GASS: A Data Movement and Access Service for Wide Area Computing Systems

Joseph Bester* Ian Foster* † Carl Kesselman‡ Jean Tedesco† * Steven Tuecke*

Abstract

In wide area computing, programs frequently execute at sites that are distant from their data. Data access mechanisms are required that place limited functionality demands on an application or host system yet permit high-performance implementations. To address these requirements, we propose a data movement and access service called Global Access to Secondary Storage (GASS). This service defines a global name space via Uniform Resource Locators and allows applications to access remote files via standard I/O interfaces. High performance is achieved by incorporating default data movement strategies that are specialized for I/O patterns common in wide area applications and by providing support for programmer management of data movement. GASS forms part of the Globus toolkit, a set of services for high-performance distributed computing. GASS itself makes use of Globus services for security and communication, and other Globus components use GASS services for executable staging and real-time remote monitoring. Application experiences demonstrate that the library has practical utility.

1 Introduction

A frequent obstacle to the creation of high-performance computations that operate effectively in networked computing systems, or *computational grids* as they are often called [5], is a need to access data that is not colocated with a site at which computation is performed. This problem is challenging because we desire a solution that can simultaneously meet the following requirements:

- impose few requirements on the application programmer, so that applications can be easily modified (or written from scratch) to execute efficiently in grid environments;
- impose few requirements on the resource provider, so that new resources can easily be incorporated into a grid environment; and
- allow high-performance implementations and support application-oriented management of bandwidth, so as to meet the requirements of high-performance applications.

No existing technology meets all three requirements. Distributed file systems (e.g., [16]) provide convenient access to remote data, but require substantial technology deployment and interorganizational cooperation (e.g., cross-realm authentication). Web-based file systems (e.g., [20, 1]) provide transparent access to remote resources, but require specialized kernel capabilities in the target systems. Condor [13] avoids the need for kernel services by relinking with specialized versions of I/O libraries, but only provides access to data on a user's "home" machine. Legion's context space mechanism [8] provides uniform access to Legion objects, but not to data stored in conventional file systems. The RIO remote I/O system [7] uses striping to support high-performance remote access, but requires that applications adopt the MPI-IO parallel I/O library.

In this paper, we describe a new approach to remote data access in which the following strategies are used to address the above requirements:

- We provide mechanisms optimized for common grid I/O patterns, such as executable staging, reading of configuration files, error/diagnostic output, and simulation output. Because we do not attempt to provide the full functionality of a distributed file system, we can increase achieved performance and simplify implementation.
- We define mechanisms that can be implemented without any specialized services (e.g., device drivers or cross-realm security arrangements) at a participating site. Hence, we achieve considerable flexibility in terms of our ability to support dynamic grid communities with varying resource availability and application requirements.
- We provide mechanisms that allow programmers to guide or override default data movement strategies by controlling data source selection, staging, caching, and filtering of data before transfer. These bandwidth management mechanisms allow programmers to optimize performance when required.
- We exploit capabilities provided by the Globus toolkit to support efficient communication and execution in dynamic grid environments where resource location and type may not be known until runtime.

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

†Department of Computer Science, The University of Chicago, Chicago, IL 60637, U.S.A.

‡Information Sciences Institute, University of Southern California, CA 90292, U.S.A.

This approach has been incorporated into a service called Global Access to Secondary Storage (GASS) that forms part of the Globus grid toolkit [4]. Experiments with a range of applications demonstrate practical utility, while microbenchmark studies provide insights into performance issues.

In the rest of this article, we first describe in Section 2 the nature of the grid I/O problem, then in Sections 3 and 4 describe the design and implementation of GASS, respectively. We discuss GASS applications in Section 5 and conclude in Section 7.

2 Background

Consider a scientist interested in the regional impacts of global change. This user wants to run a regional-scale atmospheric circulation model on a remote computer obtained from a national "grid," with boundary conditions provided by the output from a separate global model. The following issues are typical of the I/O requirements that arise in such grid applications:

- The global model data are stored as Hierarchical Data Format (HDF) files at various centers; having identified the grid resource on which the simulation is to run, the user wants to locate and access the "nearest" version.
- When running, the regional model must also access other input files, such as topographic data; these files may be located elsewhere in the grid.
- Diagnostic data must be streamed back to the user as the applications runs.
- Output data must also be stored somewhere: perhaps on the user's home machine, or in one or more national archives.
- In a typical usage scenario, the user runs the same scenario repeatedly while varying parameters: either as part of a formal parameter study, or simply in exploratory mode. Input files should be cacheable at the remote machine to prevent having to reload them for each run of the parameter study. Furthermore, it may be important to cache output files as well, if inter-run comparisons are to be performed.

Analysis of applications such as this leads us to identify six principal requirements for grid I/O services:

Uniform access to files. While data location may be taken into account when selecting resources for a computation, a computation and its input data will often not be colocated. Remote access can introduce many complexities, such as authentication, communication protocols, access mechanisms, naming, etc. (The heterogeneity of mechanism and policy encountered in grid environments means that we cannot assume that the resources used by a computation share a file system, user id space, or common security mechanisms.) These complexities should be hidden from the application programmer, who should be able to use a uniform naming scheme (providing a global name space for files) and the same access mechanisms to access files, regardless of location.

