Mucura: your personal file repository in the cloud

F Hernandez[1,2], W Wu[2], R Du[2], S Li[2] and W Kan[2]

[1]IN2P3/CNRS Computing Center, 43 Bd du 11 Novembre 1918, 69622 Villeurbanne Cedex, France

[2]IHEP Computing Center, P.O Box 918-7, 19B Yuquan Road, Beijing 100049, China

E-mail: fabio@in2p3.fr

Abstract. Large-scale distributed data processing platforms for scientific research such as the LHC computing grid include services for transporting, storing and processing massive amounts of data. They often address the data processing needs of a virtual organization but lack the convenience and flexibility required by individual users for their personal data storage needs. This paper presents the motivation, design and implementation status of Mucura, an open source software system for operating multi-tenant cloud-based storage services. The system is specifically intended for building file repositories for individual users, such as those of the scientific research communities, who use distributed computing infrastructures for processing and sharing data. It exposes the Amazon S3-compatible interface, supports both interactive and batch usage and is compatible with the X509 certificates-based authentication mechanism used by grid infrastructures. The system builds on top of distributed persistent key-value stores for storing user's data.

1. Introduction

By aggregating the storage capacity of hundreds of sites around the world, distributed data processing platforms, such as the LHC computing grid [1], offer services for transporting, storing and processing experimental data. On top of those services, expert users of large virtual organizations can successfully build sophisticated workflows by exploiting the possibilities of the often complex APIs and command line interfaces they expose.

However, from our perspective, average users need specific services and more convenient tools for implementing their individual storage-related workflows. In particular, scientists would benefit from flexible, always-on storage service accessible both interactively from their personal computer and in batch from their grid jobs.

With this goal in mind, we developed a prototype system for operating multi-tenant cloud-based storage services. The system enables data centers to operate extensible, always-on, personal file repositories for addressing the storage needs of individual users.

Mucura (pronounced *mükürə*) is the Spanish name of a clay container, a sort of amphora, used for storing beverages, water and cereals and employed for funeral rites of pre-Colombian ethnic groups in

Colombia and other American countries. It conveys the notion of a container for personal items we aim at in this work.

In this paper we report on our experience prototyping the system. In the next section we present the requirements for the system. In section 3 we mention related work relevant for this project. Section 4 and section 5 present the system overview and some details of its current implementation. Finally, in section 6 we expose the planned work and conclude in section 7.

2. System Requirements

2.1. Non-functional requirements

Two basic principles guided us for doing this work: the target system must be *convenient for end-users* and *easy to operate for service providers*.

In this context, convenience for end-users encompasses several aspects. First, the service must be easy to use from a connected personal computer, using familiar metaphors and tools, such as file browsing and drag-and-drop. Second, grid jobs must be able to exploit the users' individual storage area on their behalf. Finally, the system must ensure that from the users' perspective, their data is highly available.

Regarding service providers, the system must require a reduced amount of manpower for operations and must tolerate some level of failure. Disk or whole node failures must not trigger urgent human interventions. Ideally, the system should be cost effective by using commodity hardware components and should easily scale out by adding more hardware to accommodate increased storage demand. Finally, it has to provide information detailed enough for monitoring, accounting and traceability purposes.

2.2. Functional requirements

The set of operations provided by Mucura to the end-users is deliberately limited. The system must allow its users to upload and download files, to delete files and to list the contents of their own repository. In addition, users must be able to selectively share files with other individuals, registered users of the system or otherwise.

Mucura is not designed to be a general-purpose storage system and does not intend to support I/O-intensive applications nor to provide POSIX semantics. However, we are convinced that the basic functionality it provides can satisfy a wide range of storage needs of individuals from the scientific research communities.

We deliberately don't focus on performance for this system. Users should however be able to upload and download files at rates fast enough for the system to be usable in an interactive mode. Although performance optimization has not been part of our work so far, we would expect transfer performance of 2-3 MB/sec to keep interactivity at a reasonable level.

2.3. Scale

Given the personal characteristic of the storage area, each individual will use his repository in a different way. We expect individuals will use their repository for storing files of sizes ranging from a few kilobytes up to possibly 1 or 2 gigabytes. The number of files will also vary among individuals, but we expect typical users to store a few thousand files in their repository. Finally, a typical data center (or grid computing center) would provide this service up to a few thousand individual users.

3. Related work

In the research community, tools such as Chirp [2] have addressed the need for an unprivileged end-user to autonomously make personal storage areas accessible from remote locations. However, it does not address how the storage backend is implemented or operated.

