A Replica Management Service for High-Performance Data Grids

The Globus Data Management Group

http://www.globus.org/datagrid

1 Introduction

In many scientific disciplines, a large community of users requires remote access to large datasets. An effective technique for improving access speeds and reducing network loads can be to replicate frequently accessed datasets at locations chosen to be "near" the eventual users. However, organizing such replication so that it is both reliable and efficient can be a challenging problem, for a variety of reasons. The datasets to be moved can be large, so issues of network performance and fault tolerance become important. The individual locations at which replicas may be placed can have different performance characteristics, in which case users (or higher-level tools) may want to be able to discover these characteristics and use this information to guide replica selection. In addition, different locations may have different access control policies that need to be respected.

These considerations motivate this proposal for a replica management system charged with managing the copying and placement of files in a distributed computing system so as to optimize the performance of the data analysis process. Our goal in designing this service is not to provide a complete solution to this problem but rather to provide a set of basic mechanisms that will make it easy for users, or higher-level tools, to manage the replication process.

Our proposed replica management service provides the following basic functions:

- The registration of files with the replica management service.
- The creation and deletion of replicas for previously registered files.

- Enquiries concerning the location and performance characteristics of replicas.
- Selection of the best replica for a data transfer operation based on predicted performance.

The Globus replica management service has a layered architecture. At the lowest level is a *Replica Catalog* that allows users to register files as logical collections and provides mappings between logical names for files and collections and the storage system locations of file replicas. Building on this basic component, we provide a low-level API that performs catalog manipulation and a higher-level Replica Management API that combines storage access operations with calls to low-level catalog manipulation functions. These APIs can be used by higher-level tools that select among replicas based on network or storage system performance, or that create (or delete) new replicas automatically at desirable locations.

This document describe the higher-level Replica Management API that combines storage system operations with calls to low-level replica catalog API functions. Key concepts include the following:

- ***Registration:*** Registration operations add information about files on a physical storage system to existing location and logical collection entries. For example, when a long-running scientific experiment periodically produces new data files, these files are made available to users by registering them in existing location and collection entries.
- ***Copying****:* This operation *copies* a file between two storage systems that are registered as locations of the same logical collection and *updates* the destination's location entry to include the new file.
- ***Publishing:*** The publishing operation takes a file from a source storage system that is not represented in the replica catalog, *copies* the file to a destination storage system that is represented in the replica catalog, and *updates* the corresponding location and logical collection entries.

In this document, we first describe the requirements of two different application domains, high-energy physics and climate modeling, which motivate the replica management service design. We then provide a detailed description of our first implementation of the Replica Management API, followed by a discussion of additional functionality that will be supported in later versions.

A note on the word "replica": The word *replica* has been used in a variety of contents with a variety of meanings. For example, it is sometimes used to mean "a copy of a file that is guaranteed to be consistent with the original, despite updates to the latter." For the purposes of this document, we define a replica to be simply a *managed copy of a file*. The replica management system controls where and when copies are created, and provides information about where copies are located. However, the system does *not* make any statements about file consistency. In other words, it is possible for copies to get out of date with respect to one another, if a user chooses to modify a copy.

2 Motivating Examples

We present two examples of application domains in which we believe our replication service can be useful.

2.1 Objectivity Databases in Physics Experiments

Particle physics experiments are characterized by the need to perform analysis over large amounts of data. To enable the selection of the data of interest, and to simplify the development of analysis codes, several such experiments (ATLAS, BaBar, CMS), have selected object-oriented technology as a structured file representation for storing the physics data to be analyzed. Users at many sites worldwide then need to be able to access data contained in these databases.

2.1.1 Use of Objectivity

In the physics experiments of interest, Objectivity (http://www.Objectivity.com/) is the database technology that has been selected for data storage. Objectivity stores collections of objects in a single file called a *database.* Databases can be grouped into larger collections called *federations.* Objects in one database can refer (point) to objects in another database in the same or a different federation. In the physics experiments we are considering, each database file is several gigabytes in size. Federations are currently limited to 64K files; however, future versions of Objectivity will eliminate this restriction. Some experiments plan to exploit this feature and anticipate creating federations with millions of individual database files.