Diverse data sources. A grid environment can provide access to a wider variety of data sources than is encountered on a typical local area network. Hence, uniform access mechanisms should also support such data sources, which may include common stream-based data services (such as FTP or HTTP), files on a remote disk or tape, or dynamic information (such as filtered data).

Dynamic resource set. The communities of users and resources that participate in a grid computation are frequently dynamic, forming for short-lived projects and collaborations. Thus the institutional overhead of accessing remote data should be minimized. For example, a solution that requires that all-pairs security relationships be established between participating sites will be intractable in many cases.

Support for streaming I/O. In the Unix environment, a frequent use of "file I/O" is to pipe data to and from an interactive terminal, via the stdout, stderr, and stdin abstractions. There are significant advantages to supporting the same abstractions in a grid environment, as it allows us to retarget interactive applications to remote computing resources without modification. If a grid application runs on a large number of processors, then specialized mechanisms (e.g., a combining tree) may be required to address scalability concerns when implementing these abstractions remotely.

Little or no program modification. In order to reduce the cost of constructing and maintaining grid applications, remote I/O mechanisms should allow applications to be adapted for wide area execution with few modifications. For example, we would wish to avoid having to replace all I/O calls with specialized "grid I/O" calls.

Support for programmer-directed performance optimization. It is desirable both that common I/O patterns be implemented efficiently, via appropriate default strategies, and that application programmers be able to override default behaviors when this is required for performance optimization.

2.1 Existing Distributed I/O Systems

We review approaches to I/O service support for distributed computations, focusing on those which satisfy (at least partially) the needs of grid applications.

Kernel-level distributed file systems such as the Andrew File System (AFS) [16] and Distributed File System (DFS) provide essentially standard Unix file system operations in a wide area environment. However, because such systems are cumbersome to deploy, particularly on leading-edge supercomputer systems, they do not now (and in our opinion, will never) form a ubiquitous infrastructure. Other difficulties also exist: for example, file access rights for a multi-institutional computation can be difficult to coordinate. Also, because these are kernel-level services, the typical user cannot control performance.

The Prospero File System [17] was designed to integrate heterogeneous file access methods while providing individual users with customized views of file organization. The focus is on organizing resources and on controlling the complexity of resource discovery and management. These are clearly important concerns, but Prospero does not address the performance requirements of the applications that we wish to support.

In the Condor high-throughput computing system [13], application codes are linked with a runtime library that replaces I/O system calls with remote procedure calls back to the computer that submitted the job to the remote machine. This approach provides transparent access to remote files, transfers only requested data and does not require access to local storage on the remote machine. However, the lack of support for caching can lead to significant file access overhead, especially for large files. In addition, the programmer is provided with no bandwidth management capabilities: it is not possible to prestage files or to access alternative data sets with better network connectivity.

In the Legion system [8], global access to data is supported, but only for files that have been explicitly copied into "Legion space," and then only via specialized I/O methods. This restriction limits the utility of the Legion constructs, as typical applications want to use standard methods to access data in a variety of storage systems. Global access is provided via shell commands such as "legion_cp" (the Legion version of the Unix copy command). These commands can presumably be used to implement some of the bandwidth management strategies described in this paper, but Legion papers do not describe such implementations.

WebFS [20] and UFO [1] both use file system primitives to provide access to Web-based data sources, with Uniform Resource Locators (URLs) being used to provide a global file namespace. UFO runs as a user-level application and uses the Unix debugging trap interface to replace Unix file operations with operations that operate on cached copies of URLs, while WebFS uses the Unix vnode [11] to redirect I/O calls to a remote WebFS or HTTP server. While these approaches allow unmodified applications to access remote data, they restrict portability. The WebFS Vnode kernel module, which runs in protected space, calls out to a user-level WebFS daemon. Application-specific redirection and caching policies can be implemented by rewriting the daemon; however, because there is a single daemon per Vnode interface, and WebFS requires that a kernel module be loaded, there are issues with per-user flexibility.

3 GASS Architecture

The GASS service was designed to overcome the limitations of the remote I/O systems described above. The distinguishing features of GASS are optimized support for file access patterns common in grid computations and the ability for user controlled management of network bandwidth. We now examine both of these features in more detail.

3.1 Common Grid File Access Patterns

In designing GASS, our goal was not to build a general-purpose distributed file system but rather to support the following four I/O patterns that we have found to be common in high-performance grid applications. These patterns are distinguished by particularly simple access patterns and coherency requirements:

- *Read-only access to an entire file assumed to contain "constant" data*: for example, the topographic database in our example above, or a configuration file read to determine problem parameters. Such data may be accessed by multiple readers (perhaps by every process in a parallel program), but because accesses are read only and the data is assumed to be constant, no coherency control is needed.

- *Shared write access to an individual file is not required, meaning we can adopt the policy that if multiple writers exist, the file written by the "last" writer produces the final value of the entire file*: for example, a parallel application in which all processes generate the same answer, or in which any answer is valid. Multiple writers must be supported but coherency is simple to implement because it is enforced only at the file level.

- *Append-only access to a file with output required in "real time" at a remote location*: for example, a log of program activity, used perhaps to monitor execution. In the case of multiple writers, output may be interleaved, either arbitrarily or on a record-by-record basis (but with source ordering preserved). Again, the simple access pattern means that coherency control is simple.

- *Unrestricted read/write access to an entire file with no other concurrent accesses*: for example, output data produced by a scientific simulation. Because only one process accesses the file at a time, no coherency control is required.

