The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets

Ann Chervenak[*] Ian Foster[$+] Carl Kesselman[*] Charles Salisbury[$] Steven Tuecke[$]
[*] Information Sciences Institute, University of Southern California
[$] Mathematics and Computer Science Division, Argonne National Laboratory
[+] Department of Computer Science, The University of Chicago

Abstract

In an increasing number of scientific disciplines, large data collections are emerging as important community resources. In this paper, we introduce design principles for a data management architecture called the *Data Grid.* We describe two basic services that we believe are fundamental to the design of a data grid, namely, storage systems and metadata management. Next, we explain how these services can be used to develop higher-level services for replica management and replica selection. We conclude by describing our initial implementation of data grid functionality.

1 Introduction

In an increasing number of scientific disciplines, large data collections are emerging as important community resources. In domains as diverse as global climate change, high energy physics, and computational genomics, the volume of interesting data is already measured in terabytes and will soon total petabytes. The communities of researchers that need to access and analyze this data (often using sophisticated and computationally expensive techniques) are often large and are almost always geographically distributed, as are the computing and storage resources that these communities rely upon to store and analyze their data [17].

This combination of large dataset size, geographic distribution of users and resources, and computationally intensive analysis results in complex and stringent performance demands that are not satisfied by any existing data management infrastructure. A large scientific collaboration may generate many queries, each involving access to—or supercomputer-class computations on—gigabytes or terabytes of data. Efficient and reliable execution of these queries may require careful management of terabyte caches, gigabit/s data transfer over wide area networks, coscheduling of data transfers and supercomputer computation, accurate performance estimations to guide the selection of dataset replicas, and other advanced techniques that collectively maximize use of scarce storage, networking, and computing resources.

The literature offers numerous point solutions that address these issues (e.g., see [17, 14, 19, 3]). But no integrating architecture exists that allows us to identify requirements and components common to different systems and hence apply different technologies in a coordinated fashion to a range of data-intensive petabyte-scale application domains.

Motivated by these considerations, we have launched a collaborative effort to design and produce such an integrating architecture. We call this architecture the *data grid,* to emphasize its role as a specialization and extension of the "Grid" that has emerged recently as an integrating infrastructure for distributed computation [10, 20, 15]. Our goal in this effort is to define the requirements that a data grid must satisfy and the components and APIs that will be required in its implementation. We hope that the definition of such an architecture will accelerate progress on petascale data-intensive computing by enabling the integration of currently disjoint approaches, encouraging the deployment of basic enabling technologies, and revealing technology gaps that require further research and development. In addition, we plan to construct a reference implementation for this architecture so as to enable large-scale experimentation.

This work complements other activities in data-intensive computing. Work on high-speed disk caches [21] and on tertiary storage and cache management [5, 19] provides basic building blocks. Work within the digital library community is developing relevant metadata standards and metadata-driven retrieval mechanisms [16, 6, 1] but has focused less on high-speed movement of large data objects, a particular focus of our work. The Storage Resource Broker (SRB) [2] shows how diverse storage systems can be integrated under uniform metadata-driven access mechanisms; it provides a valuable building block for our architecture but should also benefit from the basic services described here. The High Performance Storage System (HPSS) [24] addresses enterprise-level concerns (e.g., it assumes that all accesses occur within the same DCE cell); our work addresses new issues associated with wide area access from multiple administrative domains.

In this paper, we first review the principles that we are following in developing a design for a data grid architecture. Then, we describe two basic services that we believe are fundamental to the design of a data grid, namely, storage systems and metadata management. Next, we explain how these services can be used to develop various higher-level services for replica management and replica selection. We conclude by describing our initial implementation of data grid functionality.

2 Data Grid Design

The following four principles drive the design of our data grid architecture. These principles derive from the fact that data grid applications must frequently operate in wide area, multi-institutional, heterogeneous environments, in which we cannot typically assume spatial or temporal uniformity of behavior or policy.