Several commercial services have emerged in the last few years offering cloud-based storage. They address several use cases relevant for individuals, such as file archiving and backup, file sharing or file synchronization among several connected computers. Examples of such services are Amazon Simple Storage Service (S3) [3], Dropbox [4], Google's GoogleDrive [5], Microsoft's SkyDrive [6], Rackspace's CloudFiles [7] and SugarSync [8] to name a few.

Some of those services are very popular, have hundreds of thousands of registered users and manage billions of stored objects. Amazon reports that at the end of the first quarter of 2012 there were 905 billion objects stored on S3 [9] and more than 1 billion objects are added daily. Dropbox reports more than 50 million registered users and stores 500 million files daily [10]. Those figures show the interest in cloud storage services and we think that such a service would also benefit scientists.

Several technologies, often proprietary, are used to implement cloud storage services. Amazon S3, the precursor of this field, is built upon a proprietary key-value structured storage system known as Dynamo [11]. Core ideas of Dynamo are implemented in open-source, general-purpose, distributed data stores such as Cassandra [12], Project Voldemort [13] and Riak [14].

The OpenStack consortium also took ideas from Dynamo for developing Swift, an open-source implementation of a distributed object store. Swift exposes its specific REST API and uses the local file system of the participating nodes to store the object contents and metadata. It implements its own mechanism for replicating files and maintaining data consistency between nodes [15].

Some hardware manufacturers have conducted exploratory work for building storage appliances based on the concepts of Amazon's Dynamo [16]. Commercial closed-source software products have recently appeared for deploying cloud-based storage, such as Basho's CloudCS [17].

Considerable effort has been invested for improving the core technologies of distributed data stores. As a result, several open source implementations of distributed databases have recently appeared that could be suitable for the purposes of our work.

However, currently there is no open source software system for deploying cloud-based storage backed by distributed data stores. With Mucura we aim at exploring this area and at providing a software solution to scientific communities. Specifically, we intend to leverage the work done by the distributed databases community by building the system on top of a third-party, highly-available, distributed data store and we want the system to expose a widely supported interface.

4. System Overview

In this section we present an architectural overview of the system and the expected benefits for individual end-users and service operators.

4.1. System Architecture

The architecture of the Mucura system follows the client-server model. The client runs either on the personal computer of the end-user or on a worker node of a grid site. Client-side software allows the users to act on their remote files in an interactive mode or from a grid batch job.

Server-side software runs on machines operated by the service provider, typically a data center. The files stored in the user repositories are physically located on machines operated by the service provider. Client and server are connected through a local or wide area network.

The end-users' view of their file repository is presented in figure 1. They see a storage area they can manage and use as if it was local to their personal computers.

4.2. System Interface

Mucura exposes to its clients a subset of the Amazon S3 REST API. The decision to be S3-compatible is motivated by the fact that this API is widely supported by several tools, both GUI- and CLI-based, either open-source or commercial. End-users can choose the most convenient tool for their particular needs and working environment.

Support to the SNIA's Cloud Data Management Interface (CDMI) [18] could be added in the future if needed. Currently, there is a noticeable lack of client tools supporting this interface so this work is not considered as a priority at this time.

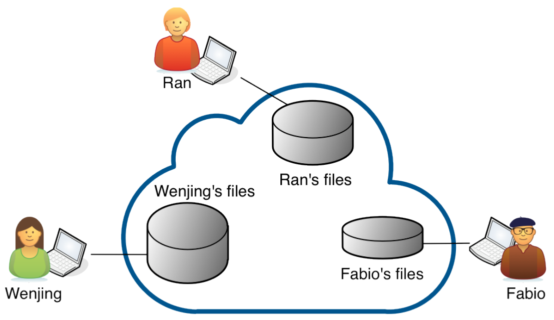

Figure 1. Users' view of their always-on personal file repository. Each user can remotely access his own files from his personal computer.

4.3. Storage backend
As presented in the previous section, we intend to explore the capabilities of third-party distributed data stores as the main backend of Mucura. Some of the available implementations of those stores have interesting features that make them attractive for our project.

Firstly, they are distributed by design and can exploit commodity hardware that is assumed subject to failure. When failures of components or network partitions occur, the system continues its operation without requiring immediate human intervention. Secondly, by design those databases store several copies of each piece of data on different nodes and provide tunable levels of redundancy to accommodate diverse application-specific use-cases. Thirdly, the storage capacity of the system can be easily increased by dynamically adding more hardware: the storage backend will redistribute the data to rebalance the workload among the available machines. All of these features are of special interest for building a highly available file-oriented storage service.

