Data Management and Transfer in High-Performance Computational Grid Environments

Bill Allcock[1] Joe Bester[1] John Bresnahan[1] Ann L. Chervenak[2] Ian Foster[1,3]
Carl Kesselman[2] Sam Meder[1] Veronika Nefedova[1] Darcy Quesnel[1] Steven Tuecke[1]

[1] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{allcock, bester, foster, nefedova, quesnel, tuecke}@mcs.anl.gov

[2] Information Sciences Institute
University of Southern California
Los Angeles, CA 90292
{annc, carl}@isi.edu

[3] Department of Computer Science & The Computation Institute
The University of Chicago
Chicago, IL 60637

Abstract

An emerging class of data-intensive applications involve the geographically dispersed extraction of complex scientific information from very large collections of measured or computed data. Such applications arise, for example, in experimental physics, where the data in question is generated by accelerators, and in simulation science, where the data is generated by supercomputers. So-called Data Grids provide essential infrastructure for such applications, much as the Internet provides essential services for applications such as e-mail and the Web. We describe here two services that we believe are fundamental to any Data Grid: reliable, high-speed transport and replica management. Our high-speed transport service, GridFTP, extends the popular FTP protocol with new features required for Data Grid applications, such as striping and partial file access. Our replica management service integrates a replica catalog with GridFTP transfers to provide for the creation, registration, location, and management of dataset replicas. We present the design of both services and also preliminary performance results. Our implementations exploit security and other services provided by the Globus Toolkit.

1 Introduction

Data-intensive, high-performance computing applications require the efficient management and transfer of terabytes or petabytes of information in wide-area, distributed computing environments. Examples of such applications include experimental analyses and simulations in scientific disciplines such as high-energy physics, climate modeling, earthquake engineering, and astronomy. In such applications, massive datasets must be shared by a community of hundreds or thousands of researchers distributed worldwide. These researchers need to be able to transfer large subsets of these datasets to local sites or other remote resources for processing. They may create local copies or replicas to overcome long wide-area data transfer latencies. The data management environment must provide security services such as authentication of users and control over who is allowed to access the data. In addition, once multiple copies of files are distributed at multiple locations, researchers need to be able to locate copies and determine whether to access an existing copy or create a new one to meet the performance needs of their applications.

We have argued elsewhere [1] that the requirements of such distributed data intensive applications are best met by the creation of a Data Grid infrastructure that provides a set of orthogonal, application-independent services that can then be combined and specialized in different ways to meet the needs of specific applications. We have argued further that this Data Grid infrastructure can usefully build on capabilities provided by the emerging Grid [2], such as resource access, resource discovery, and authentication services. Our Globus Toolkit [3] provides a widely used instantiation of the lower layers of this Grid architecture.

In this paper, we focus our attention on what we view as two fundamental Data Grid services, namely, *secure, reliable, efficient data transfer* and the ability to *register, locate, and manage multiple copies* of datasets. We describe the design, prototype implementation, and preliminary performance evaluation of our realization of these two services within the context of the Globus Toolkit. Given these two services, a wide range of higher-level data management services can be constructed, including reliable creation of a copy of a large data collection at a new location; selection of the best replica for a data transfer based on performance estimates provide by information services; and automatic creation of new replicas in response to application demands. However, we do not directly address these issues here.

2 Data-Intensive Computing Requirements

We use two application examples to motivate the design of our Data Grid services: high-energy physics experiments and climate modeling. We characterize each with respect to parameters such as average file sizes, total data volume, rate of data creation, types of file access (write-once, write-many), expected access rates, type of storage system (file system or database), and consistency requirements for multiple copies of data. In both these applications, as well as others that we have examined, such as earthquake engineering and astronomy, we see a common requirement for two basic data management services: efficient data transfer and access to large files and a mechanism for creating and managing multiple copies of files.

2.1 High-Energy Physics Applications

Experimental physics applications operate on and generate large amounts of data. For example, beginning in 2005, the Large Hadron Collider (LHC) at the European physics center CERN will produce several petabytes of raw and derived data per year for approximately 15 years. The data generated by physics experiments is of two types: *experimental data*, or information collected *by* the experiment; and *metadata*, or information *about* the experiment, such as the number of events or the results of analysis.

File sizes and numbers of files are determined to some extent by the type of software used to store experimental data and metadata. For example, several experiments have chosen to use the object-oriented Objectivity database. Current file sizes (e.g., within the BaBar experiment) range from 2 to 10 gigabytes in size, while metadata files are approximately 2 gigabytes. Objectivity currently limits database federations to 64K files. However, future versions of Objectivity will support more files, allowing average file sizes to be reduced.

Access patterns vary for experimental data files and metadata. Experimental data files typically have a single creator. During an initial production period lasting several weeks, these files are modified as new objects are added. After data production is complete, files are not modified. In contrast, metadata files may be created by multiple individuals and may be modified or augmented over time, even after the initial period of data production. For example, some experiments continue to modify metadata files to reflect the increasing number of total events in the database. The volume of metadata is typically smaller than that of experimental data.

The consumers of experimental physics data and metadata will number in the hundreds or thousands. These users are distributed at many sites worldwide. Hence, it is often desirable to make copies or *replicas* of the data being analyzed to minimize access time and network load. For example, Figure 1 shows the expected replication scheme for LHC physics datasets. Files are replicated in a hierarchical manner, with all files stored at a central location (CERN) and decreasing subsets of the data stored at national and regional data centers [4][5].

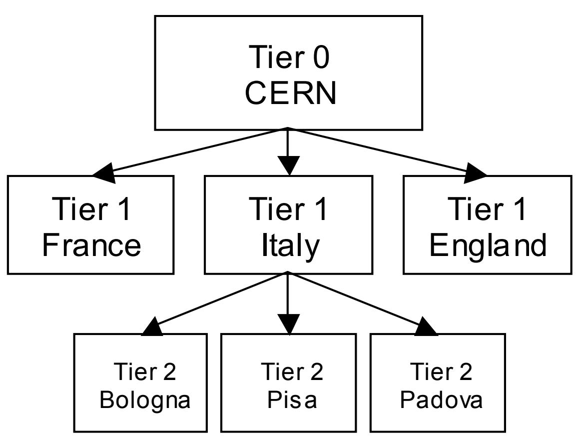

Figure 1: Scheme for hierarchical replication of Physics data

Replication of physics datasets is complicated by several factors. First, security services are required to authenticate the user and control access to storage systems. Next, because datasets are so large, it may be desirable to replicate only "interesting" subsets of the data. Finally, replication of data subject to modification implies a need for a mechanism for propagating updates to all replicas. For example, consider the initial period of data production, during which files are modified for several weeks. During this period, users want their local replicas to be updated periodically to reflect the experimental data being produced. Typically, updates are batched and performed every few days. Since metadata updates take place over an indefinite period, these changes must also be propagated periodically to all replicas.

In Table 1, we summarize the characteristics of high-energy physics applications.

