Related work

While space does not permit a detailed scrutiny of the rich literature in distributed file systems research, we do note the excellent survey by Levy and Silberschatz [11].

I/O performance in the wide area is highly sensitive to transfer size. Initial implementations of NFS [13, 14] artificially restricted transfer sizes based on the virtual memory architecture. AFS [9] transfers and copies entire files, leading to unnecessary traffic when a dataset is partitioned between multiple distributed workers. Further, the cache consistency mechanisms of NFS (regular calls to retrieve attributes via GETATTR) and AFS (callbacks) limit throughput.

Globus [5] GASS provides access to remote files through x-gass, ftp, or HTTP protocols [2]. The HTTP GASS implementation is measured alongside Legion in Section 6. We note that GASS provides a secure transport over HTTPS, which is not treated here.

GASS *stages* remote files to a locally-accessible file system on first open via the `globus_gass_open` function. GASS utilizes whole-file caching so that subsequent operations may be satisfied locally through standard system calls. If a file is opened for writing, it is copied back to its remote store when all active file descriptors are closed via `globus_gass_close`. Under the x-gass protocol, GASS also supports a streaming append operation.

From a user's perspective, the specialized calls are cumbersome. However, files may be manually pre-staged to avoid any necessary changes to legacy codes. The location-dependent URL file name scheme used by GASS is an additional burden.

3. The Legion I/O model

Legion objects represent resources and are active entities (e.g. each may be a process running in a separate address space). For example, HostObjects represent computational resources running in a Legion system and VaultObjects are responsible for maintaining state associated with Legion objects. BasicFileObjects correspond to files in a conventional file system and ContextObjects are analogous to a distributed, rooted directory tree [8].

Legion provides its users with human-readable context names. The name-space is hierarchical and rooted, but disjoint from the Unix file system. Context names are *translated* to location-independent identifiers called LOIDs (Legion Object Identifiers), via ContextObjects. In order to communicate with an object, Legion must *bind* LOIDs to OAs (object addresses), which describe the actual communication endpoint [8].

Objects export interfaces and are characterized and classified according to that interface. For example, any object implementing the BasicFileObject interface is treated as a BasicFileObject. This ability to override methods allows for specialization and extensibility.

4. User interface

In an effort to ease the transition to a grid environment, Legion presents its users with a set of familiar and intuitive interfaces (e.g. command line utilities such as `legion_ls` and C library counterparts such as `BasicFiles_creat()`). In addition, more powerful interfaces, such as a two-dimensional parallel file interface are available.

We recognize that *any* required change in legacy code is undesirable and, in cases where source code is unavailable, infeasible. While a kernel-based Legion file system would be accessible by standard system calls, we avoid operating system modifications to achieve portability. We can achieve a similar effect by interposing a Legion-aware NFS daemon.

4.1. Command line utilities

The Legion command line utilities [16] allow a user to navigate context space and manipulate its structure and the objects it contains in a manner reminiscent of traversing a rooted directory tree. This is accomplished through commands which share a similar look and feel to their Unix-like equivalents. For example, `legion_ls` lists a context. `legion_cat` displays the contents of a BasicFileObject. `legion_cp` copies files within context space or between context space and a native file system. A bevy of other command line tools mirror the remaining traditional Unix utilities.

Two more commands aid in bridging the gap between context space and traditional file systems. `legion_import_tree` recursively copies a local directory tree, creating a Legion object for each subdirectory or file. `legion_export_dir` is a light-weight utility that makes a Unix directory visible in context space, *without* creating stand-alone objects for each contained file and subdirectory. Updates effected via Legion mechanisms are immediately reflected to the underlying file system. Support for this utility is described in Section 5.2.

4.2. Remote I/O interface

The Legion file interface library provides user programs with access to BasicFileObjects [15]; bindings exist for C, C++, and Fortran. To match a user's needs and level of familiarity with Legion programming, we also provide C-like (Section 4.2.1) and low impact (Section 4.2.2) I/O interfaces.

4.2.1. Basic I/O interface

The basic I/O interface includes functions inspired by the C I/O system calls and buffered I/O library. This interface is a wrapper around the BasicFileObject implementation accessible from C or C++. By providing communication stubs, the interface relieves the programmer of the burden of accessing the BasicFileObject directly via the Legion communication primitives. Function names are prefixed by `BasicFiles`, but otherwise follow the naming and argument conventions of their C counterparts.