There are two types of data generated by physics experiments:

- Experimental data that represents the information collected by the experiment. There is a single creator of this data, and once created, it is not modified. However, data may be collected incrementally over a period of weeks.
- Metadata that captures information about the experiment (such as the number of events) and the results of analysis. Multiple individuals may create metadata[CFK1]. The volume of metadata is typically smaller than that of experimental data.

The consumers of the various types of data can number in the hundreds or thousands. Because of the geographic distribution of the participants in a particle physics experiment, it is desirable to make copies of the data being analyzed so as to minimize the access time to the data. This replication is complicated by several factors, e.g.:

- Complete data sets can be very large. Thus one may need to replicate only "interesting" subsets of the data. However, because of the way Objectivity is being used by the various physics experiments, the subsets of the data that need to be updated may span many database files, or even many federations.
- Database files may be modified. One cause for this is that the write time into database files can be quite large. In some experiments (Babar, http://www.slac.stanford.edu/BFROOT/), it can take several weeks to collect an entire data set. However, one would like to make data available incrementally, potentially every few days.

One approach to this distribution problem would be to use existing Objectivity methods for distributing database files in a federation across multiple machines. However, Objectivity was not designed to operate in the regime of wide-area, very high performance networks. Discussions with engineers within Objectivity (Harvey Newman, personal communication) confirm that this is not a recommended approach. Hence, we believe that our replica management service may represent a better approach.

2.1.2 Inter-Database References

One issue that has the potential to complicate the replication process is that Objectivity keeps track of where objects exist within a federation by building a catalog. It is possible to move a database from one federation to another (e.g., federations at different sites) by performing an *export* from the first federation followed by an *import* into the second. We note that maintenance of the catalog depends on the name of the database file, and not the contents. Therefore catalog operations can take place concurrently with data movement operations.

One potential problem in the export/import strategy is the existence of cross database pointers. Five distinct strategies have been proposed to address this problem:

1. Ignore the problem, with the knowledge that cross database links may be broken in a replicated file.
2. Identify cross-database references, and null them out, creating a self-contained database file, with the loss of some information.
3. Insure that whenever any database file is moved, all of the databases that may be referred to by this database are moved as well. This requires using database schema to identify the complete set of inter-dependent files.
4. Generate an intermediate database file that contains all objects of interest, including external references. This method can be viewed as an optimization of the previous method, with the advantage of reducing the amount of data that may be transferred. The disadvantage of this method is increased complexity in keeping track of replica location and potentially reducing the effectiveness of data caches.
5. Replace the interobject references with "remote references" that can be resolved via a (slow) inter-site object access mechanism. (This is basically a variant of #1, in which we have a global name space for object identifiers and can use the catalog to work out where "nonlocal" databases are located. This is what BaBar does, for example[CFK2].)

We note that all five methods are equivalent from our perspective: if any of them is adopted, then all can employ our proposed replication mechanism, which enables the copying of a set of databases files from one location to another.

2.1.3 Database Updates

Another potentially complicating issue is that while in principle the primary databases produced by physics experiments are read-only (once created, their contents do not change), we find in practice that during initial production of the data (a period of several

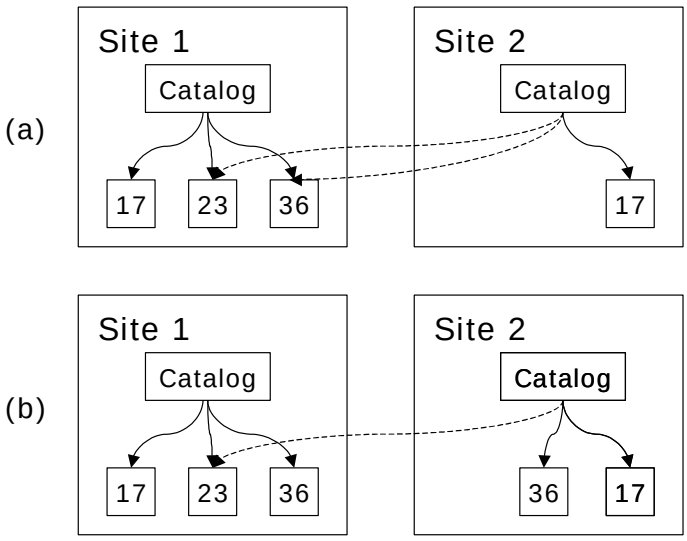
weeks) database files change as new objects are added. Users want access to these database files during this production period, so we face a need for updating of replicated database files. In addition, metadata may be either modified, or augmented over time.

