Towards A Grid File System Based On A Large-Scale BLOB Management Service

Viet-Trung Tran, Gabriel Antoniu, Bogdan Nicolae, Luc Bougé

TOWARDS A GRID FILE SYSTEM BASED ON A LARGE-SCALE BLOB MANAGEMENT SERVICE

Viet-Trung Tran[1], Gabriel Antoniu[2], Bogdan Nicolae[3], Luc Bougé[1]

[1]*ENS Cachan/Brittany, IRISA, France*

[2]*INRIA, Centre Rennes - Bretagne Atlantique, IRISA, Rennes, France*

[3]*University of Rennes 1, IRISA, Rennes, France*

{Viet-Trung.Tran, Gabriel.Antoniu, Bogdan.Nicolae, Luc.Bouge}@irisa.fr

Osamu Tatebe

University of Tsukuba, Tsukuba, Japan

tatebe@cs.tsukuba.ac.jp

Abstract This paper addresses the problem of building a grid file system for applications that need to manipulate huge data, distributed and concurrently accessed at a very large scale. In this paper we explore how this goal could be reached through a cooperation between the Gfarm grid file system and BlobSeer, a distributed object management system specifically designed for huge data management under heavy concurrency. The resulting BLOB-based grid file system exhibits scalable file access performance in scenarios where huge files are subject to massive, concurrent, fine-grain accesses. This is demonstrated through preliminary experiments of our prototype, conducted on the Grid'5000 testbed.

Keywords: Distributed File System, Object-based Storage, Massive Data, Concurrency, Grid.

1. Introduction

The need for transparent grid data management. As more and more applications in many areas (nuclear physics, health, cosmology, etc.) generate larger and larger volumes of data that are geographically distributed, appropriate mechanisms for storing and accessing data at a global scale become increasingly necessary. Grid file systems (such

as LegionFS [17], Gfarm [13], etc.) prove their utility in this context, as they provide a means to federate a very large number of large-scale distributed storage resources and offer a large *storage capacity* and a good *persistence* achieved through file-based storage. Beyond these properties, grid file systems have the important advantage of offering a *transparent access to data* through the abstraction of a shared file namespace, in contrast to explicit data transfer schemes (e.g. GridFTP-based [1], IBP [3]) currently used on some production grids. Transparent access greatly simplifies data management by applications, which no longer need to explicitly locate and transfer data across various sites, as data can be accessed the same way from anywhere, based on globally shared identifiers. Implementing transparent access at a global scale naturally leads however to a number of challenges related to scalability and performance, as the file system is put under pressure by a very large number of concurrent, largely distributed accesses.

From block-based to object-based distributed file systems. Recent research [6] emphasizes a clear move currently in progress from a block-based interface to a object-based interface in storage architectures, with the goal of enabling scalable, self-managed storage networks by moving low-level functionalities such as space management to storage devices or to storage server, accessed through a standard object interface. This move has a direct impact on the design of today's distributed file systems: object-based file system would then store data rather as objects than as unstructured data blocks. According to [6], this move may eliminate nearly 90% of management workload which was the major obstacle limiting file systems' scalability and performance.

Two approaches exploit this idea. In the first approach, the data objects are stored and manipulated directly by a new type of storage device called *object-based storage device* (OSD). This approach requires an evolution of the hardware, in order to allow high-level object operations to be delegated to the storage device. The standard OSD interface was defined in the Storage Networking Industry Association (SNIA) OSD working group. The protocol is embodied over SCSI and defines a new set of SCSI commands. Recently, a second generation of the command set, Object-Based Storage Devices - 2 (OSD-2) has been defined. The distributed file systems taking the OSD approach assume the presence of such an OSD in the near future and currently rely on a software module simulating its behavior. Examples of parallel/distributed file systems following this approach are Lustre [12] and Ceph [16]. Recently, research efforts [5] have explored the feasibility and the possible benefits of integrating OSDs into parallel file systems, such as PVFS [4].

The second approach does not rely on the presence of OSDs, but still tries to benefit from an object-based approach to improve performance and scalability: files are structured as a set of objects that are stored on storage servers. Google File System [8], and HDFS (Hadoop File System) [2]) illustrate this approach.