Mechanism neutrality. The data grid architecture is designed to be as independent as possible of the low-level mechanisms used to store data, store metadata, transfer data, and so forth. This goal is achieved by defining data access, third-party data mover, catalog access, and other interfaces that encapsulate peculiarities of specific storage systems, catalogs, data transfer algorithms, and the like.

Policy neutrality. The data grid architecture is structured so that, as far as possible, design decisions with significant performance implications are exposed to the user, rather than encapsulated in "black box" implementations. Thus, while data movement and replica cataloging are provided as basic operations, replication policies are implemented via higher-level procedures, for which defaults are provided but that can easily be substituted with application-specific code.

Compatibility with Grid infrastructure. We attempt to overcome the difficulties of wide area, multi-institutional operation by exploiting underlying Grid infrastructure [10, 20, 15] (e.g., Globus [9]) that provides basic services such as authentication, resource management, and information. To this end, we structure the data grid architecture so that more specialized data grid tools are compatible with lower-level Grid mechanisms. This approach also simplifies the implementation of strategies that integrate, for example, storage and computation.

Uniformity of information infrastructure. As in the underlying Grid, uniform and convenient access to information about resource structure and state is emphasized as a means of enabling runtime adaptation to system conditions. In practice, this means that we use the same data model and interface

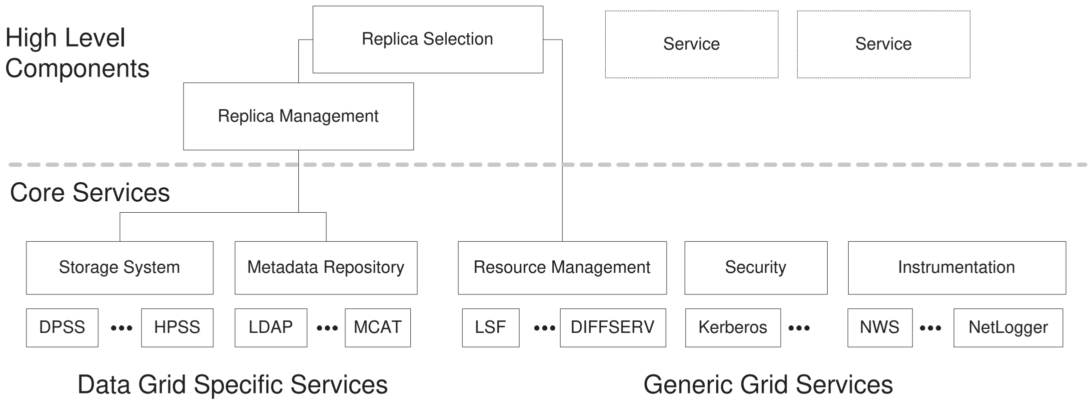

Figure 1: Major components and structure of the data grid architecture

to access the data grid's metadata and replica catalogs as are used in the underlying Grid information infrastructure.

These four principles lead us to develop a layered architecture (Figure 1), in which the lowest layers provide high-performance access to an orthogonal set of basic mechanisms, but do not enforce specific usage policies. For example, we define high-speed data movement functions with rich error interfaces as a low-level mechanism, but do not encode within these functions how to respond to storage system failure. Rather, such policies are implemented in higher layers of the architecture, which build on the mechanisms provided by the basic components.

This approach is motivated by the observation that achieving high performance in specific applications often requires that an implementation exploit domain-specific or application-specific knowledge. In data grids, as in other Grid systems, this focus on simple, policy-independent mechanisms will encourage and enable broad deployment without limiting the range of applications that can be implemented. By limiting application specific behaviors to the upper layers of the architecture, we can promote reuse of the basic mechanisms while delivering high-performance and specialized capabilities to the end user and application.