Different experiments tend to use Objectivity in different ways and hence have somewhat different requirements in this area. We explain the requirements in two experiments, BaBar and CMS.

2.1.3.1 BaBar

In BaBar, objects are appended to the databases in a federation over the course of several weeks. The practical impact of these *logical* append operations is that many database files can be changing simultaneously during this period, as they fill up, after which they do not change further. Individual database files are around ten GB in size, currently, due to an Objectivity limit of 64K files, due to be removed at the end of 2000; the goal is to reduce this size to O(1) GB, in which case there will be millions of files. In the latter case, updates to files during the production phase will become less of a problem. Metadata files are currently 2 GB in size and will also become smaller.

The following figure shows the sort of database structure used in BaBar. Different sites maintain identical catalogs, with some catalog entries referencing local copies of database files and others referencing remote copies. In (b), we see the result of replicating database file 36 at Site 2: the file is copied and the catalog at Site 2 is updated to reference the local copy.



A master-slave model is used to propagate modifications. During the production period, updated (whole) files are transmitted to destination locations once a week. They would like these updates to occur more frequently: say once every three days. The frequency of updates is constrained by trans-Atlantic bandwidth.

A master-slave model is also used to control update access to databases containing metadata. These semantics are provided by partitioning the object identifier space, so that each participating site has exclusive write access to a predefined subset of objects in the federation.

2.1.3.2 CMS

The CMS collaboration has adopted a different approach to the use of Objectivity, in which data files do not change once created. However, over time, additional database files may be added to the federation, and if so, then metadata files *do* change to reflect the increasing total number of events in the database. Metadata updates need to be propagated to all replicas.

The CMS collaboration are interested in supporting distribution of "partial databases" containing only those objects of interest to a particular scientist. We do not address this requirement in this document.

2.2 Climate Model Data

In the climate modeling community, modeling groups sometimes generate large "reference simulations" that are then of interest to a large international community. The output from these simulations can be large (many Terabytes). The simulation data is typically generated at one or more supercomputer centers and is then "released" in stages to progressively larger communities: first the research collaboration that generated the data, then perhaps to selected colleagues, and eventually to the entire community.

In contrast to the physics community, data is not maintained in databases but rather as flat files, typically structured using for example NetCDF, with associated metadata. In addition, files are not updated once released. However, we believe that the basic requirements for data distribution (replication) are sufficiently similar that a common replica management service can be employed.

3 Building Blocks

Our proposed replica management service builds upon two elements of the Globus data management architecture: the Replica Catalog, along with the catalog manipulation API, and the GridFTP protocol that provides secure, efficient, parallel data transfer.

3.1 The Replica Catalog and its API

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.

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 2 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.