Table 1: Characteristics of high-energy physics applications

Rate of data generation (starting 2005)	Several petabytes per year
Typical experimental database file sizes	2 to 10 gigabytes
Typical metadata database file sizes	2 gigabytes
Maximum number of database files in federation	Currently 64K; eventually millions
Period of updates to experimental data	Several weeks
Period of updates to metadata	Indefinite
Type of storage system	Object-oriented database
Number of data consumers	Hundreds to thousands

2.2 Climate Modeling Application

Climate modeling requires lengthy simulations and produces a large volume of model output. The reason for this is that the global climate-change "signal" or "signature" must be discernable above the "noise" of the natural variability of the climate system. Variability in the atmosphere has fairly short periods: weeks to months. However, fluctuations in ocean climate have occurred on time-scales of tens to hundreds of years; these are generally attributed to shifts in the ocean circulation, especially the Gulf Stream. The ocean also has modes of variability with periods extending to hundreds or thousands of years. As a result, very long-duration computations accompanied by frequent output of very large files are needed to analyze the simulated climate variability, which must then be compared with what is known about the observed variability.

Climate modeling research groups generate large (multi-terabyte) *reference simulations* at supercomputer centers. Typical simulation runs span 100 years. For PCM models, the output of simulations consists of four types: atmospheric monthly average data, atmospheric daily average data, ocean monthly average data, and sea ice monthly average data. Table 2 summarizes the characteristics of the climate model application.

Reference simulation data are typically released in stages to progressively larger communities: first the research collaboration that generated the data, then perhaps selected colleagues, and eventually the entire community. To determine which users are allowed to view the collection at each stage, these applications require access control. Reference simulation data are typically stored in a file system, often using a structured data format such as NetCDF with associated metadata. Files are not updated once released. However, as in the physics application, climate modeling researchers find it convenient to create local copies of portions of the data. Therefore, the application has similar needs for managing copies of datasets at multiple locations, as well as for higher-level services such as replica selection or automatic replica creation.

Table 2: Characteristics of climate modeling applications

Current total volume (PCM data)	5.72 terabytes
Total data files	Approx. 23,600
Expected additional volume, next 3 to 5 years	5 to 8 terabytes
Typical length of climate simulation model run	100 years
Total data generated by 100-year simulation run	132 gigabytes
Average data per year of simulation model	1.3 gigabytes
Atmospheric data monthly averages per year of simulation model	322 megabytes
Atmospheric data daily averages per year of simulation model	286 megabytes
Ocean model data monthly averages per year of simulation model	706 megabytes
Sea ice data monthly averages per year of simulation model	9 megabytes
Files generated for atmospheric monthly average data	1 per model year
Files generated for atmospheric daily average data	1 per 300 model days
Files generated for ocean monthly average data	1 per model year
Files generated for sea ice monthly average data	1 per model decade

3 The Globus Toolkit

The term *Grid computing* refers to the emerging computational and networking infrastructure that is designed to provide pervasive, uniform and reliable access to data, computational, and human resources distributed over wide area environments [6]. Grid services allow scientists at locations throughout the world to share data collection instruments such as particle colliders, compute resources such as supercomputers and clusters of workstations, and community datasets stored on network caches and hierarchical storage systems.

The Globus Toolkit developed within the Globus project provides middleware services for Grid computing environments. Major components include the Grid Security Infrastructure (GSI), which provides public-key-based authentication and authorization services; resource management services, which provide a language for specifying application requirements, mechanisms for immediate and advance reservations of Grid resources, and for remote job management; and information services, which provide for the distributed publication and retrieval of information about Grid resources.

Data Grid services complement and build on these components. For example, the GridFTP transfer service and the replica management service described in the rest of this paper use GSI for authentication and authorization. Higher-level data replication services can use the information service to locate the "best" replica and the resource management service to reserve the computational, network, and storage resources required by a data movement operation.

4 GridFTP: A Secure, Efficient Data Transport Mechanism

The applications that we consider use a variety of storage systems, each designed to satisfy specific needs and requirements for storing, transferring and accessing large datasets. These include the Distributed Parallel Storage System (DPSS) and the High Performance Storage System (HPSS), which provide high-performance access to data and utilize parallel data transfer and/or striping across multiple servers to improve performance [7][8], and the Storage Resource Broker (SRB), which connects heterogeneous data collections, provides a uniform client interface to storage repositories, and provides a metadata catalog for describing and locating data within the storage system [9].

Unfortunately, these storage systems typically use incompatible and often unpublished protocols for accessing data, and therefore each requires the use of its own client. These incompatible protocols and client libraries effectively partition the datasets available on the Grid. Applications that require access to data stored in different storage systems must use multiple access methods.

One approach to breaking down partitions created by these mutually incompatible storage system protocols is to build a layered client or gateway that can present the user with one interface, but that translates requests into the various storage system protocols and/or client library calls. This approach is attractive to existing storage system providers because it does not require them adopt support for a new protocol. But it also has significant disadvantages, including:

- **Performance:** Costly translations are often required between the layered client and storage system specific client libraries and protocols. In addition, it can be challenging to transfer a dataset efficiently from one storage system to another.
- **Complexity:** Building and maintaining a client or gateway that supports numerous storage systems requires considerable work. In addition, staying up to date as each storage system independently evolves is very difficult. This is further exacerbated by the need to support multiple client languages, such as C/C++, Java, Perl, Python, shells, etc. It can be challenging to transfer a dataset from one system to another.

We argue that it would be mutually advantageous to both storage providers and users to have a common level of interoperability between all of these disparate systems: a common—but extensible—underlying data transfer protocol. A common data transfer protocol for all of these customized storage systems would confer benefits to both the keepers of large datasets and the users of these datasets. Dataset storage providers would gain a broader user base, because their data would be available to any client. Dataset storage users would gain access to a broader range of storage systems and data. In addition, these benefits can be gained without the performance and complexity problems of the layered client or gateway approach.

Furthermore, establishing a common data transfer protocol would eliminate the current duplication of effort in developing unique data transfer capabilities for different storage systems. A pooling of effort in the data transfer protocol area would lead to greater reliability, performance, and overall features that would then be available to all distributed storage systems.

As a candidate for a common, extensible data transfer protocol, we propose the GridFTP data transfer protocol, which extends the standard FTP protocol to include a superset of the features offered by the various Grid storage systems currently in use. We chose to extend the FTP protocol (rather than, for example, WebDAV) because we observed that FTP is the protocol most commonly used for data transfer on the Internet and the most likely candidate for meeting the Grid's needs. FTP is a widely implemented and well-understood IETF standard protocol with a large base of code and expertise from which to build. In addition, the FTP protocol provides a well-defined architecture for protocol extensions and supports dynamic discovery of the extensions supported by a particular implementation. Third, numerous groups have added extensions through the IETF, and some of these extensions are particularly useful in the Grid.

4.1 GridFTP Functionality

GridFTP functionality includes some features that are supported by FTP extensions that have already been standardized (RFC 959) but are seldom implemented in current systems. Other features are new extensions to FTP.