Owing to the overhead of the unoptimized Legion protocol stack, fine-grained file accesses are relatively expensive. To reduce the frequency of remote procedure calls, applications can use the buffered interface. Further, Fortran bindings for buffered I/O provide interoperability when reading and writing integers, reals, and doubles.

4.2.2. Low impact buffered interface

The Legion low impact interface is aimed at minimizing changes to legacy codes wishing to access context space. An application writer uses `lio_legion_to_tempfile` to transfer the contents of BasicFileObjects into the local file system. The application may then utilize standard system calls to access the file. Finally, any modified files are copied back to context space via `lio_tempfile_to_legion`, in a technique similar to GASS staging [2].

Note that it is possible to avoid making *any* changes to legacy codes by importing the Legion BasicFileObject *before* the application executes, and copying the data back on exit. This technique is used frequently by our legacy applications.

4.3. Legion-aware nfsd

To remove the copy-in/copy-out requirements of the previous mechanism, we submit a less obtrusive approach. In this section, we describe the interposition of an NFS daemon between the kernel client and Legion, which allows unmodified applications to access context space.

4.3.1. Implementation

Our modified, Legion-aware NFS daemon, `lnfsd`, receives NFS requests from the kernel and translates these into the appropriate Legion method invocations. Upon receiving the results, it packages them in a form digestible by the NFS client. The file system is mounted like any NFS file system.

To service a user's request, the daemon must have the same rights as that user. Legion stores a user's credentials in the /tmp file system, accessible only to that user (and root). 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. The daemon is able to usurp the user's rights because Legion utilizes bearer credentials [4] or proxies [12]. Fully aware of the security implications of this approach, we are examining delegation, which restricts credentials to specific users and/or operations.

To prevent the performance degradation inherent in small data transfers, `lnfsd` attempts to communicate with Legion objects using a larger granule. It employs readahead to avoid costly demand fetches. `lnfsd` reads ahead *only* when the current request is within the readahead window. When a user's request can not be satisfied by `lnfsd`'s cache, the daemon prefetches synchronously by appending the readahead request to the demand request. Otherwise, prefetching is done asynchronously.

To avoid small write transfers, `lnfsd` utilizes asynchronous write-behind. Data are flushed from the cache after a configurable delay. Whenever data must be written back to the corresponding Legion object, `lnfsd` attempts to coalesce contiguous blocks.

The execution of duplicate requests, caused by client retransmission, can hurt performance and lead to incorrectness (for non-idempotent requests). To combat these problems, `lnfsd` maintains a request cache [10, 3].

4.3.2. Security

`lnfsd` is derived from an NFS Version 2 [14] user-space server. This early protocol focused on the fundamentals of remote access. Under the most popular and basic authentication mechanism, security in NFS is predicated on mutual trust between kernels and/or root-privileged processes. In such an environment of trust, sophisticated authentication protocols are unnecessary. A client simply supplies the server with a UID on each transaction; the server blindly and blithely obliges. To prevent the most obvious security abuses, an exports file limits the domain of trust to certain hosts. Further, the server can be configured to accept only connections from reserved ports.

Unlike NFS, Legion has targeted security as a major concern from its inception. In providing the flexibility afforded by NFS, it is imperative that we do not introduce any new insecurities. Unfortunately, we are restricted in our ability to maintain this level of security by the fixed interface between the NFS (kernel) client and `lnfsd`. Nevertheless, we are confident that a number of simplifications and assumptions allow us to meet our goal. These are:

1. Legion users on a Legion host trust privileged processes on that host.

2. `lnfsd` only accepts connections made from a reserved port.

3. The NFS client and `lnfsd` are collocated on the same host.

Because a user's credentials reside on disk, Legion is susceptible to rogue root users. This fact leads us to make the first assertion. However, this assumption is not unique to the Legion NFS implementation and so doesn't open any *new* portal for intrusion.