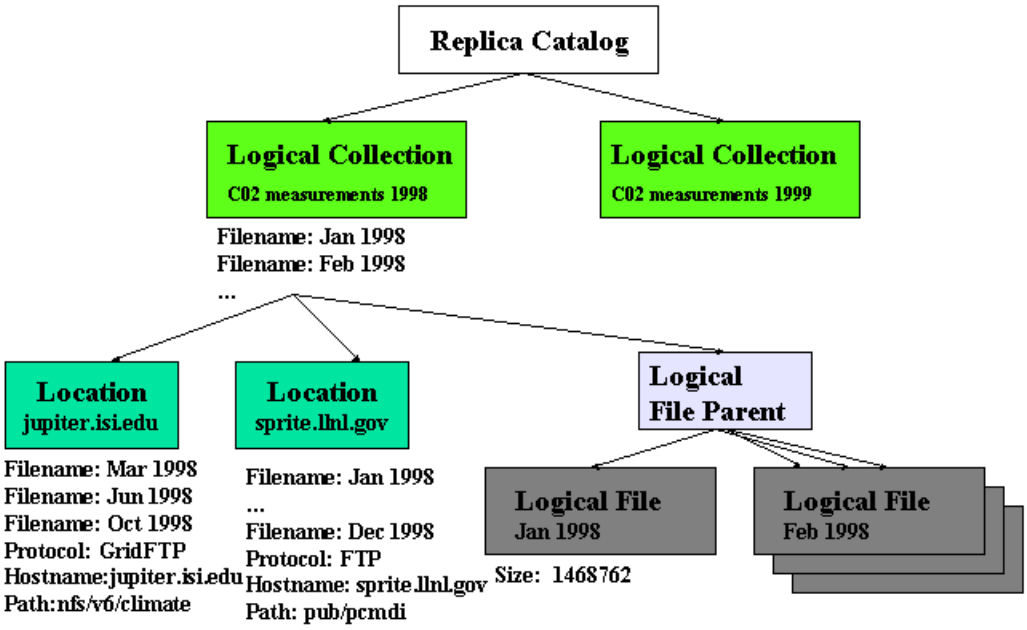


Figure 2: A Replica Catalog for a climate modeling application.

We have implemented an API for low-level replica catalog manipulation as a C library called **globus_replica_catalog.c**. The Replica Management API will make calls to this lower-level catalog manipulation API to manipulate replica catalog entries. 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.

3.2 GridFTP

GridFTP is a Grid data transfer and access protocol that provides secure, efficient data movement in Grid environments. This protocol, which extends the standard FTP protocol, provides a superset of the features offered by the various Grid storage systems currently in use. GridFTP functionality includes the following features:

- **Grid Security Infrastructure and Kerberos support:** GridFTP supports robust and flexible authentication as well as user-controlled setting of various levels of data integrity and/or confidentiality.
- **Third-party control of data transfer:** GridFTP third-party operation allow 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.
- **Parallel data transfer:** GridFTP supports parallel data transfer using multiple TCP streams in parallel, even between the same source and destination, to improve aggregate bandwidth over using a single TCP.
- **Striped data transfer:** Data may be striped or interleaved across multiple servers. GridFTP includes extensions that initiate striped transfers, which use multiple TCP streams to transfer data that is partitioned among multiple servers and provide provide further bandwidth improvements over those achieved with parallel transfers.
- **Partial file transfer:** GridFTP provides commands to support transfers of arbitrary subsets or regions of a file. In contrast, the standard FTP protocol only allows transfer of the remainder of a file starting at a particular offset.
- **Automatic negotiation of TCP buffer/window sizes:** 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. Using optimal settings for TCP buffer/window sizes can dramatically improve data transfer performance.
- **Support for reliable and restartable data transfer:** 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.

Our current implementation of the GridFTP protocol consists of two principal C libraries: the globus_ftp_control_library and the globus_ftp_client_library. The Replica Management API will perform data transfer operations by calling functions from these

two APIs. 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. 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. Eventually, this API will also provide functions for automatic negotiation of TCP buffer/window sizes; however, these are not implemented in the current version of the API.

3.3 Other Globus Services

In addition, we may use the following Globus services when developing replica management functions:

1. The Grid Security Infrastructure (GSI) protocol and tools, used to provide single-sign-on, public-key authenticated access to remote data and computers. GSI-FTP uses this.
2. The Globus Resource Allocation and Management (GRAM) protocol for accessing remote computation. We can use this, for example, to invoke remote cataloging operations following successful completion of a replica creation.
3. The Grid Information Service (GIS), which provides Lightweight Directory Access Protocol (LDAP) to information about the structure and state of storage systems, computers, and networks

4 Important Concepts and Design Decision

In this section, we discuss several important assumptions and decisions that affect the design and implementation of the replica management API.

4.1 Replication Semantics

We explicitly decide not to enforce any replica semantics. For multiple replicas of a collection, we do not maintain any information on which was the "original" or "source" location from which one or more copies were made. We do not maintain any information in the replica catalog about who is the "master" in a master/slave relationship.