Large-scale distributed object storage for massive data. Beyond the above developments in the area of parallel and distributed file systems, other efforts rely on objects for large-scale data management, without exposing a file system interface. BlobSeer [10] [9] is such a BLOB (binary large object) management service specifically designed to deal with large-scale distributed applications, which need to store massive data objects and to efficiently access (read, update) them at a fine grain. In this context, the system should be able to support a large number of BLOBs, each of which might reach a size in the order of TB. BlobSeer employs a powerful concurrency management scheme enabling a large number of clients to efficiently read and update the same BLOB simultaneously in a lock-free manner.

A two-layer architecture. Most object-based file systems exhibit a decoupled architecture that generally consists of two layers: a low-level object management service, and a high-level file system metadata management. In this paper we propose to explore how this two-layer approach could be used in order to build an object-based grid file system for applications that need to manipulate huge data, distributed and concurrently accessed at a very large scale. We investigate this approach by experimenting how the Gfarm grid file system could leverage the properties of the BlobSeer distributed object management service, specifically designed for huge data management under heavy concurrency. We thus couple Gfarm's powerful file metadata capabilities and rely on BlobSeer for efficient and transparent low-level distributed object storage. We expect the resulting BLOB-based grid file system to exhibit scalable file access performance in scenarios where huge files are subject to massive, concurrent, fine-grain accesses. We intend to deploy a BlobSeer instance at each Gfarm storage node, to handle object storage. The benefits are mutual: by delegating object management to BlobSeer, Gfarm can expose efficient fine-grain access to huge files and benefit from transparent file striping (TB size). On the other hand, BlobSeer benefits from the file system interface on top of its current API.

The remaining of this paper is structured as follows. Section 2 introduces the two components of our object-based file system: Blob-

Seer and Gfarm, whose coupling is explained in Section 3. Section 4 presents our preliminary experiments on the Grid'5000 testbed. Finally, Section 5 summarizes the contribution and discusses future directions.

2. The building blocks: Gfarm and BlobSeer

Our object-based grid file systems consists of two layers: a high-level file metadata layer, available with the Gfarm file system; a low-level storage layer based on the BlobSeer BLOB management service.

2.1 The Gfarm grid file system

The Grid Datafarm (Gfarm) [13] is a distributed file system designed for high-performance data access and reliable file sharing in large scale environments including grids of clusters. To facilitate file sharing, Gfarm manages a global namespace which allows the applications to access files using the same path regardless of file location. It federates available storage spaces of Grid nodes to provide a single file system image. We have used Gfarm v2.1.0 in our experiments.

Overview of Gfarm's architecture. Gfarm consists of a set of communicating components, each of which fulfills a particular role.

Gfarm's metadata server: the gfmd daemon. The metadata server stores and manages the namespace hierarchy together with file metadata, user-related metadata, as well as file location information allowing clients to physically locate the files.

Gfarm file system nodes: the gfsd daemons. They are responsible for physically storing full Gfarm files on their local storage. Gfarm does not implement file stripping and here is where BlobSeer can bring its contribution, through transparent file fragmentation and distribution.

Gfarm clients: Gfarm API and FUSE access interface for Gfarm. Gfarm provides users with a specific API and several command lines to access the Gfarm file system. To facilitate data access, the Gfarm team developed Gfarm2fs: a POSIX file system interface based on the FUSE library [7]. Basically, Gfarm2fs transparently maps all standard file I/Os to the corresponding routines of the Gfarm API. Thus, existing applications handling files must no longer be modified in order to work with the Gfarm file system.

