CloViS : a Secured Cloud Virtualized Storage

Olivier Lafontaine, Jacques Jorda, Abdelaziz M'Zoughi

CloViS : a Secured Cloud Virtualized Storage

Olivier Lafontaine, Jacques Jorda and Abdelaziz M'zoughi
Institut de Recherche en informatique de Toulouse
Université Paul Sabatier, Toulouse
Email: { lafontai | jorda | mzoughi }@irit.fr

ABSTRACT

Cloud data storage may be used in different cases: for backup purpose, for hosted services or due to the virtualization of servers. Whatever the reason, the data may be secured according to three criteria: availability, integrity and confidentiality. These criteria are fundamental to establish a trust relationship between the customer and the provider. However, enforcing some of these constraints may have a significant cost, and even weaken other of these requirements. Thus, the main difficult task is to balance between opposite constraints depending on the service paid by the client. To overcome this problem, we propose the use of a storage virtualization middleware, named CloViS. This software layer allows pluggable QoS libraries to be used to implement the required security mechanisms.

I INTRODUCTION

Storing data in the cloud is critical due to the number of threats likely to affect the data. These threats can be classified in two categories: the ones that are generals, and the ones that are specific to cloud environment. In the first category, we typically find injections, cross-site scripting and phishing. We won't develop these threats, as they are classical and not specific to the cloud environment. Specific threats encompass mainly the security of virtualization layers, and the trust in the provider.

Most virtualization platforms have experienced security flaws: rootkits (like the blue pill one), implementation of backdoors in modules, in unprivileged domains or in hypervisors spaces, are examples of the most commonly prevailing attacks. Using these flaws, a malicious user can access, modify or delete data, thus impacting the availability, the integrity and the confidentiality of the latter.

The provider himself – either the company or a malicious administrator, can also jeopardize data. They may be unavailable due to an infrastructure problem or for economic reasons: infrequently accessed data may be deleted or moved to high latency, low bandwidth servers to optimize the provider costs.

The responses to these threats depend on the storage implementation. Cloud data storage is mainly built on two architectures: the use of shared storage like Storage Area Network (SAN) or Network Attached Storage (NAS), or the distribution of data on clusters. The first solution is rather used in private clouds, but less used in public or hybrid clouds than the second solution. In fact, although an easier implementation, its cost and the complex underlying architecture remains an obstacle in large scale infrastructures. Moreover, it limits the concurrent I/O requests [1] compared to what observed on the second case. The latter solution is typically implemented using a specific filesystem for seamless client access, data being stored on nodes according to some specific placement policies. Thereby, our problem can be stated as: how to implement data availability, integrity and confidentiality in such a distributed storage system.

In section 2, we will review the classical solutions for cloud storage security. We will present in section 3 the existing storage solutions and show that none of them encompasses all aspects of data security. In section 4 we will present the global architecture of our virtualization middleware, and explain how it can be used to implement the required security. The section 5 will present our preliminary results and the future implementations planned.

II CLOUD STORAGE SECURITY

Whatever the threat, the provider must ensure the availability, the integrity and the confidentiality of data. The availability guarantees that data will always be accessible, even in the case of an infrastructure problem. Integrity makes it possible to verify that data is actually stored, completely and with no alteration, proving the dependability of the storage provider. Confidentiality ensures that only the proprietary or an authorized third party may access the real content of data stored. However, some operations like data searches may not be affected by this criterion.

1 DATA AVAILABILITY

A data is said unavailable is a client request gets no response, due for example to the failure of the node on which the data is stored. To relieve the failure (or more generally the unavailability) of one node, the use of redundancy is essential [2]. This redundancy may be implemented by writing one or more replicas, or by using Reed-Solomon codes. The overhead on write requests induced by these techniques can be minimized by specific implementations [3].

Whatever the redundancy scheme chosen, the node selection for data placement is primordial. Specific policies, like using burst failures detection [1] to select the best node, are usually implemented in cloud environment. But as soon as numerous data replicates are used, some coherency and concurrency algorithms must be used to guarantee safe accesses to stored data.