3 Core Data Grid Services

We now turn our attention to the basic services required in a data grid architecture. We focus in particular on two services that we view as fundamental: data access and metadata access. The data access service provides mechanisms for accessing, managing, and initiating third-party transfers of data stored in storage systems. The metadata access service provides mechanisms for accessing and managing information about data stored in storage systems. This explicit distinction between storage and metadata is worth discussing briefly. In some circumstances, for example when data is being stored in a database system, there are advantages to combining metadata and storage into the same abstraction. However, we believe that keeping these concepts separate at the architectural level enhances flexibility in storage system implementation while having minimal impact on the implementation of behaviors that combine metadata access with storage access.

3.1 Storage Systems and Data Access

In a Grid environment, data may be stored in different locations and on different devices with different characteristics. As we discussed above, mechanism neutrality implies that applications should not need to be aware of the specific low-level mechanisms required to access data at a particular location. Instead, applications should be presented with a uniform view of data and with uniform mechanisms for accessing that data. These requirements are met by the storage system abstraction and our prototype API for data access.

3.1.1 Data Abstraction: Storage Systems

We introduce as a basic data grid component the *storage system*, which provides functions for creating, destroying, reading, writing, and manipulating *file instances*. A file instance is the basic unit of information in a storage system. It consists of a named, uninterpreted sequence of bytes. The use of the term file instance is not intended to imply that data must reside in a conventional file system. A file instance may actually reside in a file system, database or other storage system. For example, a data grid implementation might use a system such as the Storage Resource Broker (SRB) to access data stored within a database management system.

Note that our definition of a storage system is a logical one: a storage system can be implemented by any storage technology that can support the required access functions. Implementations that target Unix file systems, HTTP servers, hierarchical storage systems such as HPSS, and network caches such as the Distributed Parallel Storage System (DPSS) are certainly envisioned. In fact, a storage system need not map directly to a single low-level storage device. For example, a distributed file system that manages files distributed over multiple storage devices or even sites can serve as a storage system, as can an SRB system that serves requests by mapping to multiple storage systems of different types.

A storage system will associate a set of properties, including a name and attributes such as size and access restrictions, with each of the file instances that it contains. The name assigned to a file instance by a particular storage system is arbitrary and has meaning only to that storage system. In many storage systems, a name will be a hierarchical directory path. In other systems such as SRB, it may be a set of application metadata that the storage system maps internally to a physical file instance.

3.1.2 Data Access

The behavior of a storage system as seen by a data grid user is defined by an API that describes the possible operations on storage systems and file instances. Our understanding of the functionality required in this API is still evolving, but it certainly should include support for remote requests to read and/or write named file instances and to determine file instance attributes such as size. In addition, to support optimized implementations of replica management services (discussed below) we require a third party transfer operation to transfer the entire contents of a file instance from one storage system to another.

While the basic storage system functions are relatively simple, various data grid considerations can increase the complexity of an implementation. For example, storage system access functions must be integrated with the security environment of each site to which remote access is required [12]. Robust performance within higher-level functions requires reservation capabilities within storage systems and network interfaces [11]. Applications should be able to provide storage systems with hints concerning access patterns, network performance, and so forth that the storage system can use to optimize its behavior. Similarly, storage systems should be capable of characterizing and monitoring their own performance; this information, when made available to storage system clients, allows them to optimize their behavior. Finally, data movement functions must be able to detect and report errors. While it may be possible to recover from some errors within the storage system, other errors may need to reported back to the remote application that initiated the movement.

3.2 The Metadata Service

The second set of basic machinery that we require is concerned with the management of information about the data grid itself, including information about file instances, the contents of file instances, and the various storage systems contained in the data grid. We refer to this information as *metadata*. The *metadata service* provides a means for publishing and accessing this metadata.

Various types of metadata can be distinguished. It has become common practice to associate with scientific datasets metadata that describes the contents and structure of that data. The metadata may describe the information content represented by the file, the circumstances under which the data was obtained, and/or other information useful to applications that process the data. We refer to this as *application metadata*. Such metadata can be viewed as defining the logical structure or semantics that should apply to the uninterpreted bytes that make up a file instance or a set of file instances. A second type of metadata is used to manage replication of data objects; this *replica metadata* includes information for mapping file instances to particular storage system locations. Finally, *system configuration metadata* describes the fabric of the data grid itself: for example, network connectivity and details about storage systems, such as their capacity and usage policy.

