CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2008; **20**:2049–2060
Published online 14 March 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1304

# The XtreemFS architecture—a case for object-based file systems in Grids

Felix Hupfeld[1,*,†], Toni Cortes[2,3], Björn Kolbeck[1], Jan Stender[1], Erich Focht[4], Matthias Hess[4], Jesus Malo[2], Jonathan Marti[2] and Eugenio Cesario[5]

[1]*Zuse Institute Berlin (ZIB), Takustr. 7, 14195 Berlin, Germany*
[2]*Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[3]*Universitat Politecnica de Catalunya (UPC), Spain*
[4]*NEC HPC Europe GmbH, Stuttgart, Germany*
[5]*Institute High Performance Computing and Networks of the National Research Council of Italy (ICAR-CNR), Pisa, Italy*

**SUMMARY**

**In today's Grids, files are usually managed by Grid data management systems that are superimposed on existing file and storage systems. In this paper, we analyze this predominant approach and argue that object-based file systems can be an alternative when adapted to the characteristics of a Grid environment. We describe how we are solving the challenge of extending the object-based storage architecture for the Grid in XtreemFS, an object-based file system for federated infrastructures. Copyright © 2008 John Wiley & Sons, Ltd.**

*Received 31 October 2007; Accepted 5 December 2007*

KEY WORDS: object-based file system; grid data management; distributed file system architecture

## INTRODUCTION

The file abstraction is one of the success stories of system architecture, and the current computing world is unthinkable without file systems. Files are the technology of choice for any unstructured data and provide an efficient container for abstractions with more structure.

*Correspondence to: Felix Hupfeld, Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany.
†E-mail: hupfeld@zib.de

Contract/grant sponsor: EU IST XtreemOS project; contract/grant number: FP6-033576
Contract/grant sponsor: Spanish Ministry of Science and Technology; contract/grant number: TIN2004-07739-C02-01

However, conventional network file systems are ill-adapted to Grid-like environments. These file systems are usually heavily geared toward centralized installations in a single data center and lack reliable support for remote access over wide-area networks (WANs) across multiple organizations. For Grid data management, an approach was needed to compensate these weaknesses of installed local, network or distributed file systems. Instead of extending file system architectures with the necessary features, Grid data management systems are imposed on the existing file system. Remote access protocols such as GridFTP [1] make files and namespaces remotely accessible, and replica catalogs index the whereabouts of a file's copies.

While this approach of superimposing Grid data management on file systems has proven to be efficient and effective, it is not without drawbacks. Foremost, certain characteristics of the typical Grid data management architecture prevent these systems from performing as well as other, more integrated architectures. In addition, they cannot guarantee the consistency of file content across replicas and force applications and users to adapt their usage of the system accordingly.

In this paper, we claim that an object-based file system architecture [2] can be extended to be suitable for Grid environments and argue that it is a viable alternative architecture for file data management in Grids for many-use cases. To illustrate this argument, we demonstrate how we solve some of the relevant design issues in XtreemFS, a distributed object-based file system for federated wide-area infrastructures.

We continue this paper with a detailed study of Grid data management from a system architecture perspective with emphasis on its structural shortcomings. Then, we give an overview of distributed file system architectures and describe how it can be extended for federated wide-area environments. The following section presents the architecture of XtreemFS as an example of an object-based Grid file system. We conclude our paper with references to existing file systems and Grid data management solutions.

## COMMON CHARACTERISTICS OF GRID DATA MANAGEMENT

Grid data management systems provide their clients with access to file data that is stored at remote storage and file systems. They *index* files from storage resources and provide clients with a *unified interface* for accessing files. Typically the system relies on a daemon on the storage resource that mediates the remote access protocol with the heterogeneous local access interfaces.

A *replica catalog* stores the access locations for a file (sometimes called its physical file name) and abstracts the Grid file itself from its replicas at various storage resources. It is often integrated with a *metadata catalog*, which imposes a namespace on top of this Grid file abstraction and structures the file space for later retrieval of particular files. Common structuring methods are hierarchical name spaces (with logical file names, LFNs), database-like extended metadata attributes, and collections of files.