2 DATA INTEGRITY

Data integrity audit may be carried on either by the proprietary or by a third party auditor (TPA). This verification process is complex for many reasons. First, it may be conducted without any copy of the data. Secondly, the verification overhead must be as low as possible on the provider's servers (no comprehensive check on all blocks of all files) and for the client (connected by low bandwidth network links to the provider). Thirdly, when conducted by a TPA, the latter must be able to check data integrity without breaking the confidentiality, i.e. without any knowledge of the data. Fourthly, the number of checks must be unlimited. Lastly, the solution implemented must allow dynamic data updates without involving significant overhead.

Two major schemes have been proposed for such checks. First, the Proofs of Retrievability [4], in which each file is divided in blocks, some special blocks (called sentinels) being added. The latter allows the integrity to be checked, without being differentiable from other blocks (all blocks being cyphered). Secondly, the Proofs of Data Possession [5], based on the use of signature computed for each block of each file. Integrity check is conducted by querying the provider the signature of some blocks randomly chosen. Unfortunately, no scheme currently supports dynamic updates with unlimited number of checks.

3 DATA CONFIDENTIALITY

Ciphering data is usually done using symmetric or asymmetric encryption schemes. The problem is complex because data can be shared by the proprietary (thus sharing cyphering keys [6] or using proxy re-encryption [7]), the storage provider can change the blocks lock order to ensure the required quality of service [8], or the provider may be required to implement some searchable services on cyphered data [9].

However, this ciphering is ineffective for the storage of virtual machines disks files [10]. In this context, the virtual disk file must be ciphered by the virtual machine itself, or some Trusted Platform Module must be used to ensure the confidentiality.

4 IMPLEMENTATION

Data security in cloud computing involves many aspects. Currently, some solutions encompass two criteria [13] but not all three criteria detailed previously, and it is reasonable to think that it is not necessary in most cases. In fact, depending on the use case, some security constraints will be weak or inexistent. For example, storing data on the cloud for backup purpose requires the implementation of integrity checks; however the availability is of lower importance. In contrast, when using SaaS solutions, the integrity checks are useless but the availability may be a strong requirement. However, the same software

infrastructure should offer all of these security solutions and let the end-user select the ones he needs.

III EXISTING SOLUTIONS FOR CLOUD STORAGE

Implementation of cloud storage is mainly done using either SAN or dedicated storage servers. The first solution is the most simple to use but also the most costly, and thus is not used when a large amount of data should be stored at a reasonable cost.

The second solution is mainly used to provide data storage for backup purpose as well as for SaaS, PaaS and IaaS. In order to be efficient, special servers are used with low-power processors, limited RAM but high disk integration (up to 24 disks in a 2U server). The problem is then to provide an abstraction of the physical resources to the software layer – or to be able to manage this huge number of disks through a dedicated set of tools.

The two main storage management tools are Ceph and GlusterFS. They both offer an efficient way to manage storage, but they have their own limits on security implementation.

1 CEPH

Storage implementation in Ceph / RADOS uses CRUSH algorithm [14]. CRUSH is a pseudo-random data distribution function based on maps to distribute data among available storage resources. Dynamic data placement is handled even when adding or removing low-level storage resources thus ensuring a balanced distribution of data among nodes and disks. The number of replicas and their placement are managed using a configurable map. Thus high-availability schemes may be implemented to ensure data access even on node / rack failure.

However, Ceph / RADOS aims to provide an efficient, robust and scalable distributed storage system, but neither confidentiality nor data integrity are implemented. Messages are not ciphered over the network and no encryption algorithm is used on object stores. Last, by focusing only on data availability and scalability, Ceph / RADOS misses the energy optimization goal.

2 GLUSTERFS

GlusterFS is a client/server structured filesystem where server components act as storage bricks while client components handle virtual volumes using stackable translators and a few more functionalities (distribute, replicate, stripe). Thus GlusterFS may implement some kind of confidentiality using translators (even if it does not offer a fully secure storage, some translators extensions of HekaFS are being ported), does basic replication and stripping. It helps balance load over bricks, and using a metadata hash function (just like Ceph) it avoids the problem of single point of failures on metadata server.

However, some limitations should be noted. First, the authentication of clients is done on IP address basis, leaving the door open for malicious users. Moreover, replication is not tunable: either you replicate data on all bricks – or not. You cannot add multiple level redundancies. Last, the energy-awareness is not considered and cannot be implemented simply using this filesystem.