The metadata service provides a uniform means for naming, publishing, and accessing these different types of metadata. Each type of metadata has its own characteristics in terms of frequency and mechanism of update and its logical relationship to other grid components and data items. Interesting data management applications are likely to use several kinds of metadata. Although we have referred to several different types of metadata, we propose that a single interface be used for accessing all these types.

Applications identify files of interest by posing queries to a metadata service that includes a metadata *repository* or *catalog*. Each query specifies the characteristics of the desired data. The metadata repository associates such characteristics with *logical files*, which are entities with globally unique names that may have one or more physical instances. Once the metadata service has identified logical files with the desired attributes, the replica manager (described in Section 4.1) uses replica metadata to locate the physical file instance to be accessed.

The difficulty of specifying a general structure for all metadata is apparent when one considers the variety of approaches used to describe application metadata. Some applications build a metadata repository from a specified list of file instances based on data stored in a self-describing format (e.g., NetCDF, HDF). High energy physics applications are successfully using a specialized indexing structure. The Digital Library community is developing sets of metadata for different fields (e.g., [1]). Other user communities are pursuing the use of eXtended Markup Language (XML) [4] to represent application metadata.

The situation is further complicated when one considers the additional requirements imposed by large-scale data grid environments. Besides providing a means of integrating the different approaches to metadata storage and representation, the service must operate efficiently in a distributed environment. It must be scalable, supporting metadata about many entities being contributed by myriad information sources located in a large number of organizations. The service must be robust in the face of failure, and organizations should be able to assert local control over their information.

Analysis of these requirements leads us to conclude that the metadata service must be structured as a hierarchical and distributed system. This approach allows us to achieve scalability, avoid any single point of failure, and facilitate local control over data. Distribution does complicate efficient retrieval, but this difficulty can be overcome by having data organization exploit the hierarchical nature of the metadata service.

This analysis leads us to propose that the metadata service be treated as a distributed directory service, such as that provided by the Lightweight Directory Access Protocol (LDAP) [23]. Such systems support a hierarchal naming structure and rich data models and are designed to enable distribution. Mechanisms defined by LDAP include a means for naming objects, a data model based on named collections of attributes, and a protocol for performing attribute-based searching and writing of data elements. We have had extensive experience in using distributed directory services to represent general Grid metadata [8], and we believe that they will be well suited to the metadata requirements of data grids as well.

The directory hierarchy associated with LDAP provides a structure for organizing, replicating, and distributing catalog information. However, the directory service does not specify how the data is stored or where it is stored. Queries may be referred between servers, and the LDAP protocol can be placed in front of a wide range of alternative information and metadata services. This capability can provide a mechanism for the data grid to support a wide variety of approaches to providing application metadata, while retaining a consistent overall approach to accessing that metadata.

3.3 Other Basic Services

The data grid architecture also assumes the existence of a number of other basic services, including the following:

- An authorization and authentication infrastructure that supports multi-institutional operation. The public key-based Grid Security Infrastructure (GSI) [12] meets our requirements.

- Resource reservation and co-allocation mechanisms for both storage systems and other resources such as networks, to support the end-to-end performance guarantees required for predictable transfers (e.g., [11]).

- Performance measurements and estimation techniques for key resources involved in data grid operation, including storage systems, networks, and computers (e.g., the Network Weather Service [25]).

- Instrumentation services that enable the end-to-end instrumentation of storage transfers and other operations (e.g., NetLogger [22], Pablo [18], and Paradyn [13]).

4 Higher-Level Data Grid Components

A potentially unlimited number of components can exist in the upper layer of the data grid architecture. Consequently, we will limit our discussion to two representative components: replica management and replica selection.