The second item requires that a request be made by a kernel or some other privileged process. (Note that we could

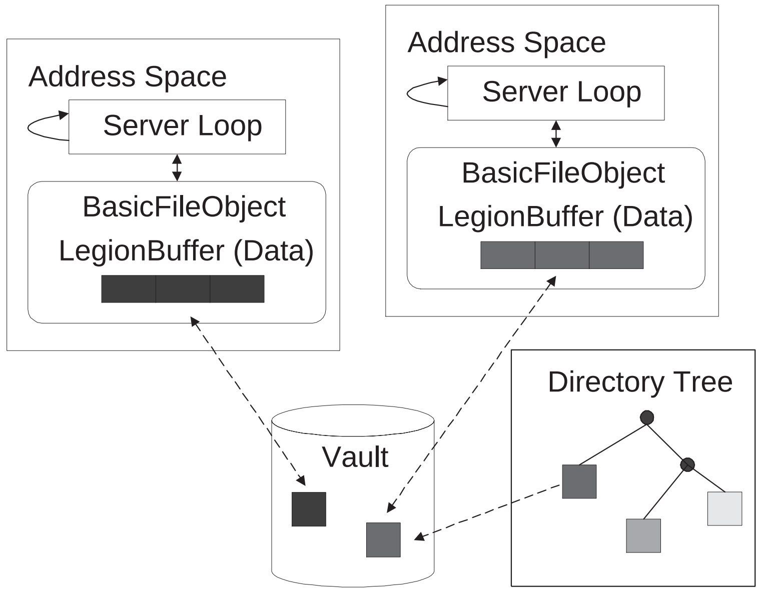

Figure 1. BasicFileObject

easily spoof the source port, but doing so would require being root or modifying a kernel on *some* host.) This allows us to rule out any attacks by non-privileged users.

A client needs a valid file handle to transact with the `lnfsd`. (The root file handle is provided during the mount operation.) Due to the third point, `lnfsd` will only send file handles to addresses on the local host. Therefore, forging an IP address does not aid an intruder as the `lnfsd` will not expose file handles on the wire. This ensures that they can not be sniffed by a malicious user.

4.4. Parallel I/O interface

Legion supports a parallel file interface [15]. This interface allows user-specified striping of data across BasicFileObjects. Such an organization allows multiple clients to access the data without contending with one another at a central server object. Secondly, individual client performance benefits because multiple BasicFileObjects may be accessed concurrently to deliver the desired data.

5. Server implementations

Having highlighted Legion's I/O interfaces, we now describe the implementations that store and manage user data: BasicFileObjects and ProxyMultiObjects. These objects export the BasicFileObject interface and are accessible through the above high-level libraries and interfaces.

5.1. BasicFileObject

A BasicFileObject is a file server, which serves exactly one file. The BasicFileObject polls for incoming requests in a server loop (see Figure 1).

The contents of the BasicFileObject are housed within a LegionBuffer, a random-access array. LegionBuffers provide interoperability between different architectures by encoding metadata on the representation of their contents and performing automatic conversions. Thus, a user need not be concerned with word sizes, byte order, or float-point representations.

BasicFileObjects utilize persistent LegionBuffers which store their data in a VaultObject. This data is not meant to be directly manipulated by the user and has no correspondence to any file rooted in a user's directory tree. Users may copy data from a Unix file to a BasicFileObject in context space. However, the contents of the file and BasicFileObject are independent and may diverge.

5.2. ProxyMultiObject

A ProxyMultiObject encapsulates a context subtree, and therefore serves both ContextObject and BasicFileObject requests. Like the BasicFileObject, it utilizes a single server loop within a single address space. However, the server demultiplexes messages to contained representations of ContextObjects and BasicFileObjects (see Figure 2).

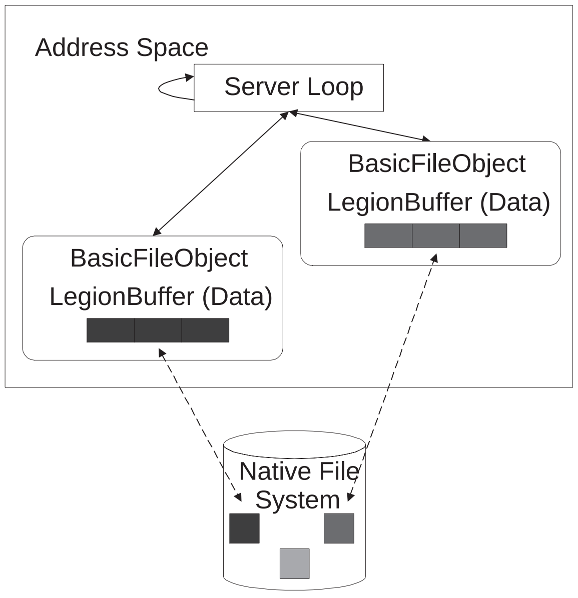

Figure 2. ProxyMultiObject

In another departure from the above BasicFileObject implementation, the LegionBuffers are tied directly to a user's existent files in a rooted directory tree. This ensures that changes to a BasicFileObject's contents are immediately and automatically reflected to the user's file.