- **Grid Security Infrastructure and Kerberos support:** Robust and flexible authentication, integrity, and confidentiality features are critical when transferring or accessing files. GridFTP must support GSI and Kerberos authentication, with user controlled setting of various levels of data integrity and/or confidentiality. GridFTP implements the authentication mechanisms defined by RFC 2228, "FTP Security Extensions".
- **Third-party control of data transfer:** To manage large datasets for distributed communities, we must provide authenticated *third-party* control of data transfers between storage servers. A third-party operation allows a user or application at one site to initiate, monitor and control a data transfer operation between two other sites: the source and destination for the data transfer. Our implementation adds Generic Security Services (GSS)-API authentication to the existing third-party transfer capability defined in the FTP standard.
- **Parallel data transfer:** On wide-area links, using multiple TCP streams in parallel (even between the same source and destination) can improve aggregate bandwidth over using a single TCP stream [10]. GridFTP supports parallel data transfer through FTP command extensions and data channel extensions.

- **Striped data transfer:** Data may be striped or interleaved across multiple servers, as in a DPSS network disk cache [11]. GridFTP includes extensions that initiate striped transfers, which use multiple TCP streams to transfer data that is partitioned among multiple servers. Striped transfers provide further bandwidth improvements over those achieved with parallel transfers. We have defined GridFTP protocol extensions that support striped data transfers.
- **Partial file transfer:** Some applications can benefit from transferring portions of files rather than complete files: for example, high-energy physics analyses that require access to relatively small subsets of massive, object-oriented physics database files. The best that the standard FTP protocol allows is transfer of the remainder of a file starting at a particular offset. GridFTP provides commands to support transfers of arbitrary subsets or regions of a file.
- **Automatic negotiation of TCP buffer/window sizes:** Using optimal settings for TCP buffer/window sizes can dramatically improve data transfer performance. However, manually setting TCP buffer/window sizes is an error-prone process (particularly for non-experts) and is often simply not done. GridFTP extends the standard FTP command set and data channel protocol to support both manual setting and automatic negotiation of TCP buffer sizes for large files and for large sets of small files.
- **Support for reliable and restartable data transfer:** Reliable transfer is important for many applications that manage data. Fault recovery methods are needed to handle failures such as transient network and server outages. The FTP standard includes basic features for restarting failed transfers that are not widely implemented. GridFTP exploits these features and extends them to cover the new data channel protocol.

4.2 The GridFTP Protocol Implementation

Our implementation of the GridFTP protocol supports partial file transfers, third-party transfers, parallel transfers and striped transfers. We do not yet support automatic negotiation of TCP buffer/window sizes. The implementation consists of two principal C libraries: the globus_ftp_control_library and the globus_ftp_client_library.

The **globus_ftp_control_library** implements the control channel API. This API provides routines for managing a GridFTP connection, including authentication, creation of control and data channels, and reading and writing data over data channels. Having separate control and data channels, as defined in the FTP protocol standard, greatly facilitates the support of such features as parallel, striped and third-party data transfers. For parallel and striped transfers, the control channel is used to specify a put or get operation; concurrent data transfer occurs over multiple parallel TCP data channels. In a third-party transfer, the initiator monitors or aborts the operation via the control channel, while data transfer is performed over one or more data channels between source and destination sites.

The **globus_ftp_client_library** implements the GridFTP client API. This API provides higher-level client features on top of the globus_ftp_control library, including complete file get and put operations, calls to set the level of parallelism for parallel data transfers, partial file transfer operations, third-party transfers, and eventually, functions to set TCP buffer sizes.

4.3 GridFTP Performance

Preliminary performance measurements of our GridFTP prototype demonstrate that we can indeed obtain high performance and reliable transfers in wide area networks. Further improvements are expected as a result of tuning and improvements to the implementation.

Figure 2 shows the performance of GridFTP transfers between two workstations, one at Argonne National Laboratory in Illinois and the other at Lawrence Berkeley National Laboratory in California, connected over the ES-Net network (www.es.net). Both workstations run the Linux operating system and have RAID storage systems with read/write bandwidth of approximately 60 megabytes per second. Gigabit Ethernet is the slowest link in the network path. The bottom curve in the graph shows GridFTP performance as the number of simultaneous TCP streams increases. For comparison, the top curve in the graph shows the performance of the same number of TCP streams measured by iperf, a tool for evaluating network performance that performs no disk I/O and has minimal CPU or protocol overhead [12]. Iperf provides one measurement of the maximum throughput of the network. Our experiment was run in random order relative to the number of streams, with the GridFTP measurement for a certain number of streams followed immediately by the iperf measurement for the same number of streams. For example, we took GridFTP measurements followed by iperf measurements for 18 simultaneous streams, then we took the two measurements for five simultaneous streams, etc. This randomization prevents any system or network trends from biasing the results, but assures that iperf and GridFTP measurements for the same number of streams are run close together temporally to reflect possible interactions with the number of streams. Each data point on the graph represents a single measurement. Iperf measurements were made using a one megabyte window size and ran for 30 seconds. GridFTP measurements recorded the time to transfer a one gigabyte file.

The graph shows that GridFTP bandwidth increases with the number of parallel TCP streams between the two workstations, until bandwidth reaches about 200 megabits per second with seven to ten TCP streams. Differences between iperf and GridFTP performance can be attributed to overheads including authentication and protocol operations, reporting performance to the client, and checkpointing data to allow restart. GridFTP achieves on average approximately 78% of the iperf bandwidth, although there is a great deal of variability. We speculate that this variability is due to the requirement that GridFTP wait for any packets that are misrouted or dropped, while iperf simply runs for its allotted time and stops regardless of whether there are outstanding packets.

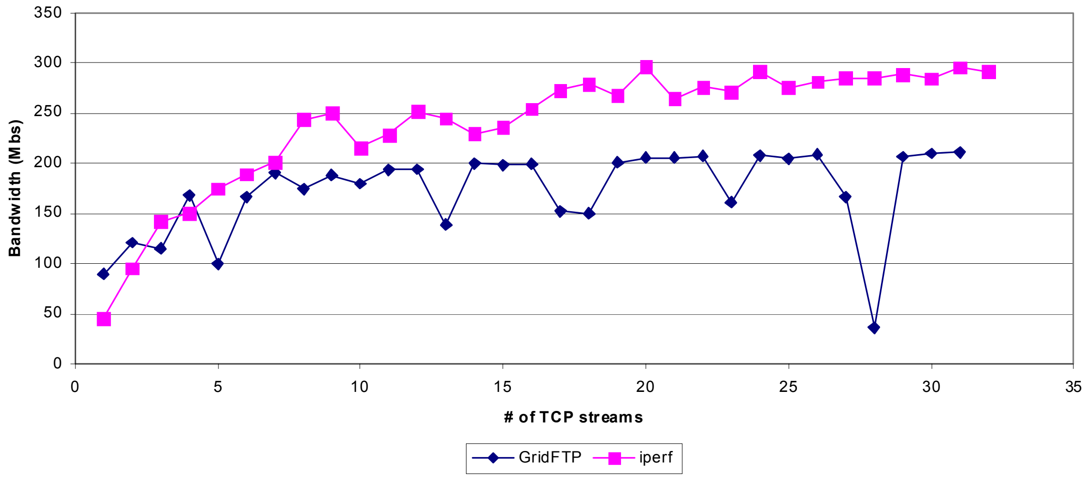

Figure 2: GridFTP performance compared to iperf measurements of network connection between Argonne National Laboratory and Lawrence Berkeley National Laboratory.