4.1 Replica Management

The role of a *Replica Manager* is to create (or delete) copies of file instances, or *replicas*, within specified storage systems. Note that a replica is a user-asserted correspondence between two physical files. Typically, replicas specified in the catalog will be byte-for-byte copies of one another, but this is not required by the Replica Manager. In the remainder of this section, we use the terms *replica* and *file instance* interchangeably.

Often, a replica is created because the new storage location offers better performance or availability for accesses to or from a particular location. A replica might be deleted because storage space is required for another purpose.

In this discussion, we assume that replicated files are read only; we are not concerned with issues of file update and coherency. Thus, replicas are primarily useful for access to "published" data sets. While this read only model is sufficient for many uses of scientific data sets, we intend to investigate support for modifying the contents of file instances in the future.

The Replica Manager maintains a *repository* or *catalog*. Entries in the catalog correspond to logical files and possibly collections of logical files. Associated with each logical file or collection are one or more replicas or file instances. The replica catalog contains mapping information from a logical file or collection to one or more physical instances of the object(s). Figure 2 in Section 5.3 shows an example of a replica catalog for a data visualization application.

A data grid may (and indeed typically will) contain multiple replica catalogs. For example, a community of researchers interested in a particular research topic might maintain a replica catalog for a collection of data sets of mutual interest. It is possible to create hierarchies of replica catalogs to impose a directory-like structure on related logical collections. In addition, the replica manager can perform access control on entire catalogs as well as on individual logical files.

Note that the existence of a replica manager does not determine when or where replicas are created, or which replicas are to be used by an application, nor does it even require that every file instance be entered into a replica catalog. In keeping policy out of the definition of the replica manager, we maximize the types of situations in which the replica manager will be useful. For example, a file instance that is not entered into the catalog may be considered to be in a local "cache" and available for local use only. Designing this as a policy rather than coupling file movement with catalog registration in a single atomic operation explicitly acknowledges that there may be good, user-defined reasons for satisfying application needs by using files that are not registered in a replica catalog.

4.2 Replica Selection and Data Filtering

Another high-level service provided in the data grid is replica selection. Replica selection is the process of choosing a replica that will provide an application with data access characteristics that optimize a desired performance criterion, such as absolute performance (i.e. speed), cost, or security. The selected file instance may be local or accessed remotely. Alternatively the selection process may initiate the creation of a new replica whose performance will be superior to the existing ones.

Where replicas are to be selected based on access time, Grid information services can provide information about network performance, and perhaps the ability to reserve network bandwidth, while the metadata repository can provide information about the size of the file. Based on this, the selector can rank all of the existing replicas to determine which one will yield the fastest data access time. Alternatively, the selector can consult the same information sources to determine whether there is a storage system that would result in better performance if a replica was created on it.

A more general selection service may consider access to subsets of a file instance. Scientific experiments often produce large files containing data for many variables, time steps, or events, and some application processing may require only a subset of this data. In this case, the selection function may provide an application with a file instance that contains only the needed subset of the data found in the original file instance. This can reduce the amount of data that must be accessed or moved.

This type of replica management has been implemented in other data-management systems. For example, STACS is often capable of satisfying requests from High Energy Physics applications by extracting a subset of data from a file instance. It does this using a complex indexing scheme that represents application metadata for the events contained within the file. Other mechanisms for providing similar function may be built on application metadata obtainable from self-describing file formats such as NetCDF or HDF.

Providing this capability requires the ability to invoke filtering or extraction programs that understand the structure of the file and produce the required subset of data. This subset becomes a file instance with its own metadata and physical characteristics, which are provided to the replica manager. Replication policies determine whether this subset is recognized as a new logical file (with an entry in the metadata repository and a file instance recorded in the replica catalog), or whether the file should be known only locally, to the selection manager.

Data selection with subsetting may exploit Grid-enabled servers, whose capabilities involve common operations such as reformatting data, extracting a subset, converting data for storage in a different type of system, or transferring data directly to another storage system in the Grid. The utility of this approach has been demonstrated as part of the Active Data Repository [7]. The subsetting function could also exploit the more general capabilities of a computational Grid such as that provided by Globus. This offers the ability to support arbitrary extraction and processing operations on files as part of a data management activity.