2.2 The BlobSeer BLOB management service

BlobSeer at a glance. BlobSeer [10] [9] addresses the problem of storing and efficiently accessing very large, unstructured data objects, in a distributed environment. It focuses on heavy access concurrency where data is huge, mutable and potentially accessed by a very large number of concurrent, distributed processes. To cope with very large data BLOBs, BlobSeer uses striping: each BLOB is cut into fixed-size *pages*, which are distributed among data providers. *BLOB Metadata* facilitates access to a range *(offset, size)* for any existing version of a BLOB snapshot, by associating such a range with the physical nodes where the corresponding pages are located. Metadata are organized as a segment-tree like structure (see [10] for details) and are scattered across the system using a Distributed Hash Table (DHT). Distributing data and metadata is the key choice in our design: it enables high performance through parallel, direct access I/O paths, as demonstrated in [11]. Further, BlobSeer provides concurrent clients with efficient fine-grained access to BLOBs, without locking. To deal with the mutable data, BlobSeer introduces a *versioning* scheme which allows clients not only to roll back data changes when desired, but also enables access to multiple versions of the same BLOB within the same computation.

Overview of BlobSeer's architecture. The system consists of distributed processes, that communicate through remote procedure calls (RPCs). A physical node can run one or more processes and, at the same time, may play multiple roles from the ones mentioned below.

Clients. Clients may issue CREATE, WRITE, APPEND and READ requests. There may be multiple concurrent clients. Their number dynamically vary in time without notifying the system.

Data providers. Data providers physically store and manage the pages generated by WRITE and APPEND requests. New data providers are free to join and leave the system in a dynamic way.

The provider manager. The provider manager keeps information about the available data providers and schedules the placement of newly generated pages according to a load balancing strategy.

Metadata providers. Metadata providers physically store the metadata, allowing clients to find the pages corresponding to the various BLOB versions. Metadata providers are distributed, to allow an efficient concurrent access to metadata.

The version manager. The version manager is the key actor of the system. It registers update requests (APPEND and WRITE), assigning BLOB version numbers to each of them. The version manager eventually publishes these updates, guaranteeing total ordering and atomicity.

Accessing data in BlobSeer. To READ data, the client contacts the version manager: it needs to provide a BLOB id, a specific version of that BLOB, and a range, specified by an offset and a size. If the specified version is available, the client queries the metadata providers to retrieve the metadata indicating the location of the pages for the requested range. Finally, the client contacts *in parallel* the data providers that store the corresponding pages.

For a WRITE request, the client contacts the provider manager to obtain a list of providers, one for each page of the BLOB segment that needs to be written. Then, the client contacts the providers in the list *in parallel* and requests them to store the pages. Each provider executes the request and sends an acknowledgment to the client. When the client has received all the acknowledgments, it contacts the version manager and requests a new version number. This version number is then used by the client to generate the corresponding new metadata. Finally, the client notifies the version manager of success, and returns successfully to the user. At this point, the version manager is responsible for eventually publishing the new version of the BLOB. The APPEND operation is a particular case of WRITE, where the offset is implicitly the size of the previously published snapshot version. The detailed algorithms for READ, WRITE and APPEND are given in [10].

2.3 Why combine Gfarm and BlobSeer?

Gfarm does not rely on autonomous, self-managing object-based storage, like the file systems mentioned in Section 1. Each Gfarm file is fully stored on a file system node, or totally replicated to multiple file system nodes. If a large number of clients concurrently access small parts of the same copy of a huge file, this can lead to a bottleneck both for reading and for writing. Second, Gfarm's file sizes are limited by the storage capabilities of the machines used as file system nodes in the Gfarm deployment. However, some powerful features, including user management, authentication and single sign-on (based on GSI: Grid Security Infrastructure [14]) are present in Gfarm's current implementation. Moreover, due to the Gfarm's FUSE access interface, data can be accessed in a transparent manner via the POSIX file system API.

BlobSeer brings different benefits: it handles huge data, which is transparently fragmented and distributed at a large scale. Thanks to its distributed metadata scheme, it sustains a high bandwidth is maintained even when the BLOB grows to large sizes, and when the BLOB faces heavy concurrent access [11]. BlobSeer is mostly suitable for massive data processing, fine-grained access, and versioning in a large-scale distributed environment. But BlobSeer lacks a file system interface that may help existing applications to use it directly. As explained above, such an interface is provided by Gfarm, together with the associated file system metadata management. It then clearly appears that making Gfarm cooperate with BlobSeer would enhance their respective functionalities and would lead to an object-based file system with better properties: huge file support (TBs), fine-grain access under heavy concurrency, versioning, user and GSI-compliant security management. In this paper we focus on providing an enhanced concurrency support. Exposing multiversioning to the file system user is currently under study and will not be addressed in this paper.