In order to access a file, the application has to download or replicate the file to its local file system first (Figure 1). When the file is on the application's local disk, the application can access the file normally. Similarly, newly created and modified files are uploaded to one of the storage resources or the new local file is registered with the system as a replica.

While this architecture has considerably simplified access to data that is kept at heterogeneous storage resources and integrates well with existing infrastructures, its architectural properties restrict the evolution of the basic approach.

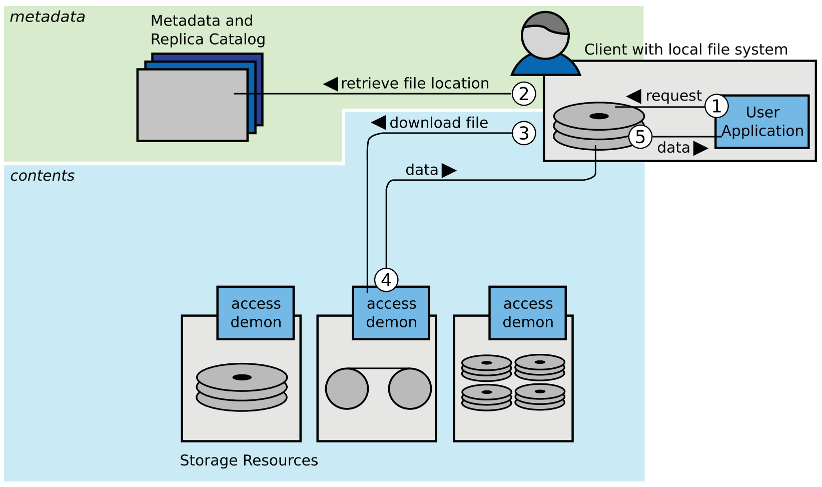

Figure 1. Components and their relationships of a typical Grid data management system. Typically, files are fetched from a storage resource before they can be accessed.

Typical Grid data management systems do not exercise control over data access beyond what is necessary for security purposes. The daemon that is running on the storage resources only mediates remote access to its files and has no notion about the system's state. It neither knows where other replicas for its files exist nor does it know in which state its files are. Also, Grid data management systems do not control client access to downloaded file copies.

Owing to this lack of control and information, Grid data management systems cannot make guarantees about the consistency of a file's replicas; thus applications are generally restricted to write-once usage patterns of files. Applications download read-only input files, process them to generate output files, and upload the latter to the data management system. In addition, storage resources are unaware of the state of their files and cannot oversee the replica creation process themselves. Thus, an extra service is usually needed for the reliable creation of replicas.

The architecture also has implications for performance. Typical Grid data management systems only operate on complete files, which increases the latency to first access of the file, because the client's access to the data has to be deferred until the file is fully recreated in the local file system. The data management system can neither automatically prioritize parts of the data that would be accessed first nor can it make partial replicas that skip downloading parts that are not required.

It can also be preferable to avoid downloading any data at all. Today, access to a remote file server can be much faster than the access to the application's local hard disk. With the fast network connection commonly found in today's computing systems, the application can exploit the aggregate bandwidth of many remote disks or profit from a large file cache in the server.

## DISTRIBUTED FILE SYSTEM ARCHITECTURES

Similar to a Grid data management system, a distributed file system makes use of multiple networked devices to store and manage its entrusted data. In contrast, however, distributed file systems implement a file system interface that mimics the semantics of a local file system as closely as possible and can therefore support applications without requiring changes to their code. By exploiting the inherent parallelism and redundancy of file system resources, a distributed file system can be made scalable, fast, and fault-tolerant.