IV STORAGE VIRTUALIZATION

Storage virtualization aims to provide an abstraction layer for a seamless access to physical storage resources. Using such a middleware, the storage space available on physical disks is aggregated on a virtual space. This virtual space is then divided on logical volumes, on which some quality of service may be defined.

1 STORAGE VIRTUALIZATION MIDDLEWARE

Based on our expertise on grid storage virtualization [11], we are implementing a storage virtualization layer dedicated to cloud computing, named CloViS (**Clo**ud **Vi**rtualized **S**torage).

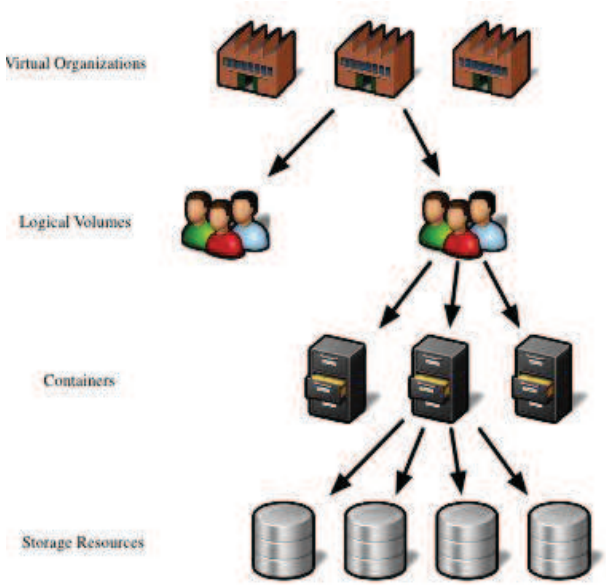
CloViS aims to aggregate physically distributed storage resources into logical volumes seamlessly accessible. These storage resources may be different, either regarding their size, their bandwidth or their latency. They may be attached to different servers running different OS releases on different hardware architectures. Whatever the components on the infrastructure, the CloViS middleware offers a unified view to the storage space.

Targeting the cloud, we must ensure isolation between clients of the provider. The storage space is then organized in Virtual Organizations (VO). A VO is dedicated to a specific client and everything managed on one VO must be accessible only to that client.

Moreover, a single client may have numerous departments and should be allowed to ensure isolation between its storage spaces. The global storage space dedicated to a client may then be divided into numerous Logical Volumes (LV).

The Cloud Service Provider (CSP) will allow some storage space on numerous low-level storage resources regarding the requirements of the client – and will bill him for it. For example, one may need a very fast storage system for running disk bandwidth consume services on one hand, and a slower but highly reliable storage system for archiving the results produced by its services. The CSP may then allocate some space on SSD for the first storage space, and some space on numerous classical disks for archiving tasks. Based on the requirements of the client, the CSP will dedicate some storage resources for him at the VO level.

It then appears that different clients may have different needs, and even a single client may need numerous storage space characteristics. Thus, every single logical volume may use numerous quality of service for data storage. These qualities of services are implemented using containers (CAN). Indeed, logical volumes are made of containers, each implementing a specific QoS. Every time a file is created, it inherits the QoS of its parent. When this QoS is modified, CloViS looks for a container implementing this QoS on this logical volume. If such container cannot be found, a new one is created using the available storage resources declared at the virtual organization level. The file



a) Storage organization in CloViS

is then moved from the original container to target one.

2 CLOVIS SOFTWARE ARCHITECTURE

CloViS is made of several components: a communication library, a coherency and consistency library, administration and monitoring tools, a virtualizer (with QoS libraries) and a client access component.

All these components (except QoS libraries that have a specific behavior and thus a specific architecture) are based on the same software model.

2.1 CLOVIS COMPONENTS

CloViS being daemonizable, it is implemented as a single-process multiple-threads software. All the components have the same architecture: a server part listen for messages received using the communication library and manages them, while a client part interacts with the other components (using their respective server parts). Message handling is done using a pool of threads in order to keep a non-blocking architecture. Let first review the main five components of CloViS.

CCOM: the communication component

CCOM implements the communication protocol used in CloViS. The communication can either be at node level (between components of a single node), local (between nodes on the same network) or distant using a proxy node in charge of managing message routing over Internet.