Figure 3 demonstrates GridFTP reliability. We show aggregate parallel bandwidth for a period of approximately fourteen hours during the SC'00 Conference in Dallas, Texas, on November 7, 2000. This data corresponds to parallel transfers between two uniprocessor hosts using varying levels of parallelism, up to a maximum of eight streams. The graph was produced with the NetLogger system [13]. Bandwidth between the two hosts reaches approximately 80 megabits per second, somewhat lower than shown for the hosts in Figure 2, most likely due to disk bandwidth limitations. Figure 3 shows drops in performance due to various network problems, including a power failure for the SC network (SCiNet), DNS problems, and backbone problems on the exhibition floor. Because the GridFTP protocol supports restart of failed transfers, the interrupted transfers are able to continue as soon as the network is restored. Toward the right side of the graph, we see several temporary increases in aggregate bandwidth, due to increased levels of parallelism. The frequent drop in bandwidth to relatively low levels occurs because our current implementation of GridFTP destroys and rebuilds its TCP connections between consecutive transfers. To address this problem, our next GridFTP implementation will support *data channel caching*. This mechanism allows a client to indicate that a TCP stream is likely to be re-used soon after the existing transfer is complete. In response to this hint, we will temporarily keep the TCP channel active and allow subsequent transfers to use the channel without requiring costly breakdown, restart, and re-authentication operations.

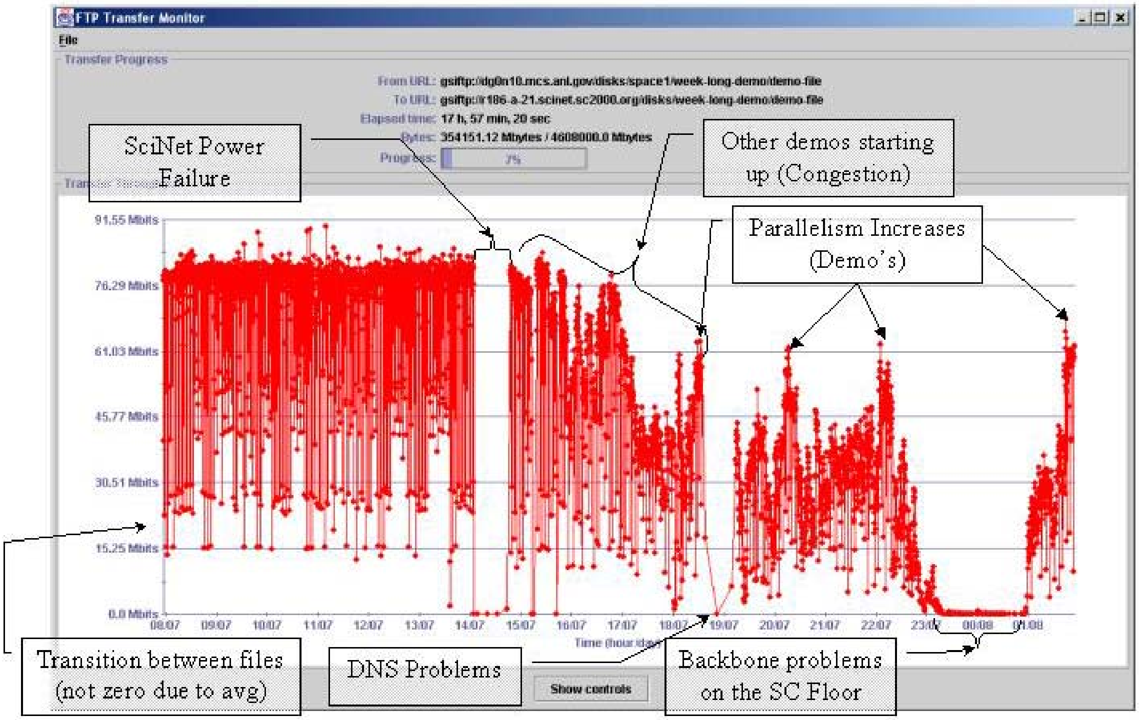

Figure 3: Bandwidth measured for a series of transfers performed over a 14 hour period, between Dallas and Chicago

Figure 4 and Table 2 address our achievable peak performance. These data were obtained during the Network Challenge competition at SC'00 in November 2000. Our configuration for this competition consisted of eight Linux workstations on the SC'00 exhibition floor in Dallas, Texas, sending data across the wide area network to eight workstations (four Linux, four Solaris) at Lawrence Berkeley National Laboratory in California. Figure 4 illustrates the configuration. We used *striped* transfers during this competition, with a 2-gigabyte file partitioned across the eight workstations on the exhibition floor. Each workstation actually had four copies of its file partition. On each server machine, a new transfer of a copy of the file partition was initiated after 25% of the previous transfer was complete. Each new transfer creates a new TCP stream. At any time, there are up to four simultaneous TCP streams transferring data from each server in the cluster of eight workstations, for a total of up to 32 simultaneous TCP streams. Our current lack of data channel caching means that there are often fewer than four simultaneous streams transferring data on each host.

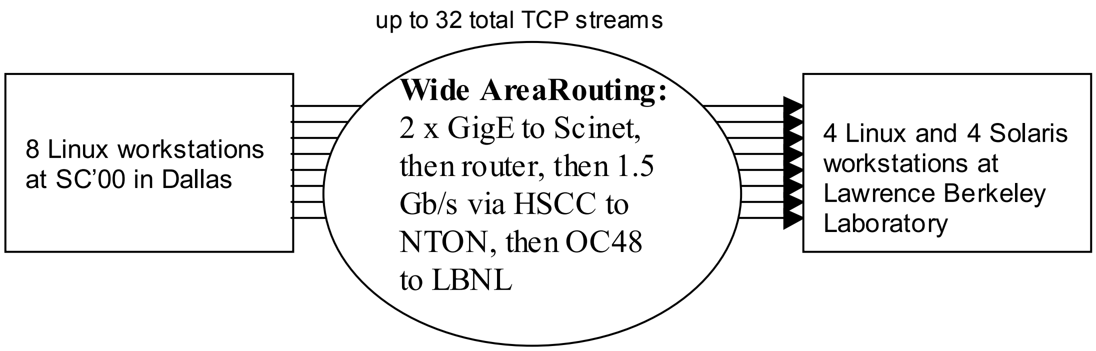

Figure 4: Experimental configuration for Network Challenge competition at SC'00.

Table 2 summarizes the results of our Network Challenge competition entry. We achieved a peak transfer rate of 1.55 gigabits/second over an interval of 0.1 seconds and of 1.03 gigabits/second over an interval of 5 seconds. Over the hour-long period of our competition entry, we sustained an average data rate of 512.9 megabits per second, corresponding to over a quarter of a terabyte transferred during that hour.