When designing a distributed file system, a choice has to be made about how to distribute the elements of the file system. A file system consists of a hierarchy of directories (the namespace), metadata of individual files (such as size and ownership), and the file contents themselves. The latter can be further subdivided into storage units such as disk or file system blocks. Earlier distributed file systems such as AFS/DFS [3] chose to make the 'distribution cut' at the namespace level and provided a global namespace that subsumes the contents of individual storage servers, which held complete files.

Today's predominant architecture, the block-based file system (such as GPFS [4]) makes the cut within a file and distributes file contents as fixed-size blocks to storage devices. The storage devices of a block-based file system are agnostic of the association between blocks and files, and the devices only answer a client's requests for individual blocks. The mapping between files and block addresses, the list of free blocks, and file metadata are kept in a centralized file system server.

While block-based distributed file systems can exploit parallel access to block stores, the file system server must be queried frequently to retrieve block addresses, a potential performance bottleneck. The storage devices, however, only have to provide access to fixed-size blocks and are therefore simple enough to be implemented in hardware. Block-based file systems are usually bound to a specific block size and therefore lack the flexibility to support heterogeneous content that would profit from block sizes that are adapted to file size and access patterns.

Recently, technology trends have encouraged storage architects to revisit the interface to storage devices [2] and augment it with logic that was previously in the file server in order to alleviate the file server's load. This technology step resulted in object-based file systems [5] that distribute the contents of a file as *objects* to storage servers (called object storage devices, OSDs). The size of a particular file's objects is fixed, resembling blocks of a block-based file system, but the size of the objects can differ between files. Also, the objects keep the association with the file they constitute.

A prototypical object-based file system relies on a metadata server to maintain the file namespace and POSIX [6] metadata and to enforce access control policies (see Figure 2). In contrast to block-based file systems, OSDs manage data layout and free space themselves without the metadata server. When accessing a file, a file system client first contacts the metadata server to retrieve the locations of a file's objects along with an access capability that represents the client's access rights as a signed entity. All subsequent read and write operations on the file are performed directly on the respective OSDs. This basic protocol avoids the potential bottleneck of the metadata server and enables parallelization of all file I/O.

At first sight the object-based architecture bears many structural similarities to the typical architecture of Grid data management systems. Both architectures typically separate file metadata from file data and expose the notion of a file to their storage resources. In addition, however, object-based file systems mediate the operations of their clients and can exercise full control over the operations where necessary. Object-based file systems also treat their storage resources as pure storage devices for file content and do not offer additional functionality that could restrict the architecture.

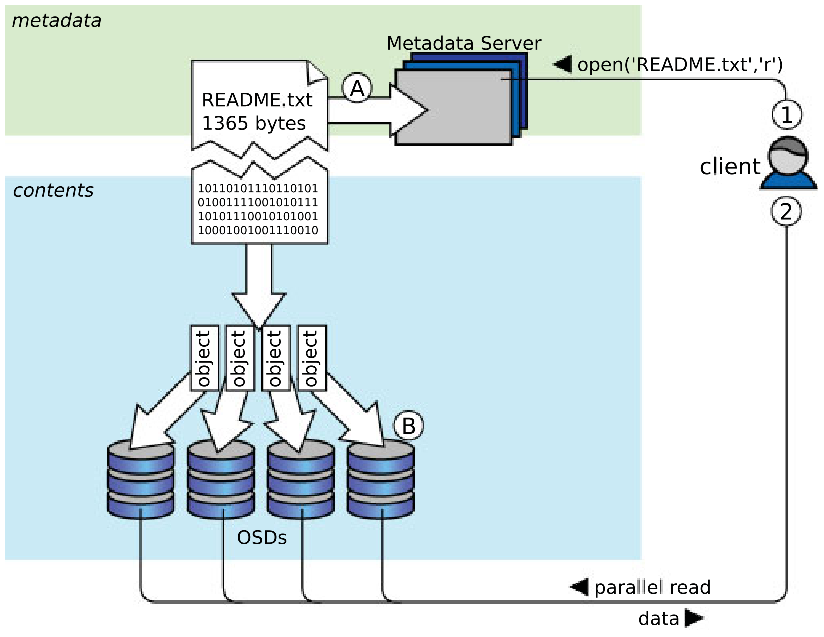

Figure 2. Components and their relationships in an object-based file system. The metadata server authenticates and authorizes the client on open() and issues a ticket. Subsequently, all I/O operations are directly performed at the object storage devices.