5 Implementation Experiences

In this section, we describe our initial design of catalogs for metadata and replica management. We used these catalogs at the Supercomputing '99 conference to support two application demonstrations: climate modeling and data visualization. In this section, we describe our experiences and the lessons learned from our prototype implementation.

5.1 An LDAP Implementation

We used the Lightweight Distributed Access Protocol (LDAP) to construct our prototype catalogs. Information in an LDAP catalog is organized in a tree structure known as a Directory Information Tree (DIT). Information in a directory or catalog resides in nodes that are placed in a hierarchical relationship.

For the climate modeling application, scientists constructed an LDAP catalog that included both metadata and physical location information for the files in the data collection. For the data visualization application, we constructed separate metadata and replica catalogs for managing the data. These catalogs are described in detail below.

5.2 Climate Modeling Application

First, we describe the LDAP catalog used in the climate modeling application. The DIT for the climate application consisted of a root node, a node for each of four collections, and a node for each logical file in the collection. The application metadata was encoded in XML and stored in the node for the collection. A query for a particular time-step or variable could be mapped to a specific logical file. The information for the logical file contained the storage system and path name where the data was located plus a template for constructing the instance file name. This information was used to construct a URL for the data. The URL was passed to a file transfer interface which moved the data into a local cache for processing.

For this prototype, the user was presented with a list of several structurally identical catalogs, each corresponding to a single replica of the data set. The user selected one catalog to be used to locate data. This prototype demonstrated the ability to map a query based on application metadata into a URL using an LDAP catalog, and to move the data using a general purpose transfer API.

Subsequent designs will provide each catalog with location information for multiple replicas. Tools will be developed to assist in the selection of data source based on user specified criteria (e.g. best performance).

5.3 Data Visualization Application

A second catalog was built to locate files for a distance visualization application, in which a desktop visualization client streamed data from remote storage systems. Each file corresponds to one timestep in a series. For example, output from an astrophysical simulation developed at the University of Chicago "FLASH" center includes three data sets. Each of these data sets includes up to 486 timestep files. In addition, each timestep file can be represented in different resolutions as well as different data layouts, such as big-endian or little-endian. The result is that there are several thousand files in the Flash dataset. In the replica catalog used in these initial experiments, each file is represented by several objects in the LDAP catalogs. Figure 2 shows example objects in a replica catalog hierarchy for this distance visualization application.

Based on our experience with this prototype, we concluded that the present implementation scales poorly for datasets like Flash. In particular, when data sets include hundreds of files in multiple data layouts and file formats, the current Replica Catalog implementation generates thousands of objects. This large number of objects slowed directory searches and increased complexity by requiring a distributed catalog implementation.

We have made several changes to the design as a result of this experience. To improve our ability to locate sets of files efficiently, we have increased support for specifying *collections* of logical files. We associate location information with each collection, making it possible to avoid describing each file individually. A single lookup is sufficient to locate the storage system and path to the collection, and a template can be used to map the logical file name to the physical file name. This also greatly reduces the storage space required. Our new design will provide a more scalable solution in terms of both performance and catalog storage requirements.

5.4 Separation of Functionality

Another lesson learned from our prototype implementation is the importance of distinguishing the functions of metadata and replica management. Ideally, the functions of the Metadata and Replica catalogs should be as distinct as possible. Any information that describes the contents of data files and collections should be stored in the Metadata Catalog. Such information includes objects and attributes related to the structure of data sets and file formats. Any *mapping* information that describes how particular logical files or collections are stored or replicated on storage systems should be maintained in the Replica Catalog.

6 Status of the Data Grid Implementation

We have made progress on several fronts in our effort to identify the basic low-level services for a data grid architecture. We have a preliminary design of the data access API. This API provides a standard interface to storage systems, including create, delete, open, close, read and write operations on file instances. This interface also supports storage to storage transfers. For this prototype API design, we have implemented interfaces to several storage systems, including local file access, HTTP and FTP servers, and DPSS network disk caches.