6. Performance

The flexibility and convenience afforded by an I/O interface may be less desirable if the perceived cost is an inefficient underlying transfer mechanism. In this section, we present performance studies which compare the grid transfer mechanisms of Globus and Legion against a baseline (ftp). ftp is a reasonable, though optimistic, baseline because the protocol has significantly less overhead than a distributed file system. We hope to approach the throughput of ftp for large, streaming file access, while realizing that the latencies of smaller transfers may well be dominated by the overhead of our approaches. By quantifying this overhead here, we hope to mitigate it in future work.

The performance evaluation environment consists of an SGI Origin2000, equipped with 56 processors and running Irix 6.5, at NCSA in Champaign, Illinois and a dual-processor 400 MHz Intel Pentium II, running version 2.2.14 of the Linux kernel, at the University of Virginia in Charlottesville, Virginia. The two sites are connected via the vBNS (OC-12 network) [1] and several intermediate routers. Throughput is limited by an OC-3 connection from the University of Virginia to the vBNS backbone.

In all experiments, data are transferred from (written to) an NFS mount on the Origin array. That is, the files involved in the ftp and GASS experiments, and the persistent state of the LegionBuffers in the Legion evaluations, reside on an NFS mount. By invalidating the relevant Irix file system buffer cache entries, we ensure that all writes are flushed on file close and that file data does not persist in the cache across trials. Likewise, we prevent client-side caching to ensure remote access to the server objects. Performance measurements do not capture client-side disk accesses.

By default, the Legion communication layer utilizes a start-and-stop protocol built atop UDP. To ensure a fair basis of comparison, we present results using UDP and TCP. In both sets of experiments, security is disabled (as is also true of the GASS runs). Were security enabled, message digesting and certificate passing would contribute additional overhead.

Experiments are characterized from the perspective of the Linux host at the University of Virginia. Therefore, during read experiments, the Linux host reads data from NCSA and simply discards the transferred data. On writes, the Linux host transfers data from memory to NCSA, where they are flushed to disk.

6.1. Protocol overhead

Our first test seeks to expose the fixed cost of connection setup and tear-down of each mechanism (Table 1).

Protocol	Latency (seconds)
ftp-read	0.028
ftp-write	0.026
gass-read	0.087
gass-write	0.147
legion-read-udp	0.325
legion-write-udp	0.487
lio-read-udp	0.507
lio-write-udp	0.631
lnfs-read-udp	0.454
lnfs-write-udp	0.681
legion-read-tcp	0.430
legion-write-tcp	0.632
lio-read-tcp	0.501
lio-write-tcp	0.677
lnfs-read-tcp	0.403
lnfs-write-tcp	0.422

Table 1: File Open Overhead [1]

Cache locking and management contribute overhead to the GASS protocol [2].

[1] In this and subsequent figures, legion refers to the Legion basic I/O interface (Section 4.2), lio denotes the Legion low impact interface (Section 4.2.2), and lnfs signifies Legion-aware NFS (Section 4.3).

The overhead of the Legion mechanisms is significant, and attributable primarily to Legion's location-independent naming scheme. Several context name/LOID translations and LOID/OA bindings are required during this test. In a longer-lived application, these translations and bindings would likely be cached and amortized over a number of Legion RPCs.

In addition to this naming overhead, the Legion basic I/O and low impact interfaces make an expensive (and superfluous) call to the remote BasicFileObject at NCSA. The Legion NFS implementation performs a `GETATTR` RPC to retrieve the BasicFileObject's attributes. It also performs a less expensive invocation within the LAN to determine the attributes of the /tmp context.