## EXTENDING OBJECT-BASED STORAGE FOR THE GRID

Existing object-based file systems are designed as parallel file systems for clusters or enterprise file systems with centralized IT infrastructures. They have all the amenities of a file system and can exploit the resources of today's hardware in an economic manner. A typical installation includes a rack of metadata servers and many OSDs with a number of hard disks. In this homogeneous environment, all OSDs are equal from a latency and bandwidth perspective, and objects can be assigned to disks in a deterministic way. The predominant source of failure is the hard disks and disk failures are handled by introducing redundancy via RAID. Users of these file systems are part of the locally controlled administration domain. All these assumptions about a local homogeneous and controlled environment cannot readily be transferred to the dynamic and heterogeneous environment of Grids. The task of adapting the object-based approach to Grid environments opens up research challenges in system architecture, distributed algorithms, and administration and security infrastructures.

In contrast to the common single-site setup of file systems, Grid installations typically encompass multiple sites, organizations, and administration domains. In order to be useful in such an environment, the file system must have a structure that supports a multi-organization view of administration, management, and security. These multi-domain virtual organizations (VOs) are the basic unit of authentication and authorization mechanisms in a Grid environment. VO systems have to be supported by the file system as well, although they might also need to be adapted to fit the well-defined semantics of file access.

In a Grid that spans multiple organizations and sites, it is essential for the file system to be prepared at any time for the case that parts of its installation join, leave, or fail. It needs a *federated* structure, where no part is preferred to the others and partial absence due to failure or downtimes can be tolerated. For instance, a local file system installation should continue to work if the network connection to the Internet fails. The file system should also remain operational if the site decides to leave the federation for good. In turn, the overall file system operation should not be affected if parts of the installation fail, leave, or become temporarily disconnected. This federation may be partially achieved by provisions in the system structure but might also need support from replication mechanisms.

The primary targets for replication mechanisms are the metadata servers and the OSDs. Apart from supporting federated installations, replication can improve availability within a site as well as general access performance. Also, it is often better to replicate data closer to the consumers so that file system clients can enjoy shorter network latencies and higher bandwidth. Generally, the replication mechanisms for file data and metadata should not place functional restrictions on the placement of their replicas so that replicas can be created where and when they are most needed. However, non-functional restrictions such as security must also be considered.

The major challenge with replicated data is to maintain its consistency. When multiple replicas are changed concurrently, the system must ensure that the file replicas are consistent and that the clients see the expected semantics of a POSIX interface. These guarantees must not be weakened by any failures in the environment, including temporary network failures and network partitions. Replication algorithms can be categorized into mechanisms that elect a primary to control the consistency [7] and mechanisms that implement a replicated state machine [8–11] that consistently executes operations on all replicas. These replication algorithms have to be checked for the suitability for replicating file data and metadata.

Once these challenges are addressed, users can benefit from all the advantages of a general file system. Because all operations of an application go through the controlled file system interface, the application is decoupled from any internal aspects of the system. The file system can see and influence any operation of the application and act accordingly to provide the client with the best possible performance. In turn, the application can simply mount and access the file system's data transparently.

## THE ARCHITECTURE OF XtreemFS

The XtreemFS file system architecture addresses many of the problems described above. XtreemFS has been specifically designed to support file replication and federated setups in Grid environments. XtreemFS organizes its file systems in the file system *volumes*, each of which represents a mountable file system with a separate directory structure and configuration. A volume's files and directories share certain default policies for replication and access.