Table 2: Network Challenge configuration and performance results

Striped servers at source location	8
Striped servers at destination location	8
Maximum simultaneous TCP streams per server	4
Maximum simultaneous TCP streams overall	32
Peak transfer rate over 0.1 seconds	1.55 Gbits/sec
Peak transfer rate over 5 seconds	1.03 Gbits/sec
Sustained transfer rate over 1 hour	512.9 Mbits/sec
Total data transferred in 1 hour	230.8 Gbytes

5 Replica Management

We next describe our second fundamental Data Grid service, that is, replica management. This component is responsible for managing the replication of complete and partial copies of *datasets,* defined as collections of files. Replica management services include:

- creating new copies of a complete or partial collection of files
- registering these new copies in a *Replica Catalog*
- allowing users and applications to query the catalog to find all existing copies of a particular file or collection of files

The replica management service is just one component in a computational grid environment that provides support for high-performance, data-intensive applications. A recently proposed architecture for computational grids [2] includes four levels:

- **Fabric:** At the lowest level of the grid architecture are the basic components and resources from which a computational grid is constructed. These include storage systems, networks, and catalogs.
- **Connectivity:** At the next level of the architecture are services concerned with communication and authentication. Typically, these are standard protocols.
- **Resource:** Services at the next highest level are concerned with providing secure, remote access to individual resources.
- **Collective:** Services at the collective level support the coordinated management of multiple resources.

Figure 5 shows a partial list of components at each level of the proposed grid architecture, with particular emphasis on components related to replica management. At the lowest *fabric* level of the architecture are the basic components that make up the Grid, including storage systems, networks and computational systems. In addition, the picture includes two catalogs: a *metadata catalog* that contains descriptive information about files and a *replica catalog* where information is stored about registered replicas. At the *connectivity* layer are various standard protocols for communication and security. At the *resource* level are services associated with managing individual resources, for example, storage and catalog management protocols as well as protocols for network and computation resource management. Finally, at the *collective* layer of the architecture are higher-level services that manage multiple underlying resources, including the replica management service that is the focus of this paper. Other services at the collective layer include services for replica selection, metadata management, management of replicated and distributed catalogs, and for information services that provide resource discovery or performance estimation.

The replica management architecture assumes the following data model. Data are organized into *files*. For convenience, users group files into *collections*. A *replica* or *location* is a subset of a collection that is stored on a particular physical storage system. There may be multiple, possibly overlapping subsets of a collection stored on multiple storage systems in a data grid. These grid storage systems may use a variety of underlying storage technologies and data movement protocols, which are independent of replica management.

We distinguish between logical file names and physical file names. A logical file name is a globally unique identifier for a file within the data grid's namespace. The logical file name may or may not have meaning for a human, for example, by recording information about the contents of a file. However, the replica management service does not use any semantic information contained in logical file names. The purpose of the replica management service is to map a unique logical file name to a possibly different physical name for the file on a particular storage device.

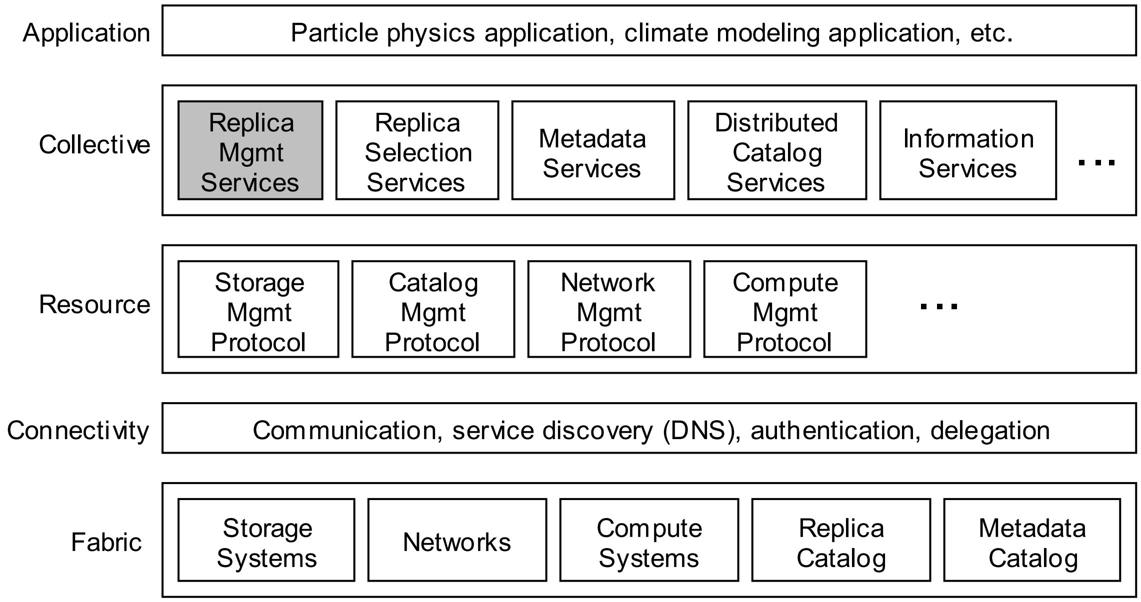

Figure 5: Shows a partial list of elements of the Data Grid Reference Architecture [2] that are relevant to replica management.

In the remainder of this section, we describe the replica management service in more detail. First, we explain key architectural design decisions for the service. Second, we present a scenario explaining how an application would use the replica management service in conjunction with other services in a data grid environment. Next, we list the key operations provided by the replica management service. Fourth, we describe the underlying Replica Catalog, which is used to store information about registered replicas. Next, we describe the API of the higher-level Replica Management Service and present some initial performance results for the replica management service. We conclude this section with a discussion of one important implementation issue: the amount of reliability and availability provided by a replica management service.

5.1 Key Architecture Decisions for the Replica Management Service

Next, we discuss several important design decisions for the replica management service. Our motivation for several of these decisions was to clearly define the role of the service and to limit its complexity.

Separation of Replication and Metadata Information

One key observation is that the objects that can be registered with the replica management service contain only the information required to map logical file and collection names to physical locations. Any other information that might be associated with files or collections, such as descriptions of file contents or the experimental conditions under which files were created, should be stored in an orthogonal *metadata* *management service*. Our architecture places no constraints on the design or the contents of the metadata service.

Typically, a user might first consult the metadata management service to select logical files based on metadata attributes such as the type of experimental results needed or the time when data were collected. Once the necessary logical files are identified, the user consults the replica management service to find one or more physical locations where copies of the desired logical files are stored.

Replication Semantics

The word *replica* has been used in a variety of contexts with a variety of meanings. At one extreme, the word replica is sometimes used to mean "a copy of a file that is guaranteed to be consistent with the original, despite updates to the latter." A replica management architecture that supports this definition of replication would be required to implement the full functionality of a wide area, distributed database, with locking of files during modification and atomic updates of all replicas.

Because of the difficulty of implementing such a distributed database, our architecture operates at the other extreme: our replica management service explicitly does not enforce any replica semantics. In other words, for multiple replicas (locations) of a logical collection, we make no guarantees about file consistency, nor do we maintain any information on which was the "original" or "source" location from which one or more copies were made. When users register files as replicas of a logical collection, they assert that these files are replicas under a user-specific definition of replication. Our replica management service does not perform any operations to check, guarantee or enforce the user's assertion.

Replica Management Service Consistency

Although our architecture makes no guarantees about consistency among registered file replicas, we must make certain guarantees about the consistency of information stored in the replica management service itself. Since computational and network failures are inevitable in distributed computing environments, the replica management service must be able to recover and return to a consistent state despite conflicting or failed operations.

One way our architecture remains consistent is to guarantee that no file registration operation should successfully complete unless the file exists completely on the corresponding storage system. Consider a replica copy operation that includes copying a file from a source to a destination storage system and registering the new file in a location entry in the replica service. We must enforce an ordering on operations, requiring that the copy operation successfully completes before registration of the file with the replica management service is allowed to complete.