These operations require no explicit coordination among accessing processes, a property that we exploit in the GASS implementation. In so doing, we explicitly exclude two multiprocess I/O structures that can occur in practice: information exchange by concurrent reading and writing of a shared file and cooperative construction of a single output file via other than append operations.

A second property of our selected operations is that because of the coherency model and the fact that readers and writers are assumed to operate on an entire file, an implementation need not communicate individual read and write operations over the network, but can instead simply transfer an entire file *to* a reading site prior to reading any element, and *from* a writing site after all writing is completed. These strategies implicitly exclude I/O structures in which a process accesses a small part of a file, as the basic GASS structures do not deal with this case efficiently. However, GASS defines functions that can be used to optimize for this case when required.

3.2 Default Data Movement Strategies

Optimal execution of grid applications requires careful management of the limited network bandwidth available. Typically, distributed file systems hide data movement operations from the application programmer. When control is provided, it is focused towards latency management (e.g., block prefetching [12, 18, 10]) rather than bandwidth management.

GASS addresses bandwidth management issues by providing a *file cache*: a "local" secondary storage area in which copies of remote files can be stored. (See Figure 2, which is also discussed in more detail below). By default, data is moved into and out of this cache when files are opened and

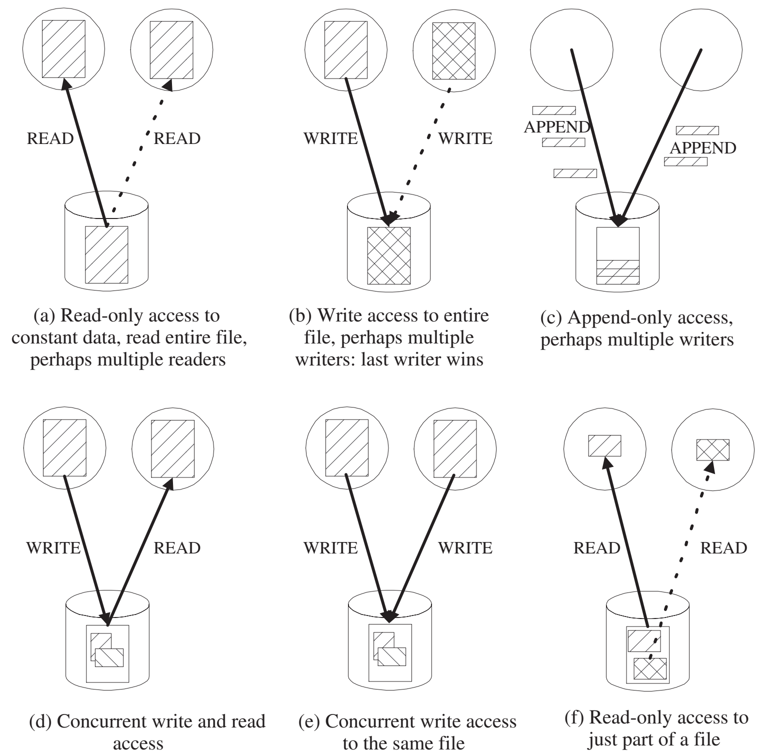

Figure 1: The GASS system is optimized for I/O patterns (a)–(c); patterns (d)–(f) are not supported efficiently

closed (except for files opened in append mode: see below), according to two standard strategies.

The first strategy is to *fetch and cache on first read open.* When a remote file is opened for reading, the local cache is checked and the entire remote file is fetched only if it is not already present. The local copy is then opened, a file descriptor is returned, and a reference count associated with the cached copy is incremented. The file can then be accessed within the application using conventional I/O calls on the cached file.

This first strategy optimizes data movement for a common situation in parallel computing, namely an input file that is accessed by many processes. Because input files are typically read in their entirety, no unnecessary data transfers are performed and a potentially large number of round-trip communications are avoided. However, the strategy may be inappropriate if a file is large: computation may be delayed too long while the file is transferred, or the local cache may be too small to hold the entire file. Alternative strategies such as prestaging and specialized GASS servers can be used in such situations; these are discussed below.

The second strategy is to *flush cache and transfer on last write close.* When a remote file that has been created or opened for writing is closed, the reference count associated with that file is checked. If this count is one, then the file is copied to the associated remote location and then deleted from the cache; otherwise, the reference count is simply decremented. This caching strategy reduces bandwidth requirements when multiple processes at the same location write to the same output file. Conflicts are resolved locally, not remotely, and the file is transferred only once.

As a special case, a remote file that is opened in append mode is not placed in the cache; rather, a communication stream (on Unix systems, a TCP socket) is created to the remote location, and write operations to the file are translated into communication operations on that stream. This strategy allows streaming output for applications that require it.

3.3 Specialized Data Movement Strategies

GASS also provides mechanisms that allow programmers to refine default data movement strategies and to manage how they are applied in particular cases. These mechanisms fall into two general classes: relatively high-level mechanisms concerned with prestaging data into the cache prior to program access and with poststaging of data subsequent to program access; and low-level mechanisms that can be used to implement alternative data movement strategies.