Similar to other object-based file systems, XtreemFS consists of clients, OSDs, and metadata servers. The metadata servers of XtreemFS also track the locations of file replicas and are called

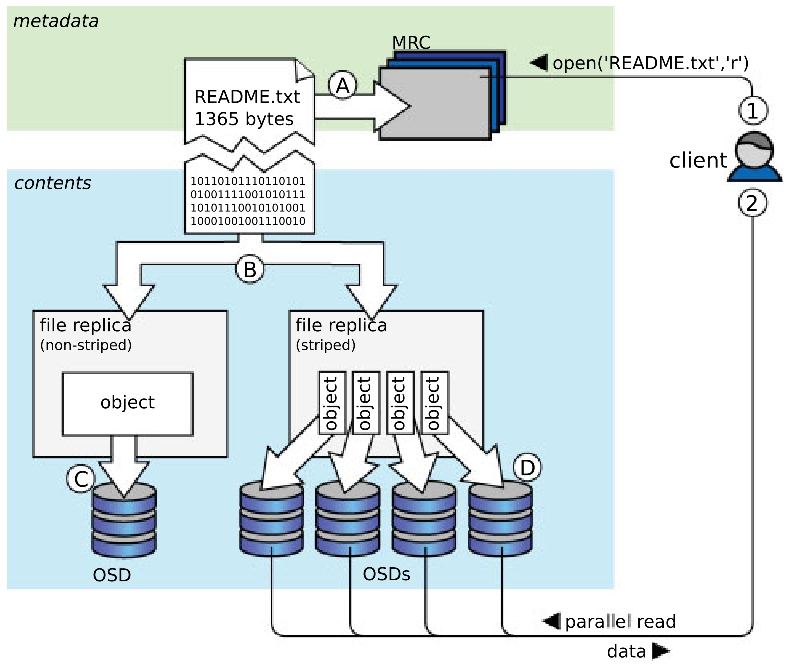

Figure 3. Components and their relationships in XtreemFS. While supporting striping over a group of OSDs, XtreemFS allows files to be replicated to different locations.

metadata and replica catalogs (MRCs, see Figure 3). In addition to these three main components, a directory service acts as a registry for servers and file system volumes.

## XtreemFS metadata servers

An XtreemFS MRC is able to host multiple file system volumes and acts as a metadata server for clients that mount and access one of the volumes. Clients are not tied to a particular MRC, and given proper access rights, a client can mount volumes from any MRC on the Grid. To authenticate, a client presents the user's credentials (usually X.509 certificates) obtained from one of the common VO infrastructures. After successful authentication, the operations of a client are subject to pluggable access policies. These access policies can implement normal file system policies such as Unix user/group rights or full POSIX ACLs, although the MRC can also support access policies that are better suited to a particular organization structure.

An MRC stores all of the information about its hosted volumes in an embedded database, which has a mechanism to replicate its data across hosts. Thus, volumes can be hosted by multiple MRCs, which increases volume availability and access performance. We have chosen to implement the replication mechanism on the operation level instead of replicating data on the storage level because it allows us to support online access to all volume replicas. With replication on the storage level, only a fail-over mechanism would be possible. We have already implemented a fault-tolerant lease-based master–slave replication algorithm, and we are currently working on a replication algorithm that coordinates full read–write access to all volume replicas.

Apart from standard file metadata, the MRC also keeps a list of replica locations for each file, along with striping information for the file, such as the striping pattern and stride on RAID devices. In a federated environment, replication policies restrict the range of OSDs to which an MRC will replicate files or the set of MRCs from which an OSD will accept replicas. Apart from these policy restrictions, our design allows files to be replicated to any OSD.

### XtreemFS directory service

The directory service of XtreemFS connects all of the components of the file system. The directory service allows clients to locate the MRCs that are responsible for a certain volume, registers all servers and allows them to exchange dynamic information, and acts as a shared store between MRCs and OSDs.

