Service Oriented ARC Storage; An Overview and Design

Jon K. Nilsen[†,‡] Salman Toor[*] Zsombor Nagy[¶]
Bjarte Mohn[§]
[*]Dept. of Information Technology, Div. of Scientific Computing, Uppsala University,
Box 337, SE-75105 Uppsala, Sweden
[†]University of Oslo, Center for Information Technology,
P. O. Box 1059, Blindern, N-0316 Oslo, Norway
[‡]University of Oslo, Dept. of Physics, P. O. Box 1048, Blindern, N-0316 Oslo, Norway
[§]Dept. of Physics and Astronomy, Div. of Nuclear and Particle Physics,
Uppsala University, Box 535, SE-75121 Uppsala, Sweden
[¶]Institute of National Information and Infrastructure Development
NIIF/HUNGARNET, Victor Hugo 18-22, H-1132 Budapest, Hungary

j.k.nilsen@usit.uio.no salman.toor@it.uu.se zsombor@niif.hu
bjarte.mohn@fysast.uu.se

***Abstract*—There is an ever increasing need to utilize geographically distributed hardware resources, both in terms of CPU and in terms of storage. The service oriented architecture provides a natural framework for managing these resources. The next generation Advanced Resource Connector (ARC) is a service oriented Grid solution that will provide the middleware to represent distributed resources in one simple framework. In this paper, we will present an overview of the ARC storage system, itself a set of services providing a self-healing, flexible Grid storage solution. We will also present some first proof-of-concept test results, with a deployment of the storage system distributed across three different countries.**

I. Introduction

The challenge of building a reliable, self-healing, fault-tolerant, consistent data management system in a web scale is an interesting task. Making the system work in a heterogeneous, distributed environment like the Grid is even more interesting. An increasing number of applications demand not only increased CPU power, but also vast amounts of storage space. The required storage space is not only restricted to the duration in which the application runs; the data should often be available for years afterwards, in certain cases even for decades. Nowadays, we can easily find single Grid jobs which produce gigabytes or even terabytes of data, ramping up the requirements of storage systems to the petabyte scale and beyond. Hard drive capacity is still increasing and storage raids can handle failing hard drives through data replication. However, even a designated storage site cannot guarantee 24/7 availability or long term security against catastrophic losses of data. The need for a distributed, self-healing storage system is evident. To make the storage system useable to Grid users, the system must provide reliable and secure file transfer protocols, a cataloging system and secure storage of the data. Several projects and designs have emerged to address such challenges [1], [2].

In the advent of the next generation of the Advanced Resource Connector (ARC) Grid middleware [3] (new release due during fall 2009), we present the ARC storage system [4]. This distributed storage system is designed to provide an easy to use, flexible and scalable system that can offer native storage and at the same time provide access to third-party solutions like dCache [5], [6] by using the same, uniform interface. Being part of ARC, the storage system is based on a service oriented architecture, in which each major component of the system runs as a separate service within the ARC Hosting Environment Daemon (HED) [7]. The HED service container gives the capability of flexible replacement of the components as well as the possibility to introduce modifications in the future.

This paper is organized as follows. After describing related work in Section II, we give a bird's eye view of the next generation Advanced Resource Connector (ARC) in Section III. An overview of the ARC Storage is given in Section IV, while the architecture of the storage system is elaborated in Section V. In Section VI we give some early, proof-of-concept results, before concluding in Section VII.

II. Related Work

dCache: dCache is a storage system which combines heterogeneous storage elements to collect several hundreds of terabytes in a single namespace. It additionally supports standard and native protocols like gridftp, srm, dcap and gsidcap. dCache is a joint effort between DESY [8], Fermi National Accelerator Laboratory [9], the Nordic Data Grid Facility (NDGF) [10] and several other collaborators. dCache has proven to be a very stable and scalable solution. However, it is relatively difficult to deploy and integrate with new applications. The ARC Storage, being a light-weight and flexible storage solution, aims more towards new user groups less familiar with Grid solutions.

BigTable: Bigtable is a distributed storage system managing structured data on the petabyte scale [11]. It is currently used within Google in projects like web indexing, Google Earth and Google Financing. Bigtable has several interesting features, one of which is the distributed lock service, Chubby [12]. Chubby is a high-availability, distributed locking service sitting on top of the B^+-tree architecture of Bigtable. Chubby has several features similar to our A-Hash service (see Section A.4), among which the use of the Paxos algorithm [13], [14], is the most striking. A major caveat for the Grid community is that Bigtable is neither free, open-source nor available to the public.