As already mentioned, the word *replica* has been used in a variety of contents with a variety of meanings. For example, it is sometimes used to mean "a copy of a file that is guaranteed to be consistent with the original, despite updates to the latter." For the purposes of this document, we define a replica to be simply a *managed copy of a file*. The replica management system controls where and when copies are created, and provides information about where copies are located. However, the system does *not* make any statements about file consistency. In other words, it is possible for copies to get out of date with respect to one another, if a user chooses to modify a copy.

4.2 Replicating Individual Files

This API provides functions for managing replication for individual files. It does not contain functions for managing multiple files. However, the latter can be efficiently built on the former, as care was taken in this API to allow for caching of session state across several single-file operations. The advantages of the single file approach is that it simplifies the API and implementation, and it allows for various multi-file approaches to be built on top of this basic API.

4.3 Replica Catalog Consistency

We guarantee that if a file appears in the replica catalog, it is complete and uncorrupted on the corresponding storage system. Note that our consistency guarantee requires that that all operations on replicas are performed through the replica management and replica catalog APIs. It is possible for files to become inconsistent if they are altered on the underlying storage system without informing the replica catalog. For example, the user must ensure that the contents of a file do not change while a file copy is in process; the replica management system does not enforce protection of the source file to avoid this corruption. This consistency guarantee enforces an ordering on some operations. For example, a file copy operation copies a file from a source to a destination operation and updates the corresponding destination location operation. The update of the location object must not occur before the copy operation to the destination storage system has successfully completed.

Another invariant of the replica catalog is that a file must appear on the list of files in a logical collection if it appears in any location of that collection. Conversely, we do not require that every file in a logical collection exist at some location.

The replica catalog must remain consistent at all times. Users must continue to be able to access the catalog during a replica management operations or after a replica management operation fails.

In the event of failure, we support rollback to return the replica catalog to a previously consistent state.

4.4 Rollback

If a replica management operation fails, all information necessary to rollback the operation should be stored in the replica catalog. Rollbacks ensure the consistency of the replica catalog by returning the catalog to the state it was in before the failed operation began. We currently plan to provide rollback for four functions on files: copy, register, publish and delete. These operations are fully explained in Section 6.

In order to provide consistency and rollback, we have introduced a *rollback record* into the globus_replica_catalog library. The rollback record is stored as an object in the replica catalog and is a child of a location object. Before any of the four operations for which rollback support is provided, we perform an atomic add of a rollback record object to the catalog. For any subsequent actions that might need to be rolled back in the case of failure, we atomically modify the rollback record entry, providing enough information to rollback any state changes if the operation fails.

4.5 Locking

Rollback records should not be confused with locks. We do not provide a distributed lock manager as part of this implementation of the Replica Management API. Locking is an orthogonal service. Some applications may require locks. However, we are not currently addressing this issue. Eventually, we may implement a distributed lock manager, but we have not committed to doing this.

One reason is that the underlying file system or storage system may not provide sufficient consistency operations to guarantee locking. We cannot provide guarantees that are not supported by the underlying storage system.

4.6 Access Control

The current replica management API implementation does not address access control. The Globus project is currently working on a Community Authorization Server (CAS) that will provide improved authentication and capability-based authorization for access to storage objects. We expect to incorporate the CAS into the second phase of replica management implementation.

4.7 Replication and Distribution of Replica Catalogs

The issue of catalog reliability often arises in discussions of the replica catalog. If the replica catalog fails during an operation, the catalog's state will be indeterminate. As with any system, the level of reliability and consistency in the face of failure involve tradeoffs of cost and complexity. If high levels of availability are required, then the system user must devote adequate resources to replicating the replica catalog, performing checkpoint operations and avoiding single points of failure. This robustness must be engineered into the catalog service. These issues are outside the scope of the replica management API.