If failures occur and the state of the replica management service is corrupted, we must *rollback* the replica management service to a consistent state.

Rollback

Certain operations on the replica management service are atomic. If they are completed, then the state of the replica management service is updated. If these operations fail, then the state of the replica management service is unchanged. Examples of atomic operations include adding a new entry to the replica management service, deleting an entry, or adding an attribute to an existing entry.

Other operations on the replica management service consist of multiple parts. For example, consider an operation that copies a file to a storage system and registers the file with the replica management service. Our architecture does not assume that complex, multi-part operations are atomic. Depending on when a failure occurs during a multi-part operation, the information registered in the replica management service may become corrupted. We guarantee that if failures occur during complex operations, we will rollback the state of the replica management service to the previously-consistent state before the operation began. This requires us to save sufficient state about outstanding complex operations to revert to a consistent state after failures.

No Distributed Locking Mechanism

It is possible for users to corrupt our replica management service by changing or deleting files on an underlying storage system without informing the replica management service. We strongly discourage such operations, but the architecture does not prevent them. After such operations, information registered in the replica catalog may not be consistent with the actual contents of corresponding storage systems. The replica management service could avoid such corruption if it could enforce that all changes to storage systems be made via calls to the replica management service. Enforcing this requirement would require a distributed locking mechanism that prevents changes to registered storage locations except via authorized replica management operations. Because of the difficulty of implementing such a distributed locking mechanism, our architecture does not assume that locking is available and does not guarantee that catalog corruption will not occur.

5.2 A Replica Management System Scenario

One of the key features of our architecture is that the replica management service is orthogonal to other services such as replica selection and metadata management. Figure 6 shows a scenario where an application accesses several of these orthogonal services to identify the best location for a desired data transfer. For example, consider a climate modeling simulation that will be run on precipitation data collected in 1998. The scientist running the simulation does not know the exact file names or locations of the data required for this analysis. Instead, the application specifies the characteristics of the desired data at a high level and passes this attribute description to a metadata catalog (1). The metadata catalog queries its attribute-based indexes and produces a list of logical files that contain data with the specified characteristics. The metadata catalog returns this list of logical files to the application (2). The application passes these logical file names to the replica management service (3), which returns to the application a list of physical locations for all registered copies of the desired logical files (4). Next, the application passes this list of replica locations (5) to a replica selection service, which identifies the source and destination storage system locations for all candidate data transfer operations. In our example, the source locations contain files with 1998 precipitation measurements, and the destination location is where the application will access the data. The replica selection service sends the candidate source and destination locations to one or more information services (6), which provide estimates of candidate transfer performance based on grid measurements and/or predictions (7). Based on these estimates, the replica selection service chooses the best location for a particular transfer and returns location information for the selected replica to the application (8). Following this selection process, the application performs the data transfer operations.

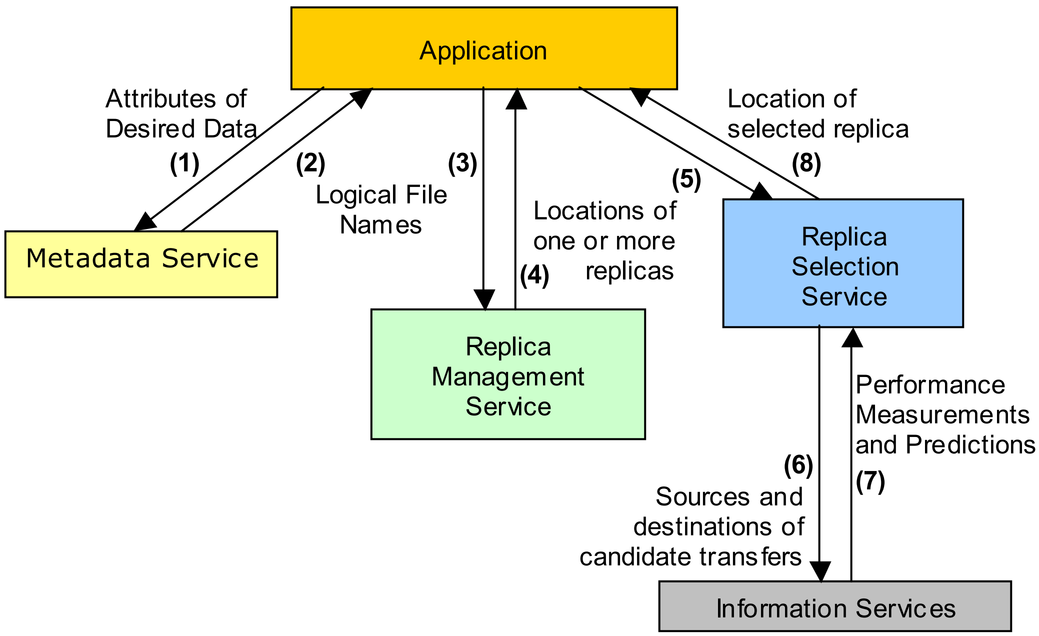

Figure 6: Shows a data selection scenario where the application consults the metadata service, replica management service and replica selection service to determine the best source of data matching a set of desired data attributes.

5.3 The Replica Catalog

A key component of the replica management system is the *replica catalog*. As mentioned above, the purpose of the replica catalog is to provide mappings between logical names for files or collections and one or more copies of those objects on physical storage systems. The catalog registers three types of *entries*: logical collections, locations, and logical files.

A *logical collection* is a user-defined group of files. We expect that users will often find it convenient and intuitive to register and manipulate groups of files as a collection, rather than requiring that every file be registered and manipulated individually. Aggregating files should reduce both the number of entries in the catalog and the number of catalog manipulation operations required to manage replicas.

Location entries in the replica catalog contain the information required for mapping a logical collection to a particular physical instance of that collection. The location entry may register information about the physical storage system, such as the hostname, port and protocol. In addition, it contains all information needed to construct a URL that can be used to access particular files in the collection on the corresponding storage system. Each location entry represents a complete or partial copy of a logical collection on a storage system. One location entry corresponds to exactly one physical storage system location. The location entry explicitly lists all files from the logical collection that are stored on the specified physical storage system.

Each logical collection may have an arbitrary number of associated location entries, each of which contains a (possibly overlapping) subset of the files in the collection. Using multiple location entries, users can easily register logical collections that span multiple physical storage systems.