Prestaging and poststaging commands are a natural extension to the cache management mechanisms that we described above. Prestaging and poststaging are useful in the situation that the data-files are very large, or the files are going to be used across multiple runs of a program, for example during a parameter study. A prestaging command is equivalent to an "open file for reading" call; it creates an entry in the cache and transfers the file, if it is not already present, and increments the entry's reference count. Hence, subsequent open calls will find the file already present in the cache. Similarly, a poststaging command is equivalent to a "close file that was open for writing" call. Prestaging and poststaging commands can be called from within an application or externally. A useful consequence of the ability to call staging commands externally is that common access pat-

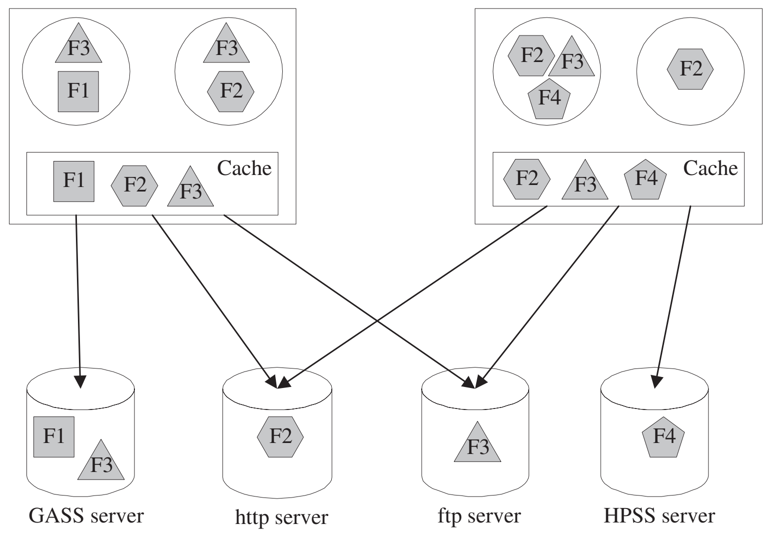

Figure 2: The GASS cache architecture. Files opened by application processes (represented by circles) are maintained in a local cache directory; they are copied from the remote location (on open, if opened for reading) and/or to the remote location (on close, if created or opened for writing).

terns (such as file staging and redirection of standard input and output) can be accomplished completely outside of the application, hence avoiding a need for source modifications.

Low-level cache mechanisms provide more fine-grained control over cache behavior. For example, a user can specify not only *when* but also *where* a file is cached. This capability can be important for two reasons. First, because files accessed by grid applications can be large, the typical distributed file system strategy of placing cached data in a common pre-allocated file area may be impractical: cache usage by different users can interfere with each other and precious scratch disk space must be partitioned between normal and AFS/DFS usage. In contrast, GASS mechanisms allow file caching to be directed to specific locations, on a per-file and/or per-user basis: for example, to access-controlled user file systems.

A second benefit of user-level control of cache location is that it becomes possible for GASS operations to exploit "local" distributed file systems. For example, cached files can be placed in a file system that can be accessed from more then one computational resource via NFS, AFS, or DFS, or into a high-performance cluster file system such as the Distributed Parallel Storage System (DPSS) [19]. In keeping with the Globus philosophy of providing an inter-domain service layer that integrates diverse local services [4], GASS provides a means by which remote file access between resources can be achieved while enabling an application to exploit performance advantages of specific local policy, as required.

3.4 GASS Operation

Grid applications access remote files using GASS by opening and closing the files with specialized open and close calls (on Unix systems, `globus_gass_open`, `globus_gass_fopen`, `globus_gass_close`, `globus_gass_fclose`). These trigger the GASS cache management operations described above to optimize performance based on the default data movement strategies. From an application viewpoint, the GASS open and close calls act like their standard Unix I/O counterparts, except that a URL rather than a file name is used to specify the location of the file data. File descriptors or streams returned from these calls can be used with normal read and write operations: *only* open and close calls need be modified, all other I/O calls can be used unchanged.

The use of specialized open and close calls means that some program modifications are required before an application can use GASS. However, the difficulty of inserting these calls is minimized by ensuring that the GASS call have the same arguments, are semantically equivalent, and are backward compatible to the Unix functions that they replace. Relinking (as in Condor) or kernel support (DFS, WebFS) could be used to avoid the need for application modification, at the cost of some increase in implementation complexity and portability.

A URL used in a GASS open call specifies a remote file name, the physical location of the data resource on which the file is located, and the protocol required to access the resource. An advantage of thus making the location of the file server explicit (unlike for example DFS, which hides file server locations in global file names) is that an application can use domain-specific heuristics to select from among alternative copies. For example, in Figure 2, file F3 is replicated on two different servers; different processes may choose to access the "nearer" copy. The figure also emphasizes the point that GASS can be used to access files stored in a variety of storage devices: specialized GASS servers or FTP servers (already supported) or HTTP, HPSS, DPSS, or other servers (work in progress).

The GASS system also exposes additional lower-level APIs, discussed below, which can be used to implement specialized data movement and access strategies.

3.5 Integration with the Globus Toolkit

GASS services are integrated with other Globus services in a number of mutually beneficial ways. Here we mention just one. The Globus Resource Allocation Manager (GRAM) is the Globus component that is used to allocate computational resources and to initiate and manage computation. An application that wants to initiate computation on a remote resource issues a request to the corresponding GRAM, specifying such parameters as executable name, arguments, and resource requirements. The availability of GASS services has made it straightforward to extend the GRAM API to allow both executables and standard input, output, and error streams to be named by URLs, with GASS mechanisms used to fetch a URL-named executable into the cache, to fetch standard input, and to redirect standard output and error. The changes required to provide this support requires with just a few additional GASS calls.