3. Towards an object-based file system based on Gfarm and BlobSeer

3.1 How to couple Gfarm and BlobSeer?

Since each gfsd daemon running on Gfarm's file system nodes is responsible for physically storing Gfarm's data on its local file system, our first approach aims at integrating BlobSeer calls at the gfsd daemon. The main idea is to trap all requests to the local file system, and map them to the corresponding BlobSeer API in order to leave the job of storing Gfarm's data to BlobSeer. A Gfarm file is no longer directly stored as a file on the local system; it is stored as a BLOB in BlobSeer. This way, file fragmentation and striping is introduced transparently for Gfarm at the gfsd level.

Nevertheless, this way of integrating BlobSeer into gfsd daemon clearly does not fully exploit BlobSeer's capability of efficiently handling concurrency, in which multiple clients simultaneously access the same BLOB. The gfsd daemon always acts as an intermediary for data transfer between Gfarm clients and BlobSeer data providers, which may limit the data transfer throughput. For this reason, we propose a second approach. Currently, Gfarm defines two modes for data access, *local access mode* and *remote access mode*. The *local access mode* is the mode in which the client and the gfsd daemon involved in a data transaction are on the same physical node, allowing the client to directly access its

local disk. In contrast, the *remote access mode* is the mode in which a client accesses data through a remote gfsd daemon.

Our second approach consists in introducing into Gfarm a new access mode, called *BlobSeer direct access mode*, allowing Gfarm clients to directly access BlobSeer. In this mode, as explained in Section 2.2, clients benefit from a better throughput, as they access the distributed BLOB pages in parallel. During data accesses, the risk to create a bottleneck at the gfsd level is then reduced, since the gfsd daemon no longer acts as an intermediary for accessing data; its task now is simply to establish the mapping between Gfarm logical files and BlobSeer's corresponding BLOB ids. Keeping the management of this mapping at the gfsd level is important, as, this way, no change is required on Gfarm's metadata server (gfmd), which is not aware of the use of BlobSeer.

3.2 The Gfarm/BlobSeer file system design

The Gfarm/BlobSeer cooperation aims at working on a large-scale distributed environment where multiple *sites* in different administrative domains interconnect with each other to form a global network. Therefore, it is vital that our design is scalable to such settings.

A global view. We assume that our object-based file system runs on a multi-site grid, where each site corresponds to a specific cluster. As shown on Figure 1, the whole system consists of a single instance of Gfarm, with one metadata server (gfmd), multiple distributed clients and multiple file system nodes (gfsd). In addition to this regular Gfarm configuration, we introduce multiple instances of BlobSeer (one per site). Any node of the grid may be a client. On each site, a dedicated node runs a gfsd daemon and the other nodes run a BlobSeer instance, with all its entities described in Section 2.2. On each site, the gfsd daemon is responsible for mapping Gfarm files to BLOBs and for managing all BLOBs on the site. This approach guarantees the independent administration of the sites. By separating the whole system into different sites, we provide a simple strategy for efficiently using different *access modes* whenever a client access a Gfarm file. Typically, if the client is on the same site with the BlobSeer instance that stores the BLOB corresponding to the desired Gfarm file, it then should use the *BlobSeer direct access mode*, allowing for parallel access of the BLOB pages by the client. Otherwise, the client may not be able to directly access the BlobSeer instance of a remote *site*, due to security policies. In that case, the *remote access mode* is more appropriate: the client may access data through the gfsd daemon of the remote *site*, which acts as a proxy.

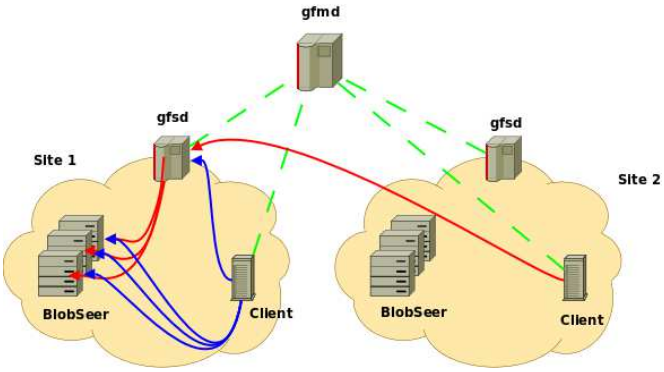


Figure 1. A global view of the Gfarm/BlobSeer system.