When a client first accesses a file, the client is authenticated and authorized by the MRC that hosts the file's volume. The directory service supports this process by storing a shared secret between a protected area that allows MRCs and OSDs to generate and verify the signature on the capability token submitted by the client. These capabilities also carry an absolute timeout that acts as a built-in revocation mechanism. Apart from this use case, absolute timeouts are part of several algorithms in other components. For all these distributed components, the directory service acts as central time source so that timeouts can be interpreted in a consistent manner.

Generally, we have tried to avoid introducing direct communication dependencies between MRCs and OSDs. This can decrease the load on MRCs and also helps when private subnets make it difficult to set up direct connections between servers. Where shared information between MRCs and OSDs is required, the directory service acts as a proxy. For example, the directory service retains dynamic information such as load and capacity information for OSDs, which MRCs can periodically query.

The XtreemFS directory server is currently centralized and does not support a federated fault-tolerant setup. To fully support the federation aspect, the directory server would need to be replicated or follow a hierarchical design that is able to tolerate failure of parts of its hierarchy.

### XtreemFS OSDs

XtreemFS OSDs provide clients with high-performance access to the objects they store. OSDs are responsible for the actual storage layout on their attached disks and they maintain a cache of recently accessed objects. Apart from these tasks, the majority of OSD logic is dedicated to ensuring the consistency of a file's replicas.

Consistency maintenance is solely the responsibility of the OSDs that store a replica of a particular file. When a client accesses data on an OSD for the first time, its request includes information for the OSD about the locations of other replicas of the file, which allows the OSD to coordinate operations in a peer-to-peer manner. No central component is involved in this process, which makes it scalable and fault tolerant. In order to coordinate operations on file data, OSDs negotiate leases [12] that allow their holder to define the latest version of the particular data without further communication efforts. Also, each OSD maintains a version number for each file object it stores, which allows the OSD to determine whether its local data are current. With the help of this lease coordination algorithm, XtreemFS can guarantee POSIX semantics even in the presence of concurrent accesses to the replicas.

The version number that an OSD stores along with each object allows the OSD to keep outdated objects or even to not store the object locally at all. Thus, an OSD can keep partial or outdated replicas under certain circumstances without compromising consistency. By exploiting this mechanism, XtreemFS can choose to update only those replicas that are currently accessed by clients and let the other replicas fall behind. As soon as a client asks an OSD for a part of a file that it has not previously accessed, the OSD checks the validity of the locally available data and retrieves the newest version if necessary.

The consistency of the replicas of a particular file is controlled by policies. These policies dictate how many replicas an OSD forwards changes to before acknowledging the write operation of the client. The user can, for example, choose a strict policy that always keeps at least three replicas up-to-date at different sites or select a looser policy that updates other replicas lazily or on demand.

The awareness of OSDs about replicas also allows us to logically create new replicas very quickly and reliably. From an external perspective, a new replica is created as soon as the added OSD has learned about that the new replica. The OSD then marks the versions of the replica's objects as obsolete. Subsequently, the replica is physically created using the described consistency protocols, either on demand by a client's accesses or automatically when a policy instructs the replica to do so. Replicas are therefore always created reliably as a decentralized interaction between the OSDs. There is no need for extra services to initiate, control, or monitor the transfer of the data.

In order to be able to create replicas in the presence of failures of some of the OSDs and to be able to remove unreachable replicas, we have designed a replica set coordination protocol that integrates with the lease coordination protocol. The replica set protocols ensure that even in the worst failure case, replicated data can never become inconsistent, while still allowing replicas to be added or removed in many failure scenarios.

Because it involves a distributed consensus process that is inherently expensive, the replica lease coordination process does not scale well. When too many OSDs per file are involved, the necessary communication increases excessively. Fortunately, a moderate number of replicas are sufficient for most purposes. If a large number of replicas are required, XtreemFS can switch the file to a read-only mode and allow an unlimited number of read-only file replicas, which fits many common Grid data management scenarios.

## Common Grid use cases and file systems

XtreemFS's object-based file system enables many features that current Grid applications can take advantage of. The objective of this section is to present some use cases that would clearly benefit from the ability to access data through a file system.