4 GASS Implementation

GASS is implemented as a layered set of APIs, as follows. While typical application programmers will not need to use the latter three APIs, they are available for situations in which specialized data movement and access strategies are required.

- *File Access API.* This high-level file descriptor- and file stream-based interface to remote data accesses was discussed above. Caching decisions are made automatically based on file open parameters; URLs accessed by this interface exist in the cache until a process terminates. The file access API is implemented in terms of the cache and client APIs described below. Operations using this API are synchronous.

- *Cache Management API.* This low-level interface provides functions for manipulation of local caches; it permits the user to control directly insertion, locking, removal, and reference counting. Operations using this API are synchronous, but structured so as to allow overlapping of multiple cache operations.

- *Client Implementation API.* This low-level interface to remote data streams allows applications to eliminate data copies (by providing their own buffers to the communication system), to select transfer unit sizes, and to use proxy servers (e.g., for executable caching). Operations using this client API are primarily asynchronous, enabling data transfer operations to be overlapped with local I/O operations to increase performance of GASS operations.

- *Server Implementation API.* This low-level interface is used to implement GASS remote file servers. It implements the server side of the GASS remote file access protocol, making calls into application-specific server code to implement the data service. The user has control over a similar set of parameters to the client API. Both the client and the server implementations share a common data queuing model. Operations using this API are primarily asynchronous.

4.1 Cache Management API

The GASS cache API provides calls to add and delete files from a cache, to enquire about cache state, and to clean up cache contents. As described above, the contents of a cache are stored in a local file system and are owned by the user whose application submitted the read or write request. For each cache, a directory is maintained that contains information about the files stored in the cache, including the GASS file name (i.e., URL), local file name, file time stamp, and tag list. In the following, we explain some subtleties that arise in the design and implementation of these components.

GASS uses a modified reference counting scheme to determine when cache entries can be deleted. Each entry has associated with it not a simple reference count but rather a *tag list*. Each time a file is added to the cache, a user-specified tag string is added to the corresponding list. Delete operations also specify tags: if an entry's tag list is empty, the file can be purged, otherwise, the file remains in the cache. Functions are provided that remove all instances of a tag from the tag list. These functions make it possible to "clean up" a cache when an application component fails without clearing its reference counts: by clearing a application-specific tag from the tag list, one can clean up after a failed application without purging files that are still in use by other applications. The Globus GRAM service has been extended to support this mode of use: files accessed by a computation are tagged with the computation name, and when an application component terminates a "cleanup" operation is performed for that tag.

Cache directory entries also support a cache timestamp whose value is defined by the user rather than by the cache management system. The time stamp can therefore be used for a variety of purposes such as recording the modify date of the remote file, the time of the last add or delete operation, or some other application-specific interpretation.

The GASS implementation needs to guard against race conditions when several processes open the same file. For example, if two processes attempt to open the same file for reading, then the first "open" call should initiate a file transfer operation and return when this is done, while the second "open" call should block until the file transfer initiated by the first call completes. The GASS cache API introduces the concept of locks to simplify the implementation of these behaviors. The function used to add an entry to the directory also associates a lock with that entry. Any subsequent operation on that entry then blocks until the lock is cleared by a separate function call.

The cache management implementation uses a locking protocol to permit multiple threads or processes to insert and remove cache entries concurrently. However, it does *not* provide any wide area cache coherency; hence, for example, changes to a remote "original" file are not propagated to cached copies. However, cache coherency mechanisms can be layered on top of the GASS abstractions if required.

4.2 Client and Server Implementation APIs

GASS was designed to interface to a wide range of data management services: as illustrated in Figure 2, targets include specialized GASS servers, ordinary FTP and HTTP servers, and high-performance storage systems such as DPSS and HPSS. (To date, the first two of these have been implemented.)

The GASS client and server APIs are designed to simplify the task of providing remote access to new data management services, such as HPSS or services accessed via SRB [15]. The operations provided by these APIs are used to establish network connections to data servers, to issue read and write requests to remote servers, and to manage the movement of data between the client and server.

One function of the client and server APIs is to provide an efficient transport mechanism for moving file data between client and server. The transport layer provides a connection-oriented data stream, moving data to and from user allocated buffers, dynamically allocated buffers, or directly from local Unix file descriptors. In many situations, the availability of alternative data sources and sinks allows a GASS client or server implementation to eliminate the need to copy the same data multiple times. For example, an application-specific GASS client that wants to transfer a remote file into local memory rather than to a local file (as the file access API does) can accomplish this efficiently by sending data from a stream and receiving data into a user provided buffer.

Data transfer APIs are designed to be used asynchronously and are thread-safe. Communication can be performed by either a stream-based protocol implemented via Unix file descriptors or via a custom data transfer protocol based on the Nexus multimethod communication library [6]. In many situations, Nexus-based protocols can deliver superior performance to IP-based wide-area networking protocols.

5 Applications

We use three examples to show how GASS is used in applications.