As described in the last section, we have also implemented replica management and metadata services. These services use LDAP directories to store attribute information about file instances,

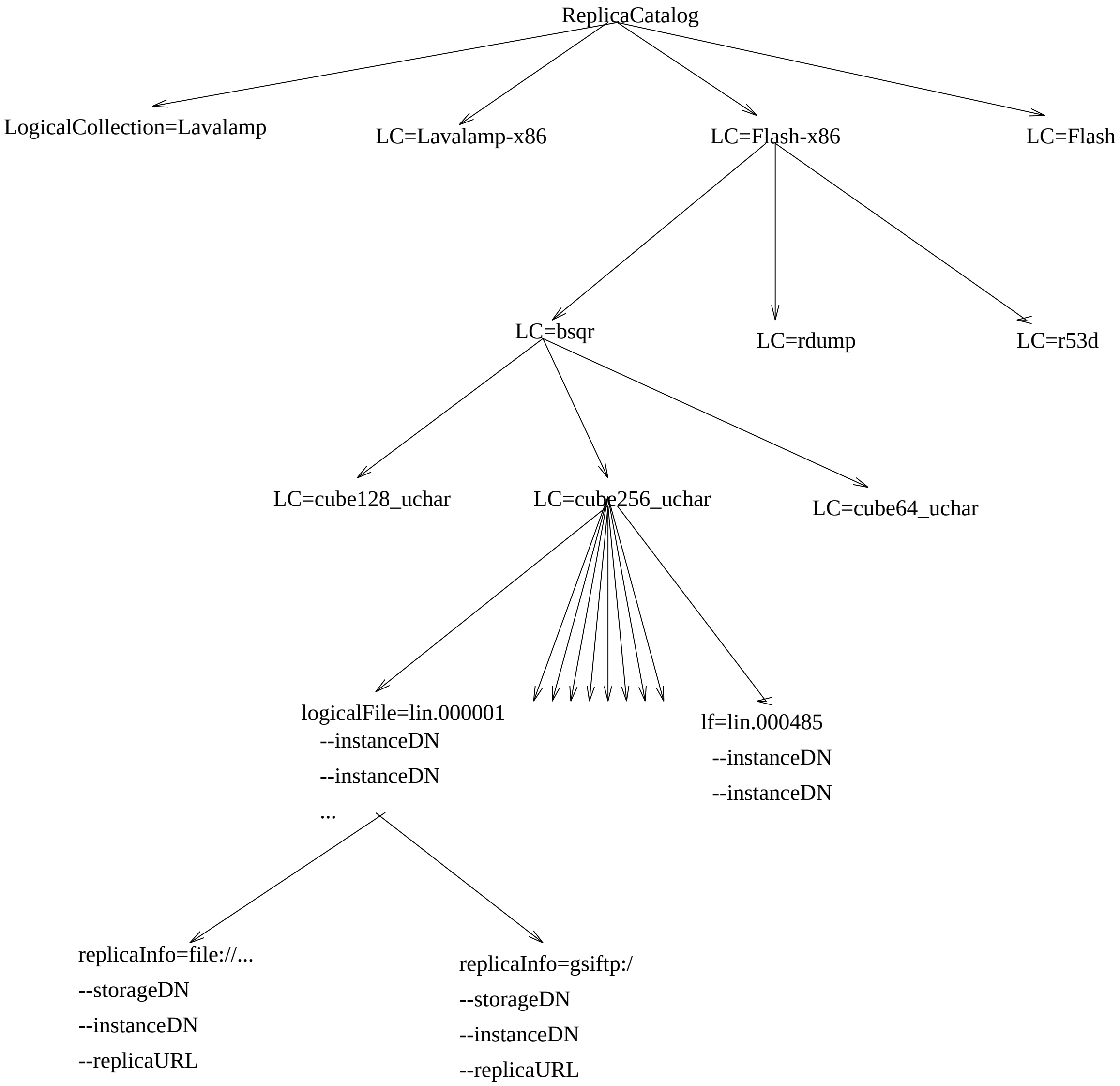

Figure 2: The structure of a replica catalog.

storage systems, logical files, and replica catalogs. Using these attributes, we can query the metadata and replica catalogs as well as the Grid information system to find the replicas associated with a logical file, estimate their performance, and select among replicas according to particular performance metrics. We described our experiences using these catalogs for two applications: climate modeling and data visualization.