Storage Resource Broker: Based on the client-server model, the Storage Resource Broker (SRB) [15], [16] provides a flexible data grid management system. It allows uniform access to heterogeneous storage resources over a wide area network. Its functionality, with a uniform namespace for several Data Grid Managers and file systems, is quite similar to the functionality offered by our Gateway service (see Section B.6). However, being built as a middleware on top of other major storage solutions, SRB does not offer its own storage solution.

Scalla: Scalla is a widely used software suite consisting of an xrootd server for data access, and an olbd server for building scalable xrootd clusters [17]. Originally developed for use with the physics analysis tool root [18], xrootd offers data access both through the specialized xroot protocol and through other third-party protocols. The combination of the xrootd and olbd components offers a cluster storage designed for low latency high bandwidth environments. In contrast, ARC Storage is optimized for high latency more suitable for the Grid environment.

III. The Advanced Resource Connector

The next generation of Advanced Resource Connector (ARC) Grid middleware is developed by NorduGrid [19] and the EU KnowARC project [20]. It consists of a set of pluggable components. These components are the fundamental building blocks of the ARC services and clients. ARC services run inside a container called the Hosting Environment Daemon (HED) and there are four kinds of pluggable components with well defined tasks: Data Management Components are used to transfer the data using various protocols, Message Chain Components are responsible for the communication within clients and services as well as between the clients and the services, ARC Client Components are plug-ins used by the clients to connect to different Grid flavors, and Policy Decision Components are responsible for the security model within the system.

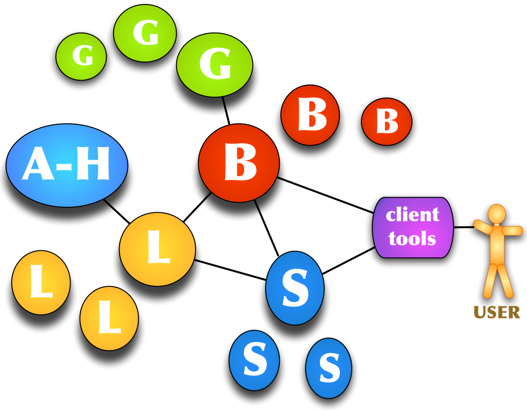

Fig. 1. Schematic of the ARC Storage architecture. The figure shows the main services of ARC Storage and the communication channels. The B stands for Bartender, the S Shepherd, the L Librarian, the G Gateway and the A-H stands for A-Hash. The straight lines denotes the communication between services.

To deliver the non-trivial quality of services required by the Grid, there are a number of services running inside the HED. For example, Grid job execution and management is handled by the A-REX service [21], policy decisions are taken by the Charon service, the ISIS service is responsible for the information indexing, batch job submission is handled by the Sched service, etc. In the later sections our discussion will focus on the architecture and the design of another major set of services, i.e., the ARC storage.

IV. The ARC Storage System

The ARC Storage consists of a set of SOAP based services residing within HED. Together, the services provide a self-healing, reliable, robust, scalable, resilient and consistent data storage system. Data is managed in a hierarchical global namespace with files and subcollections grouped into collections[1]. A dedicated root collection serves as a reference point for accessing the namespace. The hierarchy can then be referenced using Logical Names. The global namespace is accessed in the same manner as in local filesystems.

Being based on a service oriented architecture, the ARC Storage consists of a set of services as shown in Fig. 1. The services are as follows: The Bartender (B) provides the high-level interface to the user; the Librarian (L) handles the entire storage namespace, using the A-Hash (A-H) as a metadatabase; the Gateway (G) provides access to third-party storage systems; and the Shepherd (S) is the frontend for the physical storage node. See Section V for a detailed discussion of the different services. The services communicate with each other through the Message Chain Components in HED. The communication channels are depicted by straight lines in Fig. 1.

The system supports file transfer through several transfer protocols, with client side tools that hide various technical details such as protocol specification, port numbers

[1] A concept very similar to files and directories in most common file systems.

and so on. To provide fault-tolerance, the system implements automatic file replication, where the replicas of a file[2] are always stored on separate storage nodes[3]. To ensure a resilient, self-healing system, the Shepherd regularly sends heartbeats to one of the Librarians. If the Shepherd fails to send a heartbeat, one of the Librarians will automatically initiate re-replication of the replicas between the other Shepherds.