6.2. Bandwidth measurements

In this section, we present file transfers of various sizes. Through these experiments we gain an appreciation for the throughput achieved by each mechanism.

The first set of figures compare UDP- and TCP-based Legion mechanisms. Figure 3 clearly shows each TCP mechanism outperforming its UDP counterpart for read operations. In each case, the low impact interface performs similarly to the basic I/O interface. This is not surprising as each mechanism is simply reading data in one megabyte chunks. Governed by the NFS protocol, `lnfsd` periodically queries the remote BasicFileObject to satisfy `GETATTR` requests. Further, the maximum transfer size requests by the NFS kernel client corresponds to a page (4K on a Pentium II). Thus throughput between the kernel and `lnfsd` is limited. For these reasons, Legion NFS performance suffers.

Figure 4 highlights Legion write performance. Unlike the read case, we notice that the low impact interface performs significantly worse than the basic I/O interface under both TCP and UDP. This is to be expected; the file must first be fetched from the remote server, modified locally, and finally written back. Because of this additional data copying, Legion NFS outperforms the low impact interface over TCP.

In Figures 3 and 4, the performance of TCP implementations surpasses the corresponding UDP mechanism. Though TCP outperforms UDP in this wide-area environment, since TCP entails a higher overhead than UDP on a per-connection basis, we would not want to blindly deploy TCP in local-area networks. Future releases of Legion will support dynamic selection of the network transport (TCP or UDP) based on IP addresses.

A comparison of Figure 3 with Figure 4 and Figure 5 with 6 show that writes outstrip reads over TCP. From a file system perspective, a read access may be semantically quite different from a write operation. However informal experiments removed all disk accesses, so that reads and writes were simple network transfers differing only in transfer direction. The anomalies witnessed above did not disappear. This manifests the potential instability and non-uniformity of the environments in which wide-area file systems will need to thrive.

Figure 5 compares Legion read mechanisms against the corresponding GASS and ftp operations. Figure 6 treats writes similarly.

In the case of reads, GASS closely mirrors ftp. This is expected as GASS is a thin veneer over the simple HTTP protocol. The Legion basic I/O and the copy-in/copy-out mechanisms approach the bandwidth of ftp to within 70-85%. Legion's degraded performance is likely attributable to its extensible protocol stack.

Legion NFS performance lags significantly, achieving 30% of ftp's throughput for transfers greater than a megabyte. For small transfer sizes, Legion NFS latencies are dominated by `LOOKUP`s of directory path components and `GETATTR` RPC calls. Given its readahead potential, we would not expect Legion NFS performance to be so abysmal under mass transfers. We hypothesize that as `lnfsd`'s block cache saturates, the implementation is burdened by unoptimized and frequent cache lookups.

For writes, the Legion basic I/O interface outperforms both GASS and the Legion low impact interface. As described above, this behavior is to be expected as the Legion basic I/O interface avoids the copy-in/copy-out semantics of the other two mechanisms.

Legion NFS write performance is consistent with its read bandwidth. A sophisticated block cache contributes considerable overhead and once again appears to be the source of poor write throughput. With a stream-lined cache implementation, Legion NFS should be able to leverage asynchronous write-behind to achieve performance similar to that exhibited by the Legion I/O interface.

Once again, it is interesting to note that ftp writes are significantly more efficient than ftp reads. This suggests that the asymmetric anomaly described above is not a Legion artifact, but rather attributable to the network configuration of our heterogeneous test environment.

The graphs show that mass transfers are more efficient because they incur fewer invocations of the respective protocol and its associated overhead. For example, the overhead of a NULL RPC in Legion has been measured to be 8 milliseconds. This makes small data transfer costly.

7. Conclusions

We presented the Legion I/O interfaces, highlighting their flexibility and intended audiences. The low impact interface and NFS implementations provide support for legacy codes. The Unix-like I/O library and command line