Currently, our replica catalog is implemented as an LDAP directory. LDAP provides limited support for replicating directories…

Distribution is easily implemented in LDAP directories using index nodes…

4.8 Post-Processing Files After Data Transfer

The Replica Management API is designed to provide complex operations that include both data transfer and replica catalog updates. Some applications want to perform post-processing operations between data transfer operations and updating the catalog. Examples of post-processing include decryption of data, running verification operations such as checksums to confirm that data files were not corrupted during data transfer, or attaching transferred files to an object-oriented database.

For this version of the replica management API implementation, we restrict the set of allowed post-processing operations to those that do not change data files, since such transformation operations are essentially creating new data files. Of the examples above, running checksum computations or registering files in a database would be allowed, but decryption of the data would not.

In particular, only post-processing operations that allow simple rollback on failures are allowed. We stipulate that after a failure, we must be able to restart the post-processing operation from the beginning, and it must be possible to run the post-processing operation several times if necessary.

4.9 Synchronous Functions

The functions of the replica management API are synchronous. We made the decision not to support asynchronous functions in our initial implementation of the API. One reason is that the underlying replica catalog API is currently also synchronous, and we would have to make that API asynchronous before making the replica management API asynchronous. A consequence of this decision is that we are threadsafe but not multithreaded. Those who want multiple outstanding requests to the replica management system will have to run multiple instances of the program.

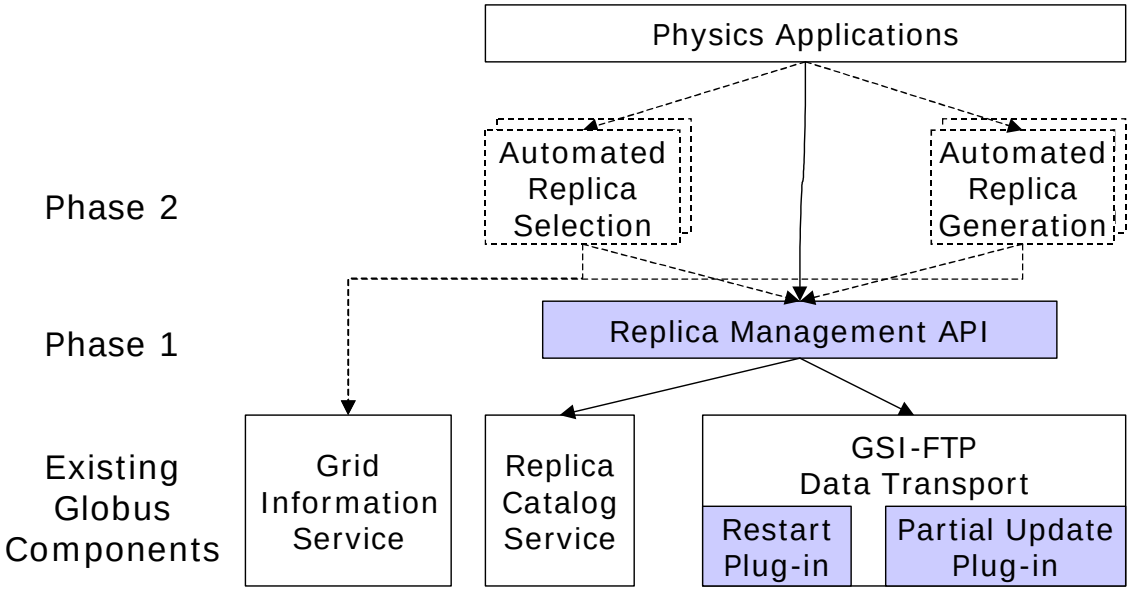
4.10 Storage System Requirements

We require that storage systems using the Globus Replica Management service support the GridFTP protocol.

5 Proposed Replica Management Solution

The following figure shows the general structure of the proposed system.

<Need to add explanatory text here.>



6 API Overview

In this section, we provide a brief overview of the functions provided by the Globus Replica Management API. For a complete description of the API, refer to http://www-unix.mcs.anl.gov/~quesnel/ReplicaManagement/html/. API functions can be loosely broken into two categories: session management and catalog creation/file management.