In the default implementation, the services will provide a full-featured and consistent data storage system using files as an atomic unit. The scope of the ARC storage system in the foreseen future is restricted to provide support for file-based data services.

The ARC Storage supports third-party storage services in two ways. Using the Gateway service, files already stored in some third-party storage can be accessed using the client tools provided by the ARC Storage. To make use of third-party services for storing new files, with the replication and fault-tolerance offered by the ARC Storage, third-party storage elements can also be used as backends for the Shepherd service.

External grid middleware components can access the ARC storage system by using ARC data service interfaces directly. ARC will also provide interface components that communicate via standard protocols like SRM, which will provide a single access point to the whole system.

V. Architecture of the ARC Storage

In a service oriented architecture the role of each service is well defined. Available objects[4] in the system are identified by unique global IDs. These IDs are categorized according to the object type:

- Each file and collection has a unique ID (GUID).
- Services are uniquely identified by a serviceID.
- The Shepherds in the system identify their files by a referenceID.

Each object in the ARC Storage has a globally unique ID. A collection contains files and other collections, and each of these entries has a name unique within the collection very much like entries in a standard directory on a local filesystem. Besides files and collections the ARC Storage has a third type of entry called mount-points, which are references to the third-party storages within the global namespace.

Replicas in a distributed storage system can have different states; they can be broken, deleted, partially uploaded, etc. In the ARC Storage, all replicas have assigned a state, some of which are '**alive**' (if the replica passed the checksum test, and the Shepherd reports that the storage node is healthy), '**invalid**' (if the replica has wrong checksum, or the Shepherd claims it has no such replica) and '**creating**' (if the replica is in the state of uploading).

In the following the details of the architecture are presented in three parts: First we will discuss the details of the core components, second we will discuss some of the important features provided by the system, and third we will discuss the security model.

A. Core Components

A.1 Bartender

The Bartender service provides a high-level interface for the storage system. Clients connect to the Bartender to create and remove files, collections and mount-points using their Logical Names. The Bartender communicates with the Librarian and Shepherd services to execute the clients' requests. However, the actual file data does not go through the Bartender, instead file transfers are directly performed between the storage nodes and the clients. There could be any number of independent Bartender services running in the system, providing high-availability and load-balancing.

A.2 Librarian

The Librarian manages the hierarchy and metadata of files, collections and mount points, as well as the health information of the Shepherd services. In addition, the Librarian handles the information about registered Shepherd services. The Librarian receives heartbeat messages from the Shepherds and changes replica states automatically if needed. The Librarian uses the A-Hash for consistently storing all metadata. This makes the Librarian a stateless service, thus enabling the system to have any number of independent Librarian services, again providing high-availability in the system.

A.3 Shepherd

The Shepherd services run as front-ends on storage nodes. A Shepherd service reports to a Librarian about the node's health state in terms of replicas. While the Bartender initiates file transfers, the actual transfers go directly between the Shepherd and the clients.

When a new replica upload is initiated, the Shepherd generates a referenceID which refers to the replica within that Shepherd. Each Shepherd has a unique serviceID, so with these two IDs the replica can be unambiguously referenced. This is called a Location of the replica.

A.4 A-Hash

A-Hash is a hash table for consistently storing data in property-value pairs. All metadata about files, collections, mount point, Shepherd's health status, and so forth, is stored in the A-Hash. As the A-Hash is the service storing the entire state of the storage system, it is absolutely crucial for the ARC Storage that the A-Hash is consistent. The distribution and replication of this service is therefore both necessary and challenging.

[2] In this paper, a file denotes an entry in the global namespace, while replica denotes the physical file stored on a storage node.

[3] A storage node is a server with a Shepherd, a Shepherd backend and some storage service.

[4] Files, collections, mount points, etc.

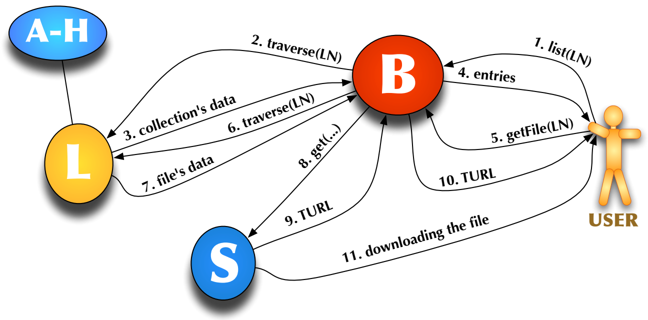

Fig. 2. Schematic showing the scenario of file download from the ARC Storage.