We first consider scientific applications that access large files routinely. For example, the large hadron collider at CERN generates large files that are read (but not written) from many nodes in a Grid. The current way of using these huge files is to copy them to the node where the file will be processed and to remove the file some time after processing has finished (traditional stage-in).

Being a file system, XtreemFS is able to control where and how its clients access a file's replicas. If there is a replica close enough to the client, applications can access this replica directly in a transparent way. Also, if only parts of a large file are accessed, the file system can replicate these parts and avoid transferring the whole file. A file system can also reduce the latency to first access considerably by creating the replica in the background and redirecting the client to local data as they become available.

Database management systems (DBMS) would also benefit from automatic and partial replica creation in our file system. Given that databases are normally huge files that are frequently accessed only partially, a file system could replicate only the parts of the database that are being used, reducing the amount of replicated data and the time and resources consumed for creating the replica.

As XtreemFS allows replicas to be physically desynchronized, we can allow MPI applications to work on different replicas of the same file when the different processes of the application are very far apart. XtreemFS assigns these processes a nearby replica for writing. Depending on the replication policy, the written data may be lazily synchronized to other replicas over time or later on demand. In either case, XtreemFS guarantees that replicas appear consistent for any subsequent read operation. Current Grid data management systems are not able to support this kind of access pattern.

## RELATED WORK

While many Grid projects have developed custom solutions for their data management problems, a couple of software products for Grid data management have emerged and been widely adopted. Each of these systems is similar in architectural spirit to the general scheme we presented in the section but targets different applications domains.

Among the most prominent systems are the data management services of the Globus toolkit [13,14]. Globus users install GridFTP [1] daemons on their storage resources that export the host file system and enable it for remote access. A replica catalog (RLS) indexes these storage locations and implements additional naming facilities on top of it. Globus integrates well with the local structure and access rights management of the local system and is able to preserve file naming across replicas. Its GridFTP framework [1] allows high-performance parallel transfer of files between the resources.

SDSC's storage resource broker (SRB) [15] provides a complete data management solution. Unlike Globus, it does not focus on preserving a storage host's file system name space and stores its files under their file identifiers. It also provides support for federated installations in multiple sites.

dCache [16] makes a strong emphasis on archiving data with the help of tertiary storage systems such as tape robots. It can act as a front end to these resources and allows clients to access files via a file system-like interface over NFS.

The AMGA metadata catalog [17] of the EGEE project provides support for fine-grained access control on extended metadata and features powerful replication capabilities. It can partially and fully replicate metadata on multiple sites and supports federation of metadata by a master–slave-like replication semantics.

Grid datafarm (Gfarm) is a system for managing files in Grids which follows a file system-like approach. It is specialized for workloads in which applications create a large amount of data that are consumed by other applications later on. While the first version of Gfarm allows files to be written only once [18], Gfarm v2 [19] aims to offer full file system functionality. There is also an effort under way to standardize Grid file system at the Grid File System Working Group (GFS-WG) of the open grid forum (OGF).

While distributed file systems are recently shifting from block-based architectures to object-based storage, most file systems are still based on the block-based approach. Traditional block-based parallel file systems such as RedHat's global file system (GFS) and Oracle's cluster file system (OCFS2) rely on the capability of every computer to access the block devices over a storage area network (SAN) and parallelize the file system code to enable coordinated concurrent access to the storage devices. The direct access to the block devices is usually realized using fiber channel or iSCSI. GPFS [4] is a proprietary representative of the block-based file systems which has off-loaded a part of the block management from the clients to network storage device (NSD) servers and shows improved scaling behavior.

Existing object-based file systems are designed for single-site installations and for high-performance parallel access to the storage resources. Commercial (Panasas' ActiveScale, [20]), open-source (Lustre [21]), and research systems (Ceph [20]) are available.