These functions are not application-specific. It is the responsibility of the application to identify the identify the files that need to be managed and invoke the appropriate API functions. For example, applications using the Objectivity object-oriented database must first identify objects of interests, then map these objects to specific files, and then make API function calls to manage these files.

6.1 Session Management

This portion of the API provides functions for managing session handles and attributes. This provides the ability to cache connection states to GridFTP and LDAP servers, so that multiple operations on multiple files can be executed through this API.

The section of the API also provides functions for rolling back the effects of an operation that must be aborted and for restarting an operation.

6.1.1 Session Handles and Attributes

A session handle is a data structure that maintains information about the configuration and state of a connection to the replica management system. The session handle data structure includes a data structure for handle attributes. These attributes describe the ftp client and the connection to the replica catalog. The handle attributes data structure also contains a placeholder for attributes that will be used in the future to describe a connection to a community authorization server (CAS).

The API includes functions to initialize and destroy a session handle. In addition, there are functions to initialize and destroy the session attribute data structure. For each type of attribute, there are functions to set and get the attribute value.

6.1.2 Rollback

The API includes a rollback function call whose purpose is to return the replica catalog to its previous state prior to the current operation. Since a session may consist of a series of function calls, the rollback function call requires the user to provide a pointer to the rollback record of the particular operation being rolled back.

6.1.3 Restart

The API includes a restart function that restarts the current operation. The function consults the rollback record stored in the replica catalog, which contains sufficient information to restart the operation. In addition, the user can specify a “pre-register” function and arguments to the function. This function is intended to undo the results of a previously called post-processing function that was executed between copying a file to a storage system and registering it in the replica catalog.

6.2 Catalog Creation and File Management

This section of the API creates new entries in the replica catalog and updates existing location and collection entries to reflect new files as they are registered. It also allows us to delete files from the catalog.

6.2.1 Creating Catalog Entries

There are simple API functions for creating new, empty logical collection and location entries in the catalog. Catalog objects created by these functions contain no filenames.

6.2.2 Registering Files

A file registration operation involves registering a file that already exists on a storage system known to the replica catalog. In other words, there is a location entry in the replica catalog corresponding to the storage system where the file resides. The registration operation adds the file name to existing location and collection objects. No copy operation is performed for the file, and no new replica catalog objects are created.

6.2.3 Publishing Files

A file publication operation involves a file that exists on a storage system that is not known to the replica catalog. In other words, there is no location entry in the replica catalog corresponding to the storage system where the file resides. This operation requires copying the file to a storage system known to the replica catalog and adding the filename to the replica catalog collection and location entries, if it does not already exist. If the file being published is an updated version of an existing file, the user may specify whether or not the existing file is to be overwritten by the new file.

In addition, the user may optionally specify a post-processing or "pre-register" function that is to be executed between the storage copy operation and the update of catalog entries. As described in Section 4, examples of post-processing operations include verification of file correctness using checksums or attaching the file to an object-oriented database. In order to make rollback possible, we require that the pre-register function not alter the contents of the file and that the function must be restartable in case the restart function is called.

6.2.4 Copying Files

A file copy operation copies a file between two storage systems that are registered with the replica catalog as two locations of the same logical collection and *updates* the destination's location entry to include the new file.

6.2.5 Deleting Files

A delete file operation removes the file from a specified replica catalog location and optionally also removes it from the location's corresponding storage system.

7 Ideas for Phase 2 Development

We suggest here some of the features that might be incorporated into higher-level functions during Phase 2 of this project. Note that in contrast to Phase 1 activities, some of the functionality required here may well be application-dependent.

Incorporation of advance reservation: The Globus architecture addresses advance reservation issues via its General-Purpose Architecture for Reservation and Allocation (GARA) system. With appropriate support within storage systems, we can, for example, ensure that there is sufficient space at a destination storage system for a transfer to complete successfully.

Automatic selection of replica source locations: We can imagine a variant of the replica creation function that does not require a "source" as an argument: instead, it consults to replica catalog to determine where replicas are located, consults GIS to determine relevant properties of those locations (e.g., transfer speeds), and then performs copies from the "best" locations. Semi-automatic variants can also be defined. If these techniques are used widely, then the performance and scalability of the replica catalog becomes a significant concern.