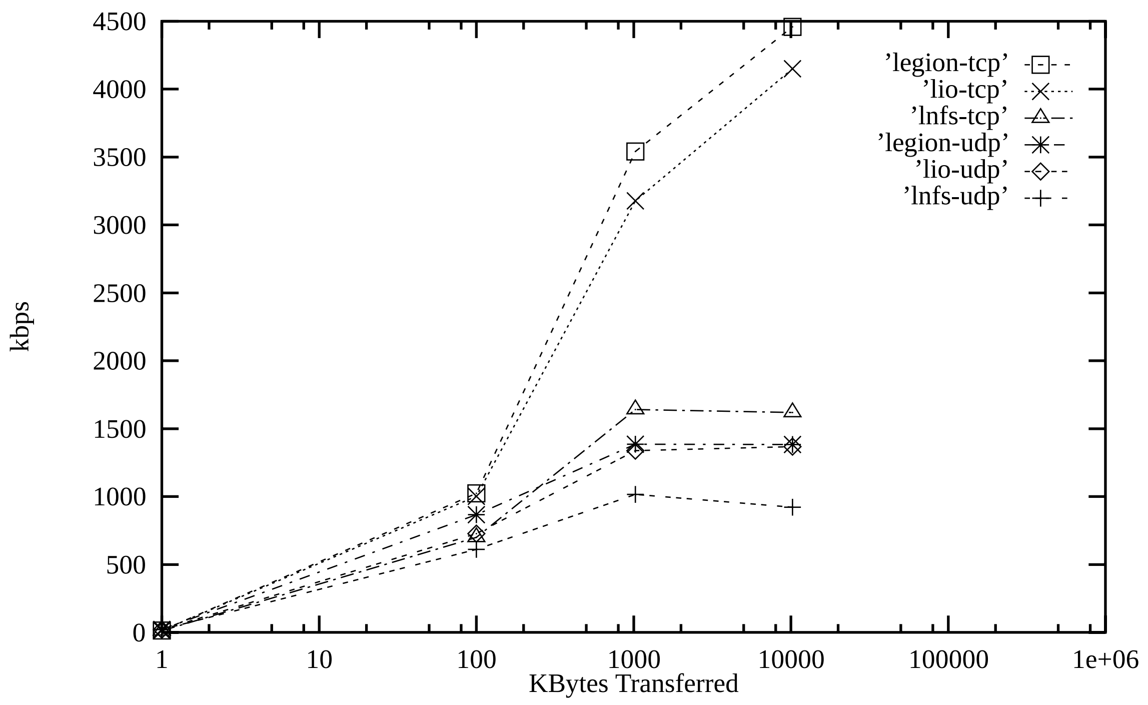

Figure 3. Legion Read Bandwidth

utilities provide a familiar model for many users, easing their transition to a grid environment.

A presentation of Legion and GASS mechanisms compared against a baseline allows us to judge the current performance of grid-based I/O systems. We find that the Legion basic I/O interface (TCP implementation) achieves 55-65% of ftp's efficiency for writes and 70-85% for writes. While NFS Legion performance lags, the ease of use it affords may make it a viable option. Further, the implementation is immature, with several optimizations planned.

Equipped with an understanding of grid-based I/O performance relative to more traditional remote access, we expect to attack the I/O bottleneck from various angles. Our first efforts will quantify the time spent in the Legion protocol. The Legion I/O model offers an opportunity to tailor files to specific file access patterns. This could easily be supported by attribute value tags, such as 'read-only' or 'single-writer'. Taking such properties into account will have a significant performance impact.

Acknowledgments

This work was partially supported by National Computational Science Alliance under grant 5-26025 and utilized the NCSA SGI/CRAY Origin2000. We would also like to thank Mark Morgan for answering a constant stream of questions and John Karpovich who implemented the dynamic network protocol selection in Legion. Marty Humphrey, Fritz Knabe, and Robert Schutt provided valuable insights regarding security in `lnfsd`.

References

[1] http://www.vbns.org, February 2000.

[2] 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.

[3] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: NFS version 3 protocol specification, June 1995. See also RFC1094 [14]. Status: INFORMATIONAL.

[4] 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.

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

[6] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[7] A. S. Grimshaw and et al. Metasystems. *Communications of the ACM*, pages 46–55, Nov. 1998.

[8] A. S. Grimshaw, M. J. Lewis, A. J. Ferrarri, and J. F. Karpovich. Architectural support for extensibility and autonomy in wide-area distributed object systems. Technical report, University of Virginia, June 1998.