5.1 The Globus Executable Management System

Because wide area computing environments can allocate computational resources dynamically, with the identity and type of resource being determined only at runtime, executable staging (i.e., automatically copying an executable to a machine prior to running that executable) becomes an important requirement in a grid programming system. GASS can be used to address this requirement as follows. We allow an executable to be named by a URL and augment the Globus resource management system [3] to use GASS client and caching API calls to transfer the executable from a remote executable repository to the machine in question. The GASS server providing the executable can be a specialized executable server, or more commonly is provided by `globusrun` (see below), a generic program startup tool provided as part of Globus. By performing simple variable substitutions on executable URLs, sophisticated executable management strategies can be implemented, for example, providing access to architecture-specific paths.

5.2 GASS Command Line Tools, `globus-rcp`, and `globusrun`

The Globus toolkit includes a small set of command line tools that provide access to basic GASS functionality. These tools include a basic GASS server; commands to "get" and "put" remote files; and a tool that allows one to examine and manipulate the contents of a GASS cache. These tools allow the programmer to implement pre-staging, post-staging, and other remote file operations without modifying user applications.

These command line tools have been used to implement a `globus-rcp` which, like the Unix `rcp` command, transfers files between two machines on the network, including third-party-initiated data transfer. By combining GASS functionality with Globus mechanisms for remote process startup and process-to-process authentication, this useful tool could be constructed as a shell script.

A second interesting Globus utility that makes use of GASS is `globusrun`, a program that automates many of the tasks associated with performing remote computation over potentially multiple computers. `globusrun` uses GASS for executable staging and for redirection of standard output and error streams to the terminal from which a program is invoked. Both functions are implemented in `globusrun` by using the GASS server API to incorporate "GASS server" functionality into the `globusrun` client. The "job manager" started by Globus at a remote site then uses the GASS client API to request the executable from the GASS server and the GASS cache API to create a unique temporary file name for the executable. Using the cache avoids unnecessary downloads when multiple processes share a filesystem. Note that the same GASS server API calls used to embed GASS server functionality into `globusrun` could just has easily been used to provide file service functions in an arbitrary application program.

Output redirection is achieved by using the GASS cache to manage a set of output files, one for each process created by the program. This use of the cache provides scalability beyond the number of file descriptors available to a UNIX process and provides line-buffered output. For systems which have a scheduler which starts the processes on the compute nodes, this overhead is minimal; for machines (such as SMPs) which do not have a scheduler but rely on the job manager to fork the process, the startup time is roughly linear, with a majority of the time being spent in cache operations to manage the standard output and error streams.

The `globusrun` utility is itself used to implement a variety of tools and applications. For example, MPICH-G, a wide-area implementation of the Message Passing Interface (MPI) [9], uses `globusrun` to initiate MPI programs on remote computers.

5.3 SF-Express: Distributed Supercomputing

SF-Express, a large-scale, distributed interactive simulation (DIS) application is typical of the type of application that can benefit from GASS services. SF-Express harnesses multiple supercomputers to meet the computational demands of large-scale network-based simulation environments [14]. A large simulation may involve many tens of thousands of entities and require thousands of processors. Globus services can be used to locate, assemble, and manage those resources. For example, in March 1998 SF-Express was run on 1386 processors distributed over 13 supercomputers at nine sites [2]. This 100,000-entity simulation set a new world record and met a performance goal originally scheduled for 2002.

An SF-Express run requires access to various types of data in order to initialize, including the terrain on which the simulation is to take place, descriptions of the entities that are to take part of the simulation, and the details of the specific simulation to be performed. While some data may change from simulation to simulation (scenarios, for example), much of it is constant between simulations, for example terrain, and all of the data is constant within a simulation run. Hence, SF-Express input patterns fit the GASS model and in fact when SF-Express was modified to use GASS services, the savings in setup time and complexity were significant.

SF-Express also uses GASS for two other purposes. On some machines, the executable management facility described above is used to stage simulation code, and on all machines append-mode file access is used to redirect diagnostic data streams back to the initiating computer. The latter use of GASS proved to be a tremendous advantage. In one early SF-Express experiment, prior to the use of Globus, logistical obstacles meant that it took *one month* to retrieve just a *subset* of the SF-Express log files from the various sites where a large run was performed. With GASS, these log files are available immediately and can be used to follow the progress of a simulation.

6 Performance Studies

We perform several experiments to determine the costs associated with GASS. Each experiment is conducted in two distinct environments: with the cache on local disk, as would be typical on a supercomputer with large, local, scratch storage; and with the cache on an NFS-mounted disk, as would be typical on a workstation or distributed memory cluster. For the former, we use an SGI Origin 2000 running Irix 6.5, using the /tmp filesystem on a SCSI disk with SGI's XFS filesystem. For the latter, we use an NFS filesystem between two IBM RS/6000s running AIX 4.2.1. In all tests, the remote data is located on an SGI Onyx2 with local SCSI disk and the XFS filesystem, located 40 km distant and connected via an OC3 network with a few intermediate routers.

6.1 GASS Cache Overheads

Our first experiments are designed to measure the costs inherent in the use of the GASS cache. We are interested in two such costs: the overhead incurred because data is read to disk and then read into memory, rather than being fetched into memory directly from the remote location; and costs associated with the locking and cache index file operations performed by the GASS cache.

We compare the wallclock running time of three programs. The first, `To Memory`, uses the GASS client API to transfer data directly into the memory of an application process. The second, *GASS Cache*, calls `globus_gass_open` using an x-gass URL to stage remote data into the cache, reads the file into memory, and closes the file by calling `globus_gass_close` to remove the cached copy. GASS cache overhead is incurred in both GASS calls. The third program, *No Cache*, uses the GASS client API with the x-gass URL to stage remote data into a local file (not in the GASS cache), opens the file, reads the file info memory, closes the file, and removes the local file. In effect, we measure the overhead of using the GASS cache versus manually staging files prior to opening them.