Automatic creation of new replicas: Similarly, we can imagine a function that monitors data access patterns and generates new replicas at selected locations in order to reduce overall network load.

Support movement of complete logical collections: We have been asked to extend our replica management API to include the function "Copy a complete logical collection to a specified location." Such a function might require accessing multiple source locations to find all the files in the logical collection.

8 Complementary Activities

We note that the activities planned here enable a wide variety of complementary activities. We hope to work with other groups to realize capabilities such as those listed in Section 7 above as well as the following:

- *Improved logging*. The basic logging capabilities that we require in individual storage systems can be extended to provide for distributed analysis of data access patterns. Integration with logs generated by the replica management service itself and with performance measurement tools will allow for monitoring and improvement of performance. It will be desirable for logs to be maintained on stable and secure storage in order to enable their use by intrusion detection systems.
- *Improved fault recovery*. The replica management service should be improved over time to incorporate increasingly sophisticated fault recovery. For example, it may be desirable to be able to recover from a total loss of the contents of the replica catalog, via reconstruction of the catalog from log records.
- *Improved performance measurement*. Our Phase 1 deliverables will incorporate some basic performance measurement capabilities, but achieving consistently

high performance across transcontinental and intercontinental networks will require more sophisticated measurement and monitoring support.

- *Integration of dynamic container creation.* As noted above, some groups are interested in replication at the object level, via the creation and distribution of "dynamic containers."
- *End-to-end scheduling.* Reservation capabilities in the network as well as in storage systems will allow for end-to-end scheduling of replica creation operations.

Acknowledgments

This document reflects the results of a number of fruitful discussions:

- In Pittsburgh, involving Bill Allcock, Ann Chervenak, Ian Foster, Andy Hanushevsky, Koen Holtman, Carl Kesselman, Asad Samar, Steven Tuecke, and others.
- At Argonne National Laboratory, involving Ann Chervenak, Ian Foster, Wolfgang Hoschek, Carl Kesselman.
- At SLAC, involving Jacob Becla, Ann Chervenak, Ian Foster, Andy Hanushevsky, Carl Kesselman, Bob Jacobsen, Richard Mount, Harvey Newman, Charles Young.
- Via email with numerous people including those named above and Fabrizio Gagliardi.

---

[i] V. Paxson, End-to-End Internet Packet Dynamics, SIGCOMM '97, LBNL-40488 (ftp://ftp.ee.lbl.gov/papers/vp-pkt-dyn-sigcomm97.ps)

Page: 4
[CFK1]We need to clarify metadata update issue: single writer (as in Babar), what about other experiments. Is there updates to objects, or are new metadata objects just added.

Page: 5
[CFK2]Is there a distinction between a reference to a database located in the current server, and one located in a different server. Also, is the story about federations correct: Federations are defined by a catalog. Catalog identifies database files, database files may or may not be local, a database may be in more then one federation.

Page: 14
[CFK3]*AH comments: Some people suggested that there be a* **globus_replica_catalog_iscurrent_file** *function that takes a filename/systemname and indicates whether or not the file is actually current relative to the master. I know this can be done by a higher level function but, perhaps it should be treated as a core function so that everyone doesn't re-implement it. How you decide whether something is current or not is an open question at the moment*

Page: 15
[CFK4]Andy would like to see the following callbacks:

- *Start Callback:* called at initiation of replication operation
- *Complete callback.* User-supplied function to be called at completion of a file transfer operation
- *Finish Callback.* User-supplied function to be called at the completion of a replication operation

I don't understand when these would be called.

Page: 15
[CFK5]Restart behavior might be handled by one of the callbacks.

Page: 16
[CFK6]AH: My suggestion is that the decision of whether or not to store check-sums in the catalog be a site option when the file is registered. Then the checksums should be stored in the catalog since it is meta-data about the file. Note that the checksums must be stored for each destination site. Since the checksums may be very long, it's probably not wise to pass them to GSI-FTP. Instead, there should be a replaceable function that allows GSI-FTP to get an appropriate number of checksums on the fly.