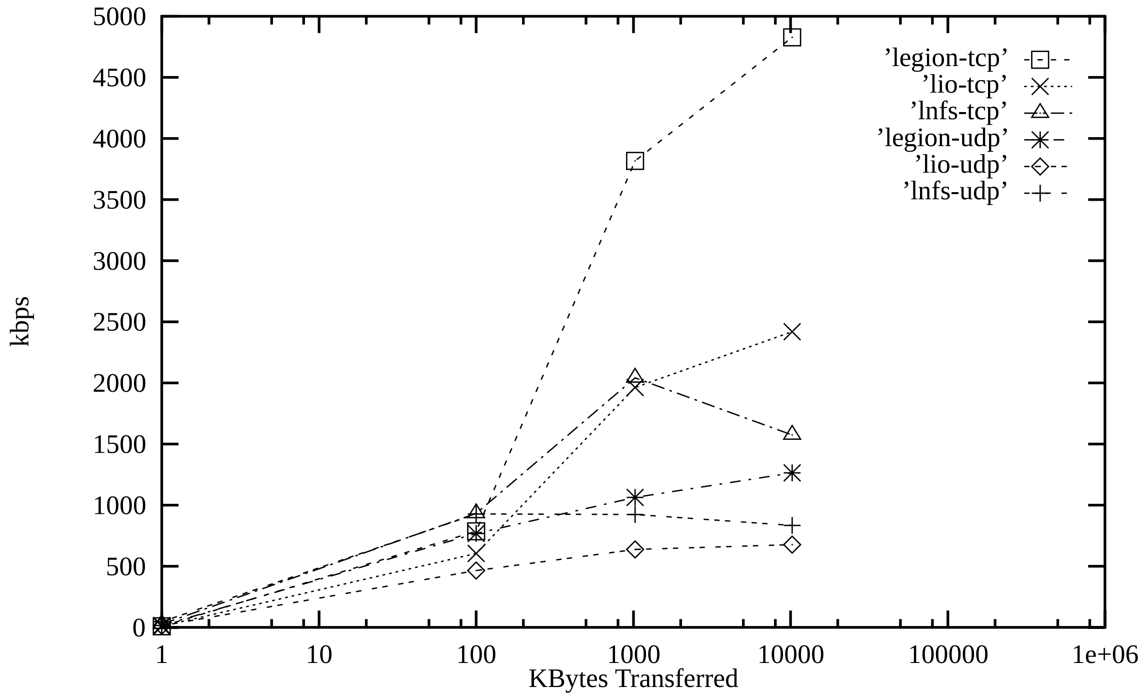

Figure 4. Legion Write Bandwidth

[9] 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 of Computer Systems*, 6(1):51–81, Feb. 1988.

[10] C. Juszczak. Improving the performance and correctness of an nfs server. In *Proceedings Winter USENIX 1989*, pages 53–63, San Diego, CA, January 1989.

[11] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.

[12] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. In *Thirteenth International Conference on Distributed Computing Systems*, pages 283–291, May 1993.

[13] 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.

[14] Sun Microsystems, Inc. RFC 1094: NFS: Network File System Protocol specification, Mar. 1989. See also RFC1813 [3]. Status: INFORMATIONAL.

[15] The Legion Research Group. Legion 1.6 development manual, August 1999.

[16] The Legion Research Group. Legion 1.6 user manual, August 1999.