B. Features

B.1 Heartbeat monitoring

In the proposed architecture, each Shepherd periodically sends heartbeats to a Librarian with information about replicas whose state changed since the last heartbeat, and the corresponding GUIDs. These heartbeats are then stored in the A-Hash, making them visible to all the Librarians in the system. If any of the Librarians notices that a Shepherd is late with its heartbeat, it will invalidate all the replicas in that Shepherd.

B.2 Replication

Shepherds periodically ask the Librarian if the file of the replica stored on its storage node has enough replicas. If the file does not have enough replicas, the Shepherd informs the Bartender and the Bartender initiates a put request that returns a Transfer URL to the Shepherd. The Shepherd finally uploads the new replica. The Shepherds which gets the new replica notifies the Librarian that the replica is alive. The Librarian then add this to the corresponding file.

B.3 Deletion

In a replicating storage system, there are several possible solutions for file deletion, since both the replicas and the metadata need to be removed. In our solution, we use the process of *lazy deletion*, [22]. When a client requests that a file should be deleted (and the user has the proper permissions) the Bartender instructs the Librarian to remove the file's GUID from the A-Hash. When the Shepherd, periodically checking all its replicas, discovers that a replica has no file (and hence, no Location), it will automatically delete the replica.

B.4 Fault tolerance

Due to the unpredictable nature of the Grid environment, it is essential to have some degree of automatic recovery system in case of unexpected failures. Fault tolerant behavior is required both at the level of metadata and on the level of physical storage. While the work on fault-tolerant metadata is still in progress, the following two scenarios will explain the currently available recovery mechanism for physical storage:

- In the case of a file having invalid checksum, the Shepherd immediately informs the Librarian and the Librarian changes the state of the given replica to '**invalid**'. To recover its replica, the Shepherd contacts a Bartender and asks for another replica of the file. The Bartender chooses a valid replica, initiates a file transfer from a Shepherd having the replica, and returns the TURL to the Shepherd with the invalid replica. When the Shepherd has recieved the replica and compared the checksum, it notifies the Librarian that the replica is alive again.
- In the case of a Shepherd going offline, a Librarian will, as mentioned earlier, notice the lack of heartbeats and invalidate all the replicas, initiating new replication for all the files stored in this storage node. However, if the Shepherd again comes online, there will evidently be more replicas than needed. The first Shepherd to notice this will set its replica's state to '**thirdwheel**', i.e., obsolete. At the next occasion, the Shepherd will remove the replica, if and only if it has the only '**thirdwheel**' replica of this file. If there are more replicas with this state, all replicas will be set back to '**alive**' and the process is repeated. This scenario will be discussed further in Section VI.

B.5 Client tools

Being the only part a user will (and should) see from a storage system, the client tools are an important part of the ARC Storage. Currently ARC supports two ways of accessing the storage solution:

- The **Command-line Interface** (CLI) provides access to the storage through the methods `stat`, `makeCollection`, `unmakeCollection`, `putFile`, `getFile`, `delFile`, `list` and `move`. Methods for modifying access and ownership will be available in the near future. The CLI assumes a relatively high level of computer competence from the user. However, the CLI, being a stand-alone tool, can be used to access ARC Storage from any computer (also including Microsoft Windows PCs) that has network access and a Python installation.
- The **FUSE module** provides a high-level access to the ARC Storage. Filesystem in Userspace (FUSE) [23] provides a simple library and a kernel-userspace interface. Using FUSE and the ARC Storage Python interface, the FUSE module allows users to mount the ARC Storage namespace into the local namespace. This way, the user can use her/his favorite file manager to access her/his files and collections. The FUSE module provides most of the features provided by the CLI, with the exception of modifying some non-posix metadata.

Its worth mentioning that the client tools access the storage system through the Bartender only. Currently upload and download is realized through HTTP(S), but there are plans to add support for other protocols, such as SRM and

GridFTP.