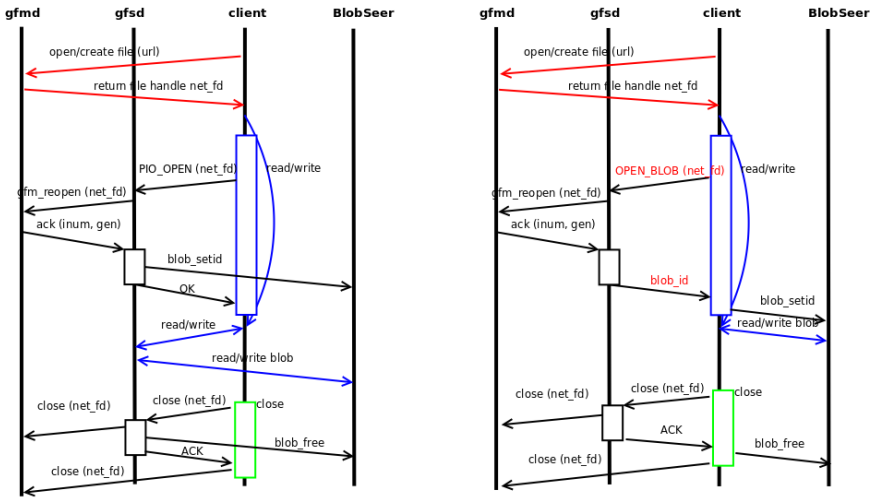


Figure 2. The internal interactions inside Gfarm/BlobSeer system: remote access (left) vs BlobSeer direct access mode (right).

Description of the interactions between Gfarm and BlobSeer. Figure 2 describes the interactions inside the Gfarm/BlobSeer system, both for *remote access mode* (left) and *BlobSeer direct access mode* (right). When opening a Gfarm file, the *global path name* is sent from the client to the metadata server. If no error occurs, the metadata server returns to the client a *network file descriptor* as an identifier of the requested Gfarm file. The client then initializes the *file handle*. On a *write* or *read* request, the client must first initialize the access node (if not done yet), after having authenticated itself with the gfsd daemon. Details are given below.

Remote access mode. In this access mode, the internal interactions of Gfarm with BlobSeer only happen through the gfsd daemon. After receiving the *network file descriptor* from the client, the gfsd daemon inquires the metadata server about the corresponding Gfarm's global ID and maps it to a BLOB id. After opening the BLOB for reading and/or writing, all subsequent read and write requests received by the gfsd daemon are mapped to BlobSeer's data access API.

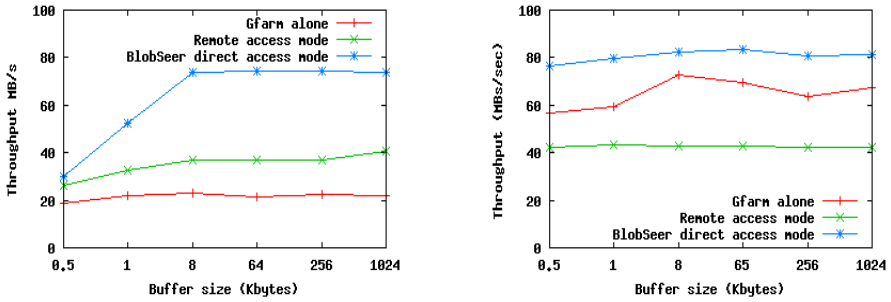
BlobSeer direct access mode. In order for the client to directly access the BLOB in the *BlobSeer direct access mode*, there must be a way to send the ID of the desired BLOB from the gfsd daemon to the client. With this information, the client is further able to directly access BlobSeer without any help from the gfsd.