## CONCLUSION

In this paper, we have analyzed where the typical architecture of Grid data management systems has deficiencies and argued that a Grid-aware adaptation of the object-based file system architecture is able to address them. As an example, we have shown how some of the challenges of adapting object-based storage to wide-area federated infrastructures are solved in XtreemFS.

Object-based file systems for Grids will not be able to support the full range of application domains of Grid data management systems. For example, it could be difficult to integrate them with existing legacy installations that depend on interfacing with a Grid data management system. Nevertheless, we think that there are enough use cases, especially in new Grid installations, where applications could considerably benefit from running on a real file system that is designed for the environment.

### REFERENCES

1. Allcock W, Bresnahan J, Kettimuthu R, Link M. The Globus striped GridFTP framework and server. *SC '05*: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society: Washington, DC, U.S.A., 2005; 54.
2. Mesnier M, Ganger G, Riedel E. Object-based storage. *IEEE Communications Magazine* 2003; **8**:84–90.
3. Satyanarayanan M. Scalable, secure, and highly available distributed file access. *Computer* 1990; **23**(5):9–18, 20–21.
4. Schmuck F, Haskin R. GPFS: A shared-disk file system for large computing clusters. *FAST '02*: *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association: Berkeley, CA, U.S.A., 2002; 19.
5. Factor M, Meth K, Naor D, Rodeh O, Satran J. Object storage: The future building block for storage systems. *Local to Global Data Interoperability—Challenges and Technologies*, Sardinia, Italy. IEEE Computer Society: Washington, DC, 2005.
6. The Open Group. The Single Unix Specification, Version 3.
7. Jiménez-Peris R, Patiño-Martínez M, Alonso G, Kemme B. Are quorums an alternative for data replication? *ACM Transactions on Database Systems* 2003; **28**(3):257–294.
8. Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 1978; **21**(7):558–565.
9. Burrows M. Chubby distributed lock service. *Proceedings of the 7th Symposium on Operating System Design and Implementation*, *OSDI'06*, Seattle, WA, 2006.
10. Lorch JR, Adya A, Bolosky WJ, Chaiken R, Douceur JR, Howell J. The SMART way to migrate replicated stateful services. *EuroSys '06*: *Proceedings of the 2006 EuroSys Conference*. ACM Press: New York, NY, U.S.A., 2006; 103–115.

11. Chandra TD, Griesemer R, Redstone J. Paxos made live: An engineering perspective. *PODC '07*: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press: New York, NY, U.S.A., 2007; 398–407.
12. Lampson BW. How to build a highly available system using consensus. *Tenth International Workshop on Distributed Algorithms* (*WDAG 96*), vol. 1151, Babaoglu O, Marzullo K (eds.). Springer: Berlin, Germany, 1996; 1–17.
13. Foster I, Kesselman C. Globussss: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
14. Foster I. Globus toolkit version 4: Software for service-oriented systems. *IFIP International Conference on Network and Parallel Computing* (*Lecture Notes in Computer Science*, vol. 3779). Springer: Berlin, 2005; 2–13.
15. Baru CK, Moore RW, Rajasekar A, Wan M. The SDSC storage resource broker. *CASCON* 1998; 5.
16. Fuhrmann P, Gülzow V. dCache, storage system for the future. *Euro-Par*, 2006; 1106–1113.
17. Santos N, Koblitz B. Distributed metadata with the AMGA metadata catalog. *Proceedings of the Workshop on Next-Generation Distributed Data Management—HPDC-15*, 2006.
18. Tatebe O, Morita Y, Matsuoka S, Soda N, Sekiguchi S. Grid datafarm architecture for petascale data intensive computing. *CCGRID '02*: *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society: Washington, DC, U.S.A., 2002; 102.
19. Tatebe O, Sekiguchi S, Soda N, Morita Y, Matsuoka S. Gfarm v2: A grid file system that supports high-performance distributed and parallel data computing. *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics* (*CHEP 2004*), 2004.
20. Panasas ActiveScale File System (PanFS). Whitepaper.
21. Lustre: A Scalable, High-Performance File System. Whitepaper.