Results are shown in Table 1. A comparison of the *To Memory* and *GASS Cache* numbers shows that for large (100 MB) files, there is a significant advantage to bypassing the cache and reading directly into memory, but that for smaller files, the use of the cache does not incur a significant cost. A comparison of the `GASS Cache` and *No Cache* lines shows that GASS cache operations cost approximately 0.05 seconds to local disk and approximately 0.6 seconds to NFS disk, for small files. Overhead increases somewhat for larger files, which we do not yet understand, since the GASS cache operations are the same regardless of the file size.

6.2 GASS Cache Contention

As noted above, the GASS file access API can avoid repeated fetches of a remote file that is read by multiple processes on the same system. We performed three sets of experiments to determine how effectively the GASS cache operates in such scenarios. In each case, a parallel MPI program is started on an Origin 2000, with each node opening the same file and reading it into memory. We measure the wallclock time required for all nodes to read the entire file into memory. In all cases, local disk access is to the /tmp filesystem. Experiments are repeated for different file sizes and for different numbers of readers.

In the first test, all nodes open and read the same local file, not using any GASS mechanisms. This gives us the performance baseline reported as the first set of numbers in Figure 2. Notice that for large files, contention for the disk itself causes nearly linear scaling in runtime.

The program is then modified by replacing the `open` call with `globus_gass_open` and replacing the /tmp filename with an x-gass URL. Therefore, the first node to open the file will lock the GASS cache, transfer the file into the cache, and then allow other nodes to proceed with their opens. The GASS cache is located on the same /tmp filesystem as in the baseline test. Results are presented as the second set of numbers in Figure 2. We see that for small files, contention for the GASS cache index dominates the cost. However, for large files, the cost is dominated by the local disk I/O plus the transfer time.

For a final test, we run exactly the same program as the second test, except that prior to running the program we prestage the file into the GASS cache. This effectively removes the data transfer time from the previous numbers. These results are presented as the third set of numbers in Table 2. As expected, we see that for small files the cost is dominated by the GASS cache operations, whereas for large files the cost is dominated by the local disk I/O.

As local disk I/O speeds and transfer speeds increase, the GASS cache costs become relatively more expensive. Currently all cache operations require locking and accessing a single index file. Clearly more work is needed to remove contention for this index file.

The GASS client library supports data transfer with both GASS servers and standard ftp servers. To provide some insight into how these two mechanisms compare, we measured the average time in seconds to transfer a 10 MB file from the remote Onyx2 /tmp to the Origin's /tmp, using three methods. SGI's standard ftp client and ftpd server took 4.0 seconds; our GASS ftp client with SGI's ftpd server took 3.6 seconds, and our GASS client and server (which use our own protocols based on Nexus) took 4.7 seconds.

We see that the GASS ftp client achieves slightly better bandwidth than the native SGI ftp client, probably due to better use of socket and/or disk I/O. Bandwidth achieved using the GASS protocols is somewhat slower than from ftp protocols with the GASS client. However, this is expected, as Nexus currently performs extra copies of the data; we

Table 1: Time to transfer remote files of various sizes directly into memory (To Memory), through a /tmp file (No Cache), and through the GASS cache on /tmp (GASS Cache). All times are in seconds and are the average of multiple runs. See text for details.

	1 KB	10 KB	100 KB	1 MB	10 MB	100 MB
Local disk:						
To Memory	0.0198	0.043	0.118	0.549	4.533	10.201
No Cache	0.0418	0.151	0.188	0.622	4.794	46.853
GASS Cache	0.1306	0.203	0.233	0.920	5.161	50.148
NFS disk:						
To Memory	0.0297	0.206	0.223	1.162	5.58	16.35
No Cache	0.0976	0.385	0.298	1.521	10.86	107.80
GASS Cache	0.5165	0.879	0.920	2.109	12.88	122.19

Table 2: Results of contention experiments in which multiple processes open and read a file at the same time, via standard Unix open and close calls; GASS transfer followed by read; and GASS access to a prestaged file. All times are in seconds. See text for details.

Nodes	1 KB	10 KB	100 KB	1 MB	10 MB	100 MB
File read from /tmp without GASS mechanisms						
1	1.052	1.053	1.061	1.094	1.489	5.63969
2	1.061	1.062	1.074	1.174	2.207	14.8603
4	1.190	1.202	1.575	1.371	3.761	42.5152
8	1.910	1.751	1.735	2.011	7.361	108.068
16	2.710	3.009	3.105	3.710	18.228	226.103
32	3.694	3.825	3.913	4.725	33.982	489.276
File transferred from remote URL with GASS mechanisms						
1	1.12	1.18	1.24	2.26	5.57	54.94
2	1.71	1.72	1.89	3.05	7.86	67.07
4	2.66	2.74	2.87	5.43	8.08	80.72
8	3.85	4.44	4.90	7.55	10.19	156.40
16	6.72	7.40	8.95	9.61	19.56	280.73
32	9.69	11.3	12.57	13.76	37.81	522.77
Prestaged file read from /tmp with GASS mechanisms						
1	1.199	1.089	1.129	1.181	1.613	5.933
2	1.602	1.566	2.249	1.643	2.037	16.263
4	2.584	2.582	2.609	2.172	3.619	47.242
8	4.615	5.236	4.111	5.193	8.479	107.819
16	5.676	6.694	8.647	7.276	14.707	227.843
32	9.358	8.920	9.013	9.822	35.451	486.753

expect the GASS protocol's performance to improve when these copies are eliminated in a future version of Globus.