4. Experimental evaluation

To evaluate our Gfarm/BlobSeer prototype, we first compared its performance for read/write operations to that of the original Gfarm version. Then, as our main goal was to enhance Gfarm's data access performance under heavy concurrency, we evaluated the read and write throughput for Gfarm/BlobSeer in a setting where multiple clients concurrently access the same Gfarm file. Experiments have been performed on the Grid'5000 [15] testbed, an experimental grid infrastructure distributed on 9 sites around France. In each experiment, we used at most 157 nodes of the Rennes site of Grid'5000. Nodes are outfitted with 8 GB of RAM, Intel Xeon 5148 LV CPUs running at 2.3 GHz and interconnected by a Gigabit Ethernet network. Intra-cluster measured bandwidth is 117.5 MB/s for TCP sockets with MTU set at 1500 B.

Access throughput with no concurrency. First, we mounted our object-based file system on a node and used Gfarm's own benchmarks to measure file I/O bandwidth for sequential reading and writing. Basically, the Gfarm benchmark is configured to access a single file that contains 1 GB of data. The block size for each read (respectively write) operation varies from 512 bytes to 1,048,576 bytes.

We used the following setting: for Gfarm, a metadata server and a single file system node. For BlobSeer, we used 10 nodes: a version manager, a metadata provider and a provider manager were deployed on a single node, and the 9 other nodes hosted data providers. We used a page_size of 8 MB. We measured the read (respectively write) throughput for both access modes of Gfarm/BlobSeer: *remote access mode* and



(a) Writing (b) Reading

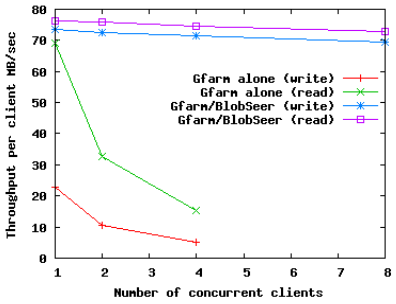
Figure 3. Sequential write (left) and read (right).

BlobSeer direct access mode. For comparison, we ran the same benchmark on a pure Gfarm file system, using the same setting for Gfarm alone.

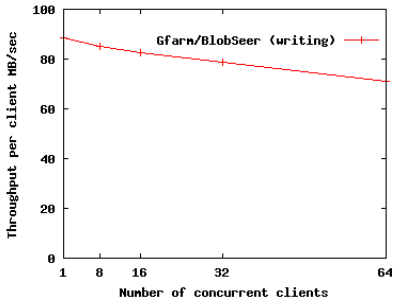
As shown on Figure 3, the average read throughput and write throughput for Gfarm alone are 65 MB/s and 20 MB/s respectively in our configuration. The I/O throughput for Gfarm/BlobSeer in *remote access mode* was better than the pure Gfarm's throughput for the write operation, as in Gfarm/BlobSeer data is written in a remote RAM and then, asynchronously, on the corresponding local file system, whereas in the pure Gfarm the gfsd synchronously writes data on the local disk. As expected, the read throughput is worse then for the pure Gfarm, as going through the gfsd daemon induces an overhead.

On the other hand, when using the *BlobSeer direct access mode*, Gfarm/BlobSeer clearly shows a significantly better performance, due to parallel accesses to the striped file: 75 MB/s for writing (i.e. 3.75 faster than the measured Gfarm throughput) and 80 MB/s for reading.

Access throughput under concurrency. In a second scenario, we progressively increase the number of concurrent clients which access disjoint parts (1 GB for each) of a file totaling 10 GB, from 1 to 8 clients. The same configuration is used for Gfarm/BlobSeer, except for the number of data providers in BlobSeer, set to 24. Figure 4(a) indicates that the performance of the pure Gfarm file system decreases significantly for concurrent accesses: the I/O throughput for each client drops down twice each time the number of concurrent clients is doubled. This is due to a bottleneck created at the level at the gfsd daemon, as its local file system basically serializes all accesses. In contrast,



(a) Gfarm alone & Gfarm/BlobSeer



(b) Heavy access concurrency on Gfarm/BlobSeer

Figure 4. Access concurrency

a high bandwidth is maintained when Gfarm relies on BlobSeer, even when the number of concurrent clients increases, as Gfarm leverages BlobSeer's design optimized for heavy concurrency.