B.6 Gateways

Gateways are used to communicate with the external storage managers. While designing this service, care was taken to:

- Retain the transparency of the global namespace while using the external storage systems.
- Develop a protocol oriented service i.e., all the external storage managers which support a certain protocol should be handled using the corresponding gateway service.

This approach provides flexibility while avoiding multiple Gateway services for different storage managers. Currently, the available Gateway service is based on the gridftp protocol.

When the user request is made, the Librarian provides the metadata related to the request to the Bartender. Requests can be related to files, collections or external mount-points. In case of creating external mount-points, the Bartender contacts the Librarian to store the mount-point for later use. In the case where a request is related to the downloading of files from the external store, the Gateway service first checks the status of the file and then sends the Transfer URL (TURL) to the client via the Bartender. Using this TURL, the client can directly get the file from external store.

C. Security Model

As is the case for all openly accessible web services, the security model is of crucial importance for the ARC Storage. The security architecture of the storage can be split into three parts; the inter-service authorization; the transfer-level authorization; and the high-level authorization.

- The **inter-service authorization** maintains the integrity of the internal communication between services. There is a lot of communication between the services of the ARC Storage. The Bartenders send requests to the Librarians and the Shepherds, the Shepherds communicate with the Librarians and the Librarians talk with the A-Hash. If any of these services is compromised or a new rogue service gets inserted in the system, the security of the entire system is compromised. To enable trust between the services, they need to know each other's Distinguished Names (DNs). This way, a rogue service would need to obtain a certificate with that exact DN from some trusted Certificate Authority (CA).
- The **transfer-level authorization** handles the authorization in the cases of uploading and downloading files. When a transfer is requested, the Shepherd will provide a one-time Transfer URL (TURL) which the client can connect to. In the current architecture, this TURL is world-accessible. This may not seem very secure at first. However, provided that the TURL has a very long, unguessable name, that it is transfered to the user in a secure way and that it can only be accessed once before it is deleted, the chance of being compromized is very low.
- The **high-level authorization** considers the access policies for the files and collections in the system. These policies are stored in A-Hash, in the metadata of the corresponding file or collection, providing a fine-grained security in the system.

The communication with and within the storage system is realized through HTTPS with standard X.509 authentication.

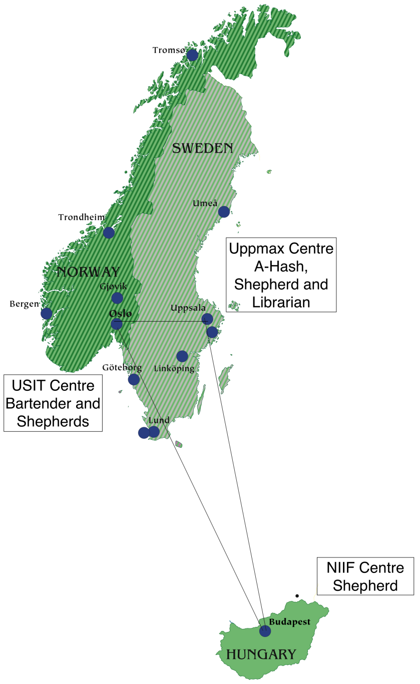

Fig. 3. The map shows the geographical distribution of the services in the test setup.

VI. Testing and Discussion

Even though the ARC Storage is in a pre-prototype state, it is already possible to deploy and use the system for testing purposes. To properly run a proof-of-concept test, the resources need to be geographically distributed. In our test scenarios we utilized resources in three different countries.

In our test deployment, we used two nodes from Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX), Sweden, three nodes from the Center for Information Technology (USIT) at the University of Oslo, Norway, and one node from National Information Infrastructure Development Institute (NIIF) Hungary.

The services were distributed as shown in Fig. 3:

- An A-Hash runs at UPPMAX.
- A Bartender runs at USIT.
- A Librarian runs at UPPMAX.
- In total five Shepherds were used for the tests: Three at USIT, having 100GB storage space each, one at UPPMAX, with 20GB, and one at NIIF, providing 16GB of storage space.