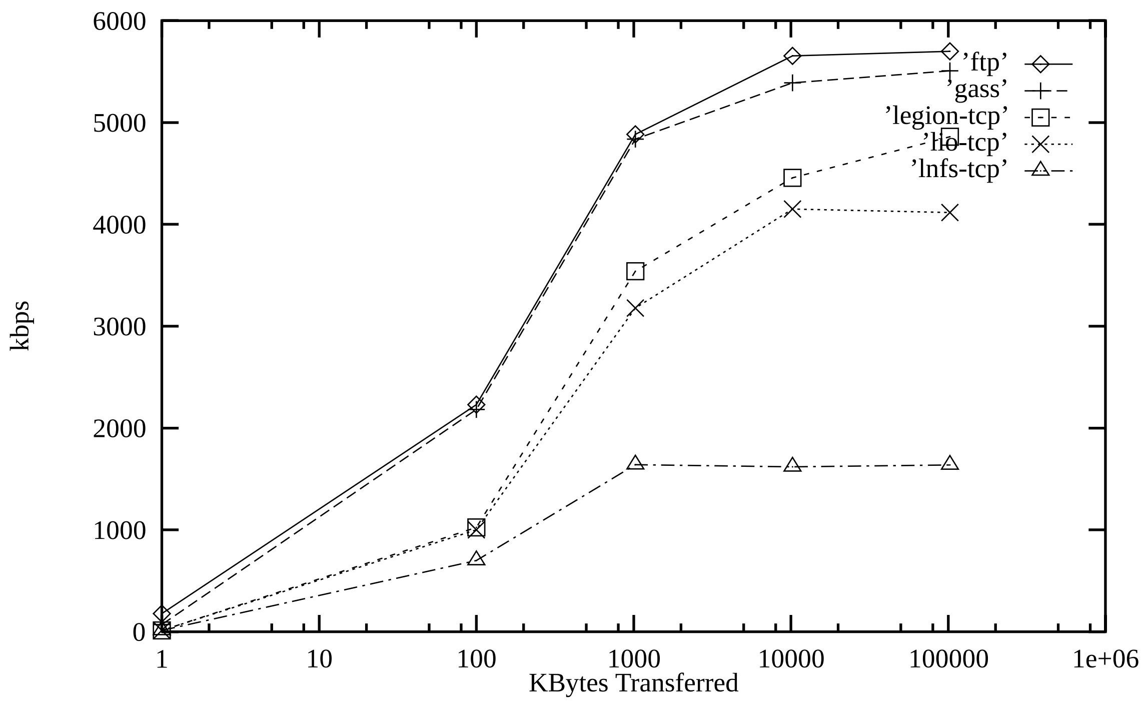

Figure 5. Read Bandwidth

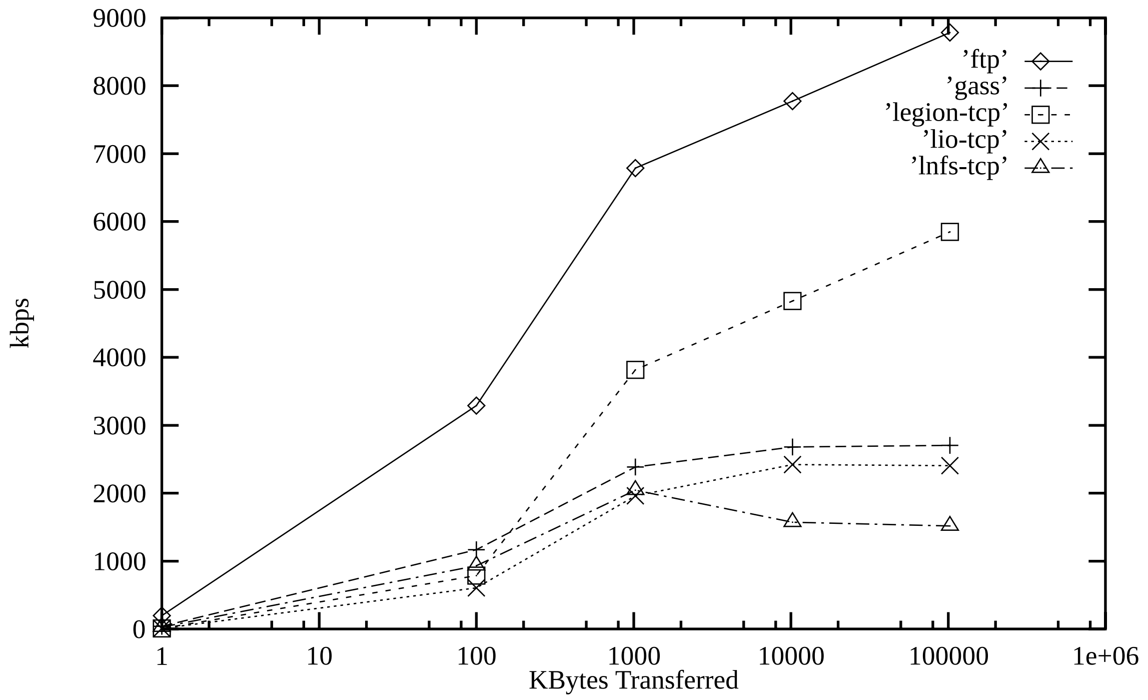

Figure 6. Write Bandwidth