This work represents the first steps in our effort to create an integrating architecture for data-intensive petabyte-scale application domains. Performance studies of the data access API are under way, and we plan to further explore basic services such as instrumentation.

Acknowledgments

We gratefully acknowledge helpful discussions with Steve Fitzgerald, Bill Johnston, Reagan Moore, Richard Mount, Harvey Newman, Arie Shoshani, Brian Tierney and other participants in the DOE "Earth System Grid" and "Particle Physics Data Grid" projects. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] M. Baldonado, C. Chang, L. Gravano, and A. Paepcke. The Stanford digital library metadata architecture. *Intl J. Digital Libraries*, 1(2):108–121, 1997.

[2] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC storage resource broker. In *Proceedings of CASCON'98 Conference*. 1998.

[3] M. Beck and T. Moore. The Internet2 distributed storage infrastructure project: An architecture for internet content channels. *Computer Networking and ISDN Systems*, 30(22-23):2141–2148, 1998.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. The extensible markup language (xml) 1.0. W3C recomendation, World Wide Web Consortium, February 1998. See http://www.w3.org/TR/1998/REC-xml-19980210.

[5] L.T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. *Information Systems Special Issue on Scientific Databases*, 20(2):155–83, 1995.

[6] S. Cousins, H. Garcia-Molina, S. Hassan, S. Ketchpel, M. Roscheisen, and T. Winograd. Towards interoperability in digital libraries. *IEEE Computer*, 29(5), 1996.

[7] Renato Ferreira, Tahsin Kurc, Michael Beynon, Chialin Chang, Alan Sussman, and Joel Saltz. Object-relational queries into multidimensional databases with the active data repository. *International Journal of Supercomputer Applications*, 1999.

[8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.

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

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

[11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

[12] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.

[13] Jeffrey Hollingsworth and Bart Miller. Instrumentation and measurement. In *[10]*, pages 339–365.

[14] William Johnston. Realtime widely distributed instrumentation systems. In *[10]*, pages 75–103.

[15] William E. Johnston, Dennis Gannon, and Bill Nitzberg. Grids as production computing environments: The engineering aspects of NASA's Information Power Grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing.* IEEE Computer Society Press, 1999.

[16] M. Lesk. *Practical Digital Libraries: Books, Bytes, and Bucks.* Morgan Kaufmann Publishers, 1997.

[17] Reagan Moore, Chaitanya Baru, Richard Marciano, Arcot Rajasekar, and Michael Wan. Data-intensive computing. In *[10]*, pages 105–129.

[18] Daniel Reed and Randy Ribler. Performance analysis and visualization. In *[10]*, pages 367–393.

[19] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Storage management for high energy physics applications. In *Computing in High Energy Physics 1998 (CHEP 98).* 1998. http://www.lbl.gov/ arie/papers/proc-CHEP98.ps.

[20] R. Stevens, P. Woodward, T. DeFanti, and C. Catlett. From the I-WAY to the National Technology Grid. *Communications of the ACM*, 40(11):50–61, 1997.

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

[22] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing.* IEEE Computer Society Press, 1998.

[23] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, Internet Engineering Task Force, 1997.

[24] Richard W. Watson and Robert A. Coyne. The parallel I/O architecture of the High-Performance Storage System (HPSS). In *IEEE MSS Symposium*, 1995.

[25] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, Portland, Oregon, 1997. IEEE Press.