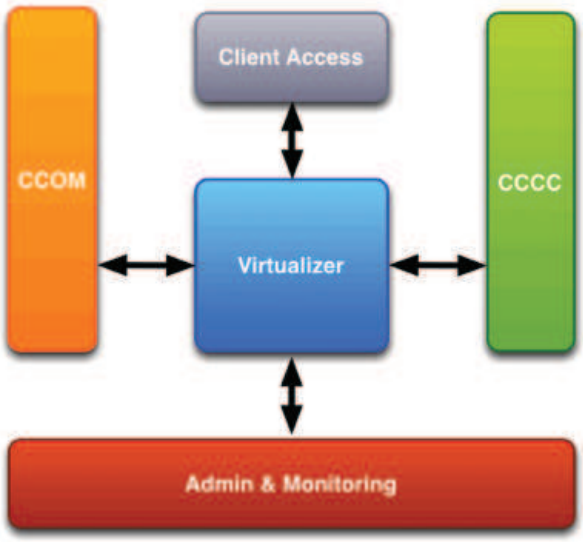
CCCC: a coherency and consistency component

Using a distributed filesystem leads to the need for a coherency mechanism (required to handle locks on objects) and a consistency protocol (to manage an adequate versioning for replicas). Note that CloViS exposing a block-like interface through the virtualizer, this component should have been of less importance (except when using a distributed filesystem as client access). However, CloViS metadata being replicated over the nodes, such component is required to ensure a consistent state.

CCCC implements an efficient, highly dependable, fully distributed locking mechanism and a basic coherency protocol, with no bottlenecks [12].

CAM: Administration and Monitoring component

This component is made of a local agent distributed over all nodes, and some command-line tools. The local agent receives through its server part all actions to achieve locally. These actions are related to both administrative and monitoring tasks: create VO / LV / CAN, add storage resources, retrieve disks SMART info, etc.



b) CloViS middleware architecture

VRT: the CloViS virtualizer

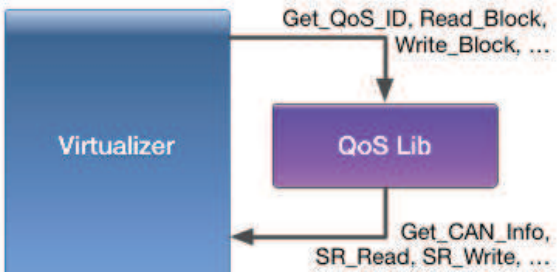
The virtualizer is a central component in CloViS architecture. It aims to handle the aggregation of physical storage resources and offers a seamless block access to the virtualized storage space. Thus, client access layer can easily interact with CloViS as if the virtualized storage was a local physical storage resource on the node.

QoS libraries ensure data distribution among the underlying physical storage resources making up the virtual space. These QoS libraries implement each a specific quality of service (data integrity, confidentiality, etc.), and their interface being both simple to understand and made of a very few functions, anyone can implement its own specific algorithm. For example, a CRUSH algorithm could be implemented in such a library to ensure a balanced distribution of data over physical storage resources.

In fact, QoS libraries use a low-level virtualizer API (read, write or delete physical blocks, retrieve the list of storage resources involved in a container) and in turn provide a storage API to the upper layer of the virtualizer (read, write or delete virtual blocks). The choice of the correct library is done at runtime: stored in extended attributes, the QoS of each file or directory is retrieved by the virtualizer. The latter then parses the list of existing QoS dynamic libraries in the suitable CloViS executable folder to select the appropriate one (the only one able to deal with that QoS) and invoke the corresponding API function.

Client access components

In fact, numerous access components may be developed. So far, we have implemented a distributed POSIX-like filesystem access based on fuse, called CloViSFS. We aim to implement also a block-mode access for Qemu and a RESTful API to show that every access type may be implemented with both efficiency and a full choice of storage QoS. However although being



c) QoS libraries and Virtualizer interactions

a critical point in such a middleware, we are still at a proof-of-concept stage and we do not have all of these access types available to date.

2.2 DESIGN CHOICES

The main components of CloViS having been introduced, some design options deserve to be explained.

First, the use of a user-space middleware instead of kernel-space modules. Despite the numerous context changes it implies and the resulting performance decrease, the easier development and support has lead to this decision.