6.3 GASS and AFS Performance

Although GASS and AFS are designed with different goals in mind, it is instructive to compare their performance. Our comparision uses two versions of the parallel program used above for GASS performance evaluation. In a first version, each process accesses the same remote file using GASS mechanisms, while in the second version, AFS mechanisms are used. In both cases, the parallel program is run on an SGI Origin 2000 (AFS client with a cache size of 150 MB, parameters set according to the MEDIUM default setup, located at NCSA in Urbana), while the servers (GASS or AFS) were similar Sun UltraSparcs, located on the same site (at Argonne National Laboratory in Chicago) and on similar networks, behind two identical routers. The AFS cache is carefully flushed before each measurement. (As GASS automatically cleans its cache when a file is not referenced anymore, no flush is necessary in the GASS case.)

We see that while AFS performs slightly better than GASS for small files (probably because of the known inefficiencies in the GASS file-based cache locking routines), for large files GASS is significantly faster. It would appear that, as asserted earlier, there are significant performance advantages to using GASS rather than AFS in wide area environments.

7 Conclusions

We have described a data access and movement service for high-performance distributed computing environments. This service implements a set of strategies designed to make efficient use of network bandwidth, while allowing the use of standard mechanisms for data access and URLs to provide

Table 3: Overall time required to read the content of a remote file using GASS and AFS to access the file. All times are in seconds. See text for details.

Nodes	1 KB	10 KB	100 KB	1 MB	10 MB
File transferred from remote URL with GASS mechanisms					
1	1.351	1.497	1.572	1.803	6.520
2	1.423	1.633	1.743	1.952	7.112
4	1.713	1.936	1.808	2.173	8.490
8	1.843	1.871	1.956	2.540	15.27
16	3.334	4.061	4.485	3.785	25.37
32	9.023	10.25	10.93	10.85	49.11
File accessed using AFS					
1	1.126	1.168	1.191	1.873	7.218
2	1.132	1.236	1.635	2.531	10.98
4	1.638	1.724	2.129	2.942	15.94
8	2.047	2.101	2.596	3.733	26.51
16	2.429	2.641	2.834	6.894	46.98
32	5.622	7.787	6.756	16.56	97.16

a uniform file namespace. Data access and movement operations are separated, hence allowing programmers to implement application-specific data movement strategies without modifying applications.

While conceptually simple, this GASS service has proved to be remarkably useful. We have found that a wide variety of higher-level services and application constructs can be implemented with ease in terms of GASS mechanisms. In addition, performance experiments show that we can deliver good data movement performance to applications. Hence, we believe that GASS represents an interesting approach to data management in grid environments.

In future work, we plan to expand the set of storage systems that are supported by GASS, in order to allow its use for data management in high-performance environments. For example, interfaces to the High Performance Storage System (HPSS) and Distributed Parallel Storage System (DPSS) will allow us to use GASS to move data from a remote hierarchical storage system (HPSS) to a locally distributed disk cache (DPSS). The Storage Resource Broker (SRB) interface is another approach to dealing with heterogeneous storage systems; we hope to explore interfacing SRB into GASS. We are also planning to investigate the integration of GASS data movement mechanisms with advance reservation capabilities currently being prototyped in Globus.

Acknowledgments

We gratefully acknowledge helpful discussions with Karl Czajkowski, Steven Fitzgerald, Nicholas Karonis, and Brian Toonen, and the assistance of Noam Freedman with Tardis performance studies. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Science Foundation, and by the ASCI Flash Center at the University of Chicago under DOE contract B341495.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The UFO global file system. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, January 1997.

[2] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, pages 29–42. IEEE Computer Society Press, 1998.

[3] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

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

[5] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[7] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proc. IOPADS'97*, pages 14–25. ACM Press, 1997.

[8] A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[10] James V. Huber, Jr, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[11] Steven R. Kleiman. Vnodes: an architecture for multiple file system types in Sun Unix. In *Proc. USENIX Summer Conference.*, pages 238–247. The USENIX Association, 1986.

[12] D. Kotz and C. Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[14] P. Messina, S. Brunett, D. Davis, T. Gottschalk, D. Curkendall, L. Ekroot, and H. Siegel. Distributed interactive simulation for synthetic forces. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[15] Reagan Moore, Chaitanya Baru, Richard Marciano, Arcot Rajasekar, and Michael Wan. Data-intensive computing. In *[5]*, pages 105–129.

[16] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, 1986.

[17] B. C. Neuman. The Prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, Fall 1992.

[18] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 41–55, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[19] B. Tierney, W. Johnston, L. Chen, H. Herzog, G. Hoo, G. Jin, and J. Lee. Distributed parallel data storage systems: A scalable approach to high speed image servers. In *Proc. ACM Multimedia 94*. ACM Press, 1994.

[20] A. Vahdat, P. Eastham, and T. Anderson. WebFS: A global cache coherent filesystem. Technical report, Department of Computer Science, UC Berkeley, 1996.