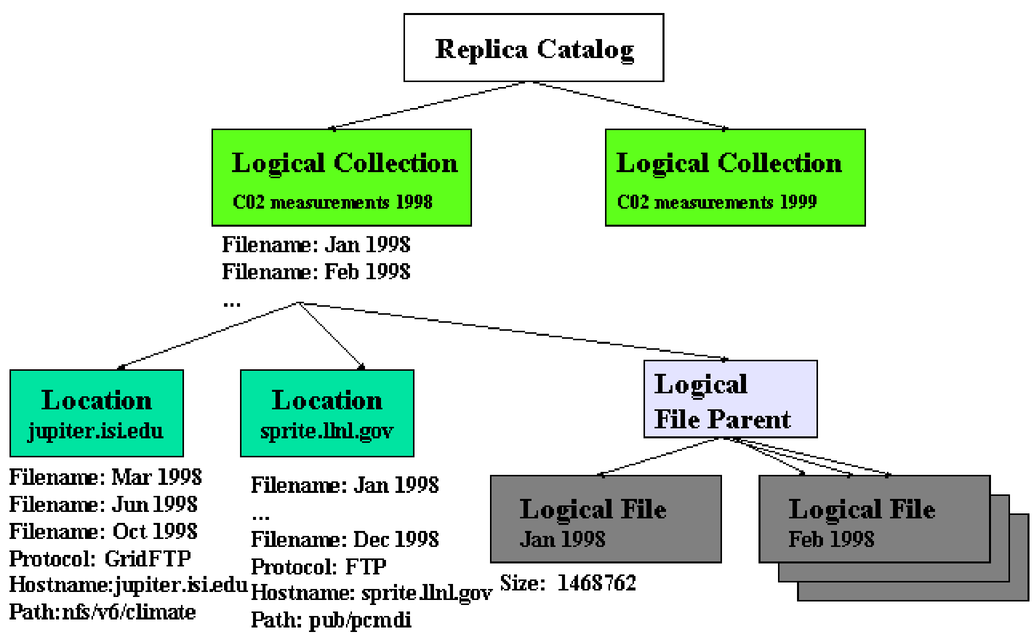

Figure 7: A Replica Catalog for a climate modeling application.

Despite the benefits of registering and manipulating collections of files using logical collection and location objects, users and applications may also want to characterize individual files. For this purpose, the replica catalog includes optional entries that describe individual *logical files*. Logical files are entities with globally unique names that may have one or more physical instances. The catalog may optionally contain one logical file entry in the replica catalog for each logical file in a collection.

Figure 7 shows an example replica catalog for a climate modeling application. This catalog contains two logical collections with CO_2 measurements for 1998 and 1999. The 1998 collection has two physical locations, a partial collection on the hos jupiter.isi.edu and a complete collection on sprite.llnl.gov. The location entries contain attributes that list all files stored at a particular physical location. They also contain attributes that provide all information (protocol, hostname, port, path) required to map from logical names for files to URLs corresponding to file locations on the storage system. The example catalog also contains logical file entries for each file in the collection. These entries provide size information for individual files.

We have implemented an API for low-level replica catalog manipulation as a C library called **globus_replica_catalog.c**. In addition, a straightforward command-line tool provides similar functionality. There are three types of operations on replica catalog entries. First, the API provides functions to create and delete catalog entries, for example, to register a new collection or location. Second, the API provides functions to add, list or delete individual attributes of a catalog entry. For example, as an experimental physics application produces new data files, the collection owner can register these files with the replica catalog by adding their names as attributes of existing logical collection and location entries. Third, the API provides functions to list or search catalog entries, including complex search operations that find all physical locations where a particular set of logical files is stored.

5.4 Replica Management

Above the replica catalog in our layered architecture is the replica management layer. Replica management operations combine storage system accesses with calls to the replica catalog. There are three of these combined operations in the replica management layer: publish, copy and delete.

The **file_publish** operation copies a file from a storage system that is not currently registered in the replica catalog onto a registered storage system and updates the replica catalog to record the presence of the new file. (A registered storage system is one that has existing location and collection entries in the replica catalog.) The publish command is intended to meet the needs of application communities that produce data files and then make them available for general use. For example, in high-energy physics or climate modeling communities, sites that produce data may keep new files private initially while local researchers operate on them. Over time, these files are published and made available to a larger group of researchers by registering them in the replica catalog.

The **file_copy** operation is quite similar to the publish command. It copies a file from one registered replica catalog location's storage system to another and updates the destination location and collection object to reflect the newly-copied file. The difference between the copy and publish operations is that the source file for the copy operation is on a registered storage system. The copy operation is used when a new replica of an existing registered file is created.

Finally, the **file_delete** operation removes a filename from a registered replica catalog location entry and optionally also deletes the file from the storage system associated with the replica catalog location.

These combined storage system and replica catalog operations present reliability challenges for the replica management system. Failures in the middle of these combined operations could result in inconsistencies in the replica catalog or storage systems. To prevent such inconsistencies from corrupting the replica management system, we provide functions to attempt to **restart** failed operations and to **rollback** operations that fail to a previous consistent catalog state.

The replica management system contains additional operations. These include functions for creating sessions for a series of operations upon the replica catalog and simple, atomic updates to individual replica catalog entries.

The functions described in this section have been implemented in the **globus_replica_management** API.

5.6 Replica Management Service Implementation and Performance

In this section, we present preliminary performance results for our prototype implementation of the Globus replica management architecture. We have implemented the replica catalog and replica management APIs in C. The replica catalog itself is currently implemented as a Lightweight Directory Access Protocol (LDAP) directory, although future implementations may use relational databases. Our experimental replica catalog is a Netscape Directory Server version 4.12 LDAP directory configured with a cache size limit of 100 objects and a limit of 100 megabytes for caching database index files. This LDAP server runs on a 333 MHz Sun Sparc Ultra-5 workstation with 384 megabytes of memory running the SunOS version 5.7 operating system. The LDAP directory maintains an index on filename attributes of logical collection and location entries. A single client submits requests to the LDAP server in our tests.

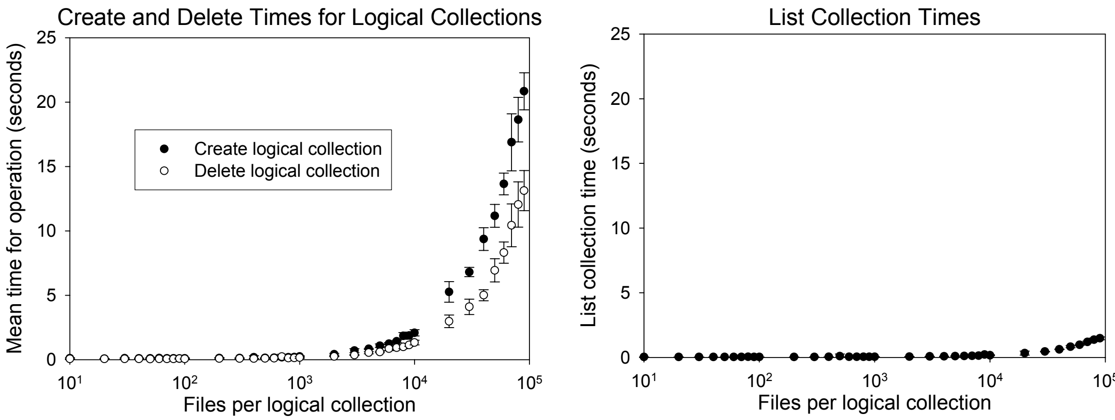

Figure 8: Microbenchmark performance results for low-level replica catalog API

Figure 8 shows microbenchmark performance results for three operations of the low-level replica catalog manipulation API: creating and deleting a logical collection and listing the contents of a collection entry. Each is graphed on the same scale as the size of collections increases from 1 file to 100,000 files. All measurements are run 10 times, with the graphs indicating mean times for the operations and error bars showing standard deviation. We include only graphs for operations on logical collections. (Graphs for operations on location entries show similar behavior.) The graph on the left indicates the time required to create and delete logical collection entries in the catalog, where each entry has one attribute per logical file name. As the number of files in a logical collection approaches 100,000, the create and delete times increases considerably, to approximately 20 seconds. This is due to the large number (tens of thousands) of filename attributes associated with these large logical collection entries. Although these times are relatively long, creation and deletion operations should be fairly rare. We also note that our experimental LDAP server runs on a relatively low-performance workstation.