Secondly, the choice of a block-like access to the virtualized storage space instead of an object-based storage like in Ceph or GlusterFS. In fact, Ceph and GlusterFS have used the latter solution to overcome the problem of metadata coherency and consistency on such a distributed environment: they both use a deterministic hash function to map objects over storage resources. It also helps in implementing copy-on-write storage by storing side-by-side various releases of a single object. However, the CCCC component made both easier and efficient the implementation of such coherency and concurrency mechanisms. Thus, a block mode access was as easier as object storage to implement in this context. The future load tests will show the relevance of our choice.

Thirdly, the use of pluggable QoS libraries. Actually, this is the core of our system and the main difference between CloViS and the others cloud-aware storage systems. Indeed, implementing a new data placement policy in Ceph is very complex or even impossible. In GlusterFS, the stackable translators make it somewhat simpler, but using different components (replicator, distributor, translator) to control data placement does not lead to a smooth and efficient implementation. In CloViS, developing a new data placement just requires the implementation of a simple dynamic library, with a “scalable” VRT API to learn: simple policies just need the basic VRT API while very complex policy may use the full CloViS API (especially the CCOM one).

V USE CASES

As previously mentioned, the aim of CloViS is to provide a framework for QoS-based storage virtualization. We have, as of now, implemented a first proof-of-concept use case. We are going to multiply use case implementations to show the advantages of such implementation and evaluate its performance against existing solutions.

1 A BASIC USE CASE

The first release of the virtualizer implements both synchronous and asynchronous low-level operations (read, write and delete on storage resources). We also have developed a basic QoS library dedicated to full replication over nodes in a synchronized way. In order to access data, we have used a preliminary release of our parallel filesystem CloViSFS.

Based on this first implementation, the read / write / delete operations in CloViS can be detailed. When a query is performed on CloViSFS, the latter transmit the request to the local virtualizer (using Read_Block, Write_Block and Delete_Block). The virtualizer thus receives a buffer, a buffer size and a block identifier made of a virtual organization ID, a logical volume ID, a container ID, an inode ID and a block ID. Using the first three ID (VO, LV and CAN), the virtualizer retrieves the QoS ID of the data. It then retrieves the adequate QoS library (Get_QoS_ID on the list of libraries in the QoS libs folder) and calls its corresponding function. The QoS library function receives the buffer, the buffer size, an internal QoS ID (if appropriate) and the whole block identifier (VO, LV, CAN, inode and block ID). It retrieves the list of storage resources involved in the container structure

using VO, LV and CAN ID by querying the virtualizer (Get_CAN_Info). Based on the internal QoS ID (if appropriate) and the list of storage resources, it decides how to access data and on which nodes. The QoS library relies on the virtualizer to realize these accesses (the sync versions of SR_Read, SR_Write and SR_Delete). In our case, full replication writes are implemented by storing the block received on all storage resources of the container. Full replication reads uses the local storage resource if any, distant SR otherwise.

Note that as shown on this example, the QoS ID is made of two parts: a component level ID (the one used by the virtualizer to access the correct QoS library), and a component-specific QoS ID (used by the library to decide how data management should be implemented for this QoS). The latter may be seen as a parameter given to the QoS library to refine its strategy. For example, when dealing with data availability, the component-specific Qos ID could specify the minimum number of replicas needed to retrieve the original data. Thus, using a container with four storage resources, the library-specific QoS ID could allow the choice between a full replication on all nodes strategy (if QoS ID is 1), stripped storage (if QoS ID is 4) or any intermediate replication level (provided it is supported by the library).

2 A MORE SOPHISTICATED LIB

As previously mentioned, using the component-specific QoS ID is usefull to implement more powerfull QoS libraries. In order to show a simple (but real) use of this ID, we have modified our first QoS library to make it supports two QoS: the previously discussed synchronous full replication (Write_Block returns when data is written on all nodes) or an asynchrous full replication. In this case Write_Block returns as soon as data is written on local storage resources (if any) and transmitted to other nodes, without waiting for the latters to receive and write it. The node initiating the original Write_Block will be notified by the other nodes as soon as data is written on disk in order to detect any failure. This solution may be used when data is frequently written, thus penalizing synchronous operations.