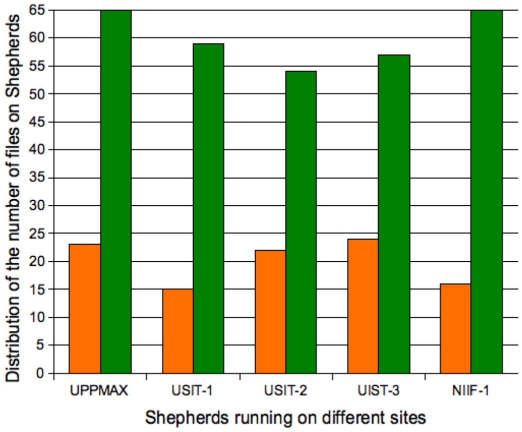

Fig. 4. The distribution of replicas over the geographically distributed storage nodes. Green bars shows the distribution in the case where we upload 100 files requesting 3 replicas where as orange bars show the distribution after uploading 100 files without replication.

The following test where carried out:

- Test 1: The distribution of data after uploading 10 large size (1 GB) files with one replica each.
- Test 2: The distribution of files after uploading 100 small size (1 kB) files with one replica each, and with simultanious uploading
- Test 3: The distribution of data after repeating Tests 1 and 2 with three replicas.
- Test 4: System behavior while some of the shepherd becomes offline.

All the tests were performed and worked as expected. However, some of the tests deserve extra attention.

Fig. 4 illustrate two test results, corresponding to Test 2 and Test 3: The orange bars show the distribution after uploading 100 small files simultaneously, without replication. This gives an indication of how the system balances the load of many clients uploading at the same time. The green bars show distribution with the same kind of uploading, but with the clients requesting three replicas for each file. The difference here is that the system simultaneously has to handle both replication and the clients requesting uploads. When the Bartender chooses a location for which to put a replica, it generates a list of Shepherds that (a) do not have a replica of the file, and (b) is not already in the process of uploading the replica. Next, it draws a Shepherd from the list at random, using a uniform random number generator. When uploading without asking for replicas we would therefore expect a relatively flat distribution of the files, as can be seen in Fig. 4.

	UPPMAX	USIT-1	USIT-2	USIT-3	NIIF
Load(MB)	6683	7638	7650	4773	2289

TABLE I

Overall load distribution on the storage nodes after Tests 1, 2 and 3 were finished

Table I shows the disk usage after Tests 1, 2 and 3. Worth noticing here is that the bandwidth NIIF was significantly lower than between UPPMAX and USIT. If the bandwith is saturated and the storage node is busy with, e.g., checksumming a large file, the heartbeat from this Shepherd may be delayed, causing the Bartender not to choose this Shepherd for the next replica. This may explain the relatively low storage load on the NIIF storage node.

	Before (MB)	During (MB)	After (MB)
UPPMAX	4888	8799	8798
USIT-1	7820	9778	6843
USIT-2	6843	-	4888
USIT-3	7820	9774	8798
NIIF	3320	2344	1367
Sum	30691	30695	30694

TABLE II

Storage load before, during and after a Shepherd outage.

A significant feature of the ARC Storage is its automatic self healing. Test 4 addresses this feature by studying the effect of taking out one Shepherd from the system, and later reinsert it. Table II shows the storage distribution in three states: Before interrupting a Shepherd, after the USIT-2 Shepherd is interrupted and redistribution of replicas is finished; and alive, when the Shepherd is restarted and the system again has stabilized. We can see that all files are properly distributed between the remaining Shepherds when one of them is disrupted and that the storage load evens out again when the Shepherd comes back online. When the system discovers that there are more replicas than needed, the first Shepherd noticing will set mark its replica as obsolete. Since the failing Shepherd didn't lose any replicas while being down, redistribution of replicas is just a matter of deleting obsolete replicas. We also see that the NIIF node actually got fewer files when USIT-2 went offline. If two Shepherds simultaniously starts uploading replicating a missing replica, there will be too many replicas. Here, a big replica on the NIIF node has randomly been chosen as obsolete. However, the total storage usage remains the same in all three cases.

VII. Conclusion and Future Work

The proposed system is still in an early phase of development, but our test results depict that the architecture is strong enough to handle the challenges for distributed large scale storage. Much effort is required to make the system production ready. However, we believe that continuing in the same direction will enable us to provide a persistent and flexible storage system which can fulfill the needs of the scientific community.

Some key areas need special attention and effort to make the proposed storage system even more stable, reliable and consistent:

- **Security:** This is still under development. The design is more or less ready, but it still needs to be implemented and properly tested.