We expect that list and search operations will be more common than create or delete operations. The graph on the right of Figure 8 shows the time to list the contents of a logical collection entry in the catalog. The list operation is much faster than create/delete operations, with times ranging from well under one second for small collections to approximately one second for collections with tens of thousands of files.

5.7 Reliability and Availability of the Replica Management Service

If the replica management service fails during an operation, it will be unavailable for some time and its state upon recovery will be indeterminate. The current design does not easily provide the ability to replicate or distribute the replica catalog, although distribution can be performed if supported by the catalog implementation technology.

The next version of the replica catalog design and implementation will be distributed. The distributed replica catalog is being designed with the following goals. First, the catalog must avoid single points of failure. The contents of the catalog must be distributed so that a failure of some portion of the catalog does not make local copies of data unavailable. Second, the catalog should scale for very large numbers of logical files, logical collections and replicas. The catalog should perform efficient replication and deletion operations for files and collections of files. Finally, the catalog should perform well for common queries, including determining whether there is a local replica of a logical file and finding all replicas of a logical file.

6 Related Work

Three grid computing projects address similar problems to those discussed in this paper.

The Storage Resource Broker (SRB) [14] from the San Diego Supercomputing Center is middleware infrastructure that provides a uniform, UNIX-style file I/O interface for accessing heterogeneous storage resources distributed over the wide area network. Using its Metadata Catalog (MCAT), SRB provides collection-based access to data based on high-level attributes rather than on physical filenames. SRB also supports automatic replication of files on storage systems controlled by SRB. In contrast to the layered Globus architecture with direct user and application control over replication, SRB uses an integrated architecture, with all access to data via the SRB interface and MCAT and with SRB control over replication and replica selection.

The Kangaroo data management system is part of the Condor high throughput computing project [15]. Kangaroo is intended to provide a reliable data movement service. A chain of Kangaroo servers temporarily buffer I/O operations, making opportunistic use of available disks and networks. Because I/O operations are handled by background processes, application CPU activity can proceed in parallel with I/O operations, resulting in better performance. In addition, Kangaroo offers greater reliability, since the Kangaroo service continues to perform I/O operations even if the process that initiated these requests fails. Kangaroo servers can service I/O requests from any available replica of the data; the use of an alternate replica is transparent to the application.

The Legion project provides an object-oriented middleware infrastructure for distributed computing environments [16]. The Legion File System (LegionFS) provides a BasicFileObject with object methods that resemble UNIX read, write and seek system calls. LegionFS uses Legion facilities for naming, security, scalability and extensibility. Replication could be provided by LegionFS by classes that map object identifiers to multiple physical object addresses. LegionFS could also be extended to map human-readable context names for objects to multiple object identifiers. In contrast to Legion, Globus does not provide an integrated, object-oriented infrastructure with a common name space. We believe that the layered Globus architecture, which extends the network protocol stack for grid services and makes extensive use of standard protocols, is a more flexible architecture and will yield higher performance.

7 Conclusions

We have argued that high-performance, distributed data-intensive applications require two fundamental services: *secure, reliable, efficient data transfer* and the ability to *register, locate, and manage multiple copies* of datasets. These two services can be used to build a range of higher-level capabilities, including reliable creation of a copy of a data collection at a new location, selection of the best replica for a data transfer operation based on performance, and automatic creation of new replicas in response to application demands.

We have presented our design and implementation of these two services. The GridFTP protocol implements extensions to FTP that provide GSI security and parallel, striped, partial, and third-party transfers, while the Globus replica management architecture supports the management of complete and partial copies of datasets. Performance studies of both components provide promising results.

These and other Globus Toolkit services are being applied by ourselves and others in a variety of large-scale application projects, including the Particle Physics Data Grid (www.ppdg.net), Earth Systems Grid, Grid Physics Network (www.griphyn.org), and European Data Grid (grid.web.cern.ch/grid) projects. Experience with these applications will motivate further refinements and additions to the services described here. We are already planning extensions, such as automated replica management, community-based access control, automated buffer size negotiation, and server-side data reduction [17].

Acknowledgements

We are grateful to Marcus Thiebaux and Soonwook Hwang for their work characterizing the performance of LDAP servers; to Brian Toonen, who helped to optimize the GridFTP code; to Gail Pieper, Laura Pearlman and Ewa Deelman for comments on this paper; and to the many colleagues in the sciences who helped us understand their requirements. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation.

References

[1] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets," J. of Network and Computer Applications, vol 23, no. 3, p 187-200, July 2000.

[2] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", To appear: Intl J. High Performance Computing Applications, 15(3), 2001.

[3] I. Foster, C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” Intl. J. Supercomputer Applications, 11(2): 115-128, 1997.
[4] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, K. Stockinger, “Data Management in an International Grid Project”, 2000 Intl. Workshop on Grid Computing (GRID 2000), Bangalore, India, December 2000.
[5] K. Holtman, “Object Level Replication for Physics”, Proceedings of 4th Annual Globus Retreat, Pittsburgh, July 2000.
[6] I. Foster, C. Kesselman (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
[7] B. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, F. Drake, "A Network-Aware Distributed Storage Cache for Data Intensive Environments", Proc. of IEEE High Performance Distributed Computing Conference (HPDC-8), August 1999.
[8] "Basics of the High Performance Storage System", www.sdsc.edu/projects/HPSS
[9] C. Baru, R. Moore, A. Rajasekar, M. Wan, "The SDSC Storage Resource Broker," Proc. CASCON'98 Conference, Nov.30-Dec.3, 1998, Toronto, Canada.
[10] Lili Qiu, Yin Zhang, Srinivasan Keshav, “On Individual and Aggregate TCP Performance,” Proc. of 7th Intl. Conference on Network Protocols (ICNP'99), Toronto, Canada, 1999.
[11] B. Tierney, W. Johnston, J. Lee, G. Hoo, “Performance Analysis in High-Speed Wide Area IP over ATM Networks: Top-to-Bottom End-to-End Monitoring,” *IEEE Networking,* 1996.
[12] dast.nlanr.net/Projects/Iperf/
[13] D. Gunter, B. Tierney, B. Crowley, M. Holding, J. Lee, “NetLogger: A Toolkit for Distributed System Performance Analysis”, Proc. of the IEEE Mascots 2000 Conference (Mascots 2000), August 2000.
[14] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," presented at Proc. CASCON'98 Conference, 1998.
[15] D. Thain, J. Basney, S. Son and M. Livny, “The Kangaroo Approach to Data Movement on the Grid,” Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10), August 2001.
[16] B. White, M. Walker, M. Humphrey and A. Grimshaw, “LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications”, to appear in Proceedings of SC’2001, November 2001.
[17] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and Saltz, J., “DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems,” Proc. 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems, 2000, 119-133.