Finally, as a scalability test, we realized a third experiment. We ran our Gfarm/BlobSeer prototype using a 154 node configuration for BlobSeer, including 64 data providers, 24 metadata servers and up to 64 clients. In the first phase, a single client appends data to the BLOB until the BLOB grows to 64 GB. Then, we increase the number of concurrent clients to 8, 16, 32, and 64. Each client writes 1 GB to that file at a disjoint part. The average throughput obtained (Figure 4(b)) slightly drops (as expected), but is still sustained at an acceptable level. Note that, in this experiment, the write throughput is slightly higher than in the previous experiments, since we directly used Gfarm's library API, avoiding the overhead due to the use of Gfarm's FUSE interface.

5. Conclusion

In this paper we address the problem of managing large data volumes at a very large-scale, with a specific focus on applications which manipulate huge data, physically distributed, but logically shared and accessed at a fine-grain under heavy concurrency. Using a grid file system seems the most appropriate solution for this context, as it provides transparent access through a globally shared namespace. This greatly simplifies data management by applications, which no longer need to explicitly locate and transfer data across various sites. In this context, we explore how a grid file system could be built in order to address the

specific requirements mentioned above: huge data, highly distributed, shared and accessed under heavy concurrency. Our approach relies on establishing a cooperation between the Gfarm grid file system and BlobSeer, a distributed object management system specifically designed for huge data management under heavy concurrency. We define and implement an integrated architecture, and we evaluate it through a series of preliminary experiments conducted on the Grid'5000 testbed. The resulting BLOB-based grid file system exhibits scalable file access performance in scenarios where huge files are subject to massive, concurrent, fine-grain accesses.

We are currently working on introducing versioning support into our integrated, object-based grid file system. Enabling such a feature in a global file system can help applications not only to tolerate failures by providing support for roll-back, but will also allow them to access different versions of the same file, while new versions are being created. To this purpose, we are currently defining an extension of Gfarm's API, in order to allow the users to access a specific file version. We are also defining a set of appropriate ioctl commands: accessing a desired file version will then be completely done via the POSIX file system API.

In the near future, we also plan to extend our experiments to more complex, multi-cluster grid configurations. Additional directions will concern data persistence and consistency semantics. Finally, we intend to perform experiments to compare our prototype to other object-based file systems with respect to performance, scalability and usability.

References

[1] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28(5):749–771, 2002.

[2] Apache Inc. http://hadoop.apache.org/core/docs/current/hdfs_design.html.

[3] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James S. Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A study in resource sharing. In *Proc. 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGRID '02)*, page 194, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[5] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices.

In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[6] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In *Local to Global Data Interoperability - Challenges and Technologies, 2005*, pages 119–123, 2005.

[7] FUSE. http://fuse.sourceforge.net/.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[9] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Distributed management of massive data. an efficient fine grain data access scheme. In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2008)*, Toulouse, 2008. Held in conjunction with VECPAR'08. Electronic proceedings.

[10] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Blobseer: How to enable efficient versioning for large object storage under heavy access concurrency. In *EDBT '09: 2nd International Workshop on Data Management in P2P Systems (DaMaP '09)*, St Petersburg, Russia, 2009.

[11] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach. In *Proceedings of the 15th Euro-Par Conference on Parallel Processing (Euro-Par 09)*, Lect. Notes in Comp. Science, Delft, The Netherlands, 2009. Springer-Verlag. To appear.

[12] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[13] Osamu Tatebe and Satoshi Sekiguchi. Gfarm v2: A grid file system that supports high-perfomance distributed and parallel data computing. In *Proceedings of the 2004 Computing in High Energy and Nuclear Physics*, 2004.

[14] The Grid Security Infrastructure Working Group. http://www.gridforum.org/security/gsi/.

[15] The Grid'5000 Project. http://www.grid5000.fr/.

[16] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[17] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In *Proc. 2001 ACM/IEEE Conf. on Supercomputing (SC '01)*, pages 59–59, New York, NY, USA, 2001. ACM Press.