- **The A-Hash**: This is currently centralized. To avoid a single point of failure in the system, and to improve the system performance, the A-Hash needs to be distributed.
- **Performance optimization:** To make a storage system ready for production, one needs to discover and improve on possible bottlenecks within the system. As soon as the A-Hash is distributed, other possible bottlenecks, such as load-balancing and self-healing mechanisms, will be investigated.
- **Protocols:** To ease the interoperability with third-party storage solutions and clients, the system needs to support storage protocols such as SRM and GridFTP. In addition the system will come with its own ARC protocol.

Even though we have more work in front of us, we have shown that the ARC Storage already is in a state where initial real-life tests can be done. We have described a simple, yet strong architecture which we believe will benefit communities in need of a lightweight, yet distributed storage solution.

VIII. Acknowledgements

We wish to thank Mattias Ellert for helpful discussions and guidance on the ARC middleware. In addition, we like to thank UPPMAX, NIIF and USIT for providing resources for running the storage tests.

The work has been supported by the European Commission through the KnowARC project (contract nr. 032691) and by the Nordunet3 programme through the NGIn project.

References

[1] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger, "Data management in an international data grid project." Springer-Verlag, 2000, pp. 77–90.

[2] Y. Deng and F. Wang, "A heterogeneous storage grid enabled by grid service," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 1, pp. 7–13, 2007.

[3] M. Ellert *et al.*, "Advanced Resource Connector middleware for lightweight computational Grids," *Future Gener. Comput. Syst.*, vol. 23, no. 1, pp. 219–240, 2007.

[4] Z. Nagy, J. K. Nilsen, and S. Toor, *Documentation of the ARC storage system*, NorduGrid, NORDUGRID-TECH-17. [Online]. Available: http://www.nordugrid.org/documents/arc1-storage-documentation.pdf

[5] M. de Riese, P. Fuhrmann, T. Mkrtchyan, M. Ernst, A. Kulyavtsev, V. Podstavkov, M. Radicke, N. Sharma, D. Litvintsev, T. Perelmutov, and T. Hesselroth, *dCache Book*. [Online]. Available: http://www.dcache.org/manuals/Book/Book-a4.pdf

[6] G. Behrmann, P. Fuhrmann, M. Grønager, and J. Kleist, "A distributed storage system with dcache," *Journal of Physics: Conference Series 119 (2008) 062014*.

[7] D. Cameron, M. Ellert, J. Jönemo, A. Konstantinov, I. Marton, B. Mohn, J. K. Nilsen, M. Nordén, W. Qiang, G. Rőczei, F. Szalai, and A. Wäänänen, *The Hosting Environment of the Advanced Resource Connector middleware*, NorduGrid, NORDUGRID-TECH-19. [Online]. Available: http://www.nordugrid.org/documents/ARCHED_article.pdf

[8] dCache project. [Online]. Available: http://www.dcache.org

[9] Fermi National Acelerator Laboratory. [Online]. Available: http://www.fnal.gov

[10] Nordic DataGrid Facility. [Online]. Available: http://www.ndgf.org/

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 205–218.

[12] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation.* Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[13] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.* New York, NY, USA: ACM, 2007, pp. 398–407.

[15] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The sdsc storage resource broker," in *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 1998, p. 5.

[16] M. Wan, A. Rajasekar, R. Moore, and P. Andrews, "A simple mass storage system for the srb data grid," in *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03).* Washington, DC, USA: IEEE Computer Society, 2003, p. 20.

[17] C. Boeheim, A. Hanushevsky, D. Leith, R. Melen, R. Mount, T. Pulliam, and B. Weeks, "Scalla: Scalable cluster architecture for low latency access using xrootd and olbd servers," Stanford Linear Accelerator Center, Tech. Rep., 2006.

[18] R. Brun and F. Rademakers, "Root – an object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1-2, pp. 81–86, April 1997. [Online]. Available: http://dx.doi.org/10.1016/S0168-9002(97)00048-X

[19] NorduGrid Collaboration. [Online]. Available: http//:www.nordugrid.org/

[20] EU KnowARC project. [Online]. Available: http://www.knowarc.eu/

[21] A. Konstantinov, *The ARC Computational Job Management Module - A-REX*, NorduGrid, NORDUGRID-TECH-14. [Online]. Available: http://www.nordugrid.org/documents/arex_tech_doc.pdf

[22] P. Celis and J. Franco, "The analysis of hashing with lazy deletions," *Inf. Sci.*, vol. 62, no. 1-2, pp. 13–26, 1992.

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