Switching the QoS for a file or a folder is simple: we just need to change the extended attribute relative to the QoS of this filesystem object. The next Block_Write will then use the adequate new version of the low-level storage resources operations.

3 THE BEST IS YET TO COME

Many challenges that need to be addressed regarding cloud storage may find a smart solution using CloViS. This extensible software layer is dedicated to offer a simple and unified view of underlying storage resources with ready-to-use or user-defined quality of service.

For example, QoS libraries dedicated to availability, integrity or confidentiality may be used to achieve data security, depending on the storage requirements. More complex data policy may use a stack of QoS libraries to mix specific data management. Moreover, the cloud user could choose between highly efficient, highly reliable or green-aware storage systems just by selecting the appropriate QoS for its data without any knowledge of the underlying infrastructure. And resources provisioning and billing could be highly simplified using such a middleware.

VI CONCLUSION

Storage security is one of the hard-to-achieve goals of cloud computing, and it is just a first step to a really flexible and efficient storage service. Current solutions focus on specific aspects of distributed storage (e.g. load balancing) without offering an enough flexible frame to encompass all cloud storage requirements.

In this paper we have presented CloViS, the storage virtualization middleware dedicated to cloud computing we have developed. We have shown how quality of service needed for various storage usages can be implemented, and how new data policy requirements may be added on the fly. We have shown that its general architecture makes it possible to consider it as a good candidate for various usages, from data backup purpose to virtual machines backend storage systems.

References

[1] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C. and Quinlan S. 2010. Availability in Globally Distributed Storage Systems. *9th USENIX Symposium on Operating Systems Designs and Implementation*, 61-74.

[2] Sobe, P. and Peter K. 2006. Comparison of Redundancy Schemes for Distributed Stoarge systems. *5th IEEE International Symposium on Network Computing and Applications*, 196-203.

[3] Narayan, S. and Chandy J. A. 2007. Parity Redundancy in a Clustered Storage System. *SNAPI '07 Proceedings of the Fourth International Workshop on Storage Network Architecture and Parallel I/Os*, 17-24.

[4] Juels, A. and Kaliski, B. S. Jr 2007. PORs: Proofs of Retrievability for Large Files. *14th ACM Conference on Computer and Communication Security*, 584-597

[5] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z. and Song, D., 2007. Provable Data Possession at Untrusted Stores. *14th ACM Conference on Computer and Communication Security*, 598-609.

[6] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q. and Fu K. 2003. Plutus: Scalable Secure File Sharing on Untrusted Storage. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 29-42.

[7] Ateniese, G., Fu, K., Green, M. and Hohenberger S. 2006. Improved Proxy Re-encryption Schemes with Applications to Secure Distributed Storage. *ACM Transactions on Information and System Security*, 9(1), 1–30.

[8] Tan, C., Liu, Q. and Wu, J. 2011. Secure Locking For Untrusted Clouds. *4th IEEE International Conference on Cloud Computing*.

[9] Ciriani, V., De Capitani, S., Foresti, S., Jajodia, S., Paraboschi, S. and Samarati, P., 2011. Combining Fragmentation and Encryption to Protect Privacy in Data Storage. *ACM Transactions on Information and System Security,* 13(3), 1-33.

[10] Kher, V. and Kim, Y., 2005. Securing distributed storage: challenges, techniques, and systems. *2005 ACM workshop on Storage security and survivability*, 9-25.

[11] Ortiz, A., Jorda, J., and Mzoughi, A. Toward a New Direction on Data Management in Grids. *15th IEEE International Conference on High Performance Distributed Computing*, 377-378.

[12] Ortiz, A., Thiébolt, F., Jorda, J. and M'zoughi, A. 2009. How to Use Multicast in Distributed Mutual Exclusion Algorithms for Grid File Systems. *Conference on High Performance Computing and Simulation (HPCS 2009)*, 122-130.

[13] Bowers K., Juels A. and Oprea A. 2009. HAIL: A High-Availability and Integrity Layer for Cloud Storage. *ACM Conference on Computer and Communications Security (CCS 2009)*, 187-197.

[14] Weil S. A., Brandt S. A., Miller E. L. Maltzahn C., 2006.CRUSH: Controlled, scalable, decentralized placement of replicated data. Procedings of the 2006 ACM/IEEE Conference on Supercomputing (SC ’06).