4.4. Benefits
Before presenting the implementation status of the system, we summarize here the benefits to end-users and to operators of the service. Benefits to end-users include:

- The files are stored in a remote location and can be accessed from any connected computer
- Users are entirely free to organize their storage space as they desire
- Grid jobs can interact with the individual repository on the user's behalf using standard grid authentication mechanisms
- Users can share files with other individuals, registered users or otherwise (for instance, to publicly share files with anonymous users)

Among the benefits to service providers, we can mention:

- No special storage nor connectivity hardware is required, so commodity hardware can be used making the system cost-effective
- More storage servers can be added to accommodate increased storage demand
- Files are stored redundantly to make them available to users even in case of hardware failures or network partitions
- No need for additional backup copies of the hosted repositories, since several copies of each file are present in the system at any instant
- As grid jobs identify themselves with X509-based credentials, no additional identification infrastructure is needed to support grid jobs
- The system generates records for traceability, monitoring and accounting purposes

5. Implementation
Figure 2 presents a high-level view of the major components of the system. In the following paragraphs we briefly present those components and their implementation status.

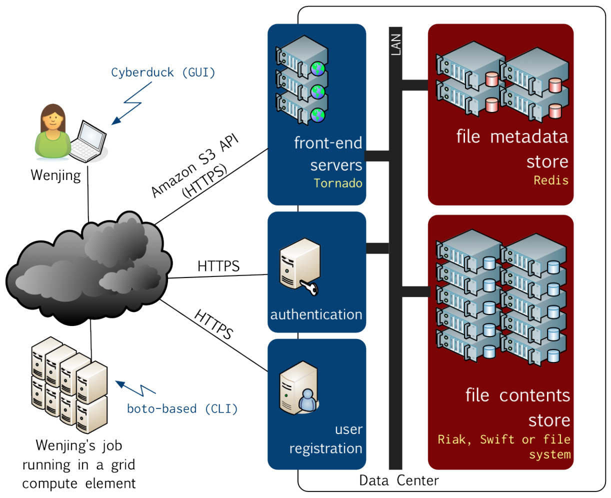

Figure 2. High-level view of the major components of the Mucura system. The metadata store and the file contents store are accessible to the users through the front-end S3-compatible servers. The software packages used for implementing each component of the system are shown.

5.1. Client

The GUI client is not specific to Mucura. Instead, we leverage existing clients compatible with the Amazon's S3 REST API exposed by Mucura. In particular, for our testing we use Cyberduck [19], an open-source implementation that runs on several operating systems.

We also developed a proof-of-concept of a command line interface on top of boto [20], a Python library for implementing clients of Amazon's S3-compatible services. The purpose of providing this interface is to allow grid jobs to interact with the user's repository on his behalf. Since the Amazon S3 authentication mechanism does not natively support grid credentials, we added a specific component for retrieving Amazon S3-compatible credentials from grid credentials. See the *Authentication* subsection below for details.

5.2. User registration

The registration service delivers 3 pieces of information necessary for the end-user to start using the system. Specifically, it provides a user identifier, a private key and the URL of the entry point to the user's individual repository.

We implemented a proof-of-concept of a web service that extracts the user's identity from his personal X509 certificate and registers him as a Mucura user. The duration of the registration is limited in time and cannot exceed the expiration date of the user's certificate. To continue using the service after their personal certificate is expired, enrolled users are required to extend their registration when they acquire a new personal certificate, typically once per year.

5.3. Authentication

As mentioned before, Mucura exposes an API that is compatible with Amazon's S3 REST API. Authentication in S3 is based on both public user identifier and a secret shared between the client and server, namely the user's secret key. Every HTTP request the client sends to the server contains the user identifier and a signature of the request, generated using the client's secret key. In turn, the S3 server computes the request signature using the secret key associated to the identifier of the requestor. Both the request signature sent by the client and the one computed by the server must match in order for the request to be considered legitimate and to be processed [21].

In order to be fully compatible with third-party S3 clients, the Mucura front-end server implements the same authentication mechanism. Although this mechanism is convenient for an interactive use, it is not appropriate for grid jobs. For this mechanism to work, batch jobs executing on worker nodes would need to have the user's secret key embedded. By doing so, the user's secret key would not be secret anymore.

To make file repositories usable by grid jobs, we implemented a specific authentication service that is only used by Mucura clients running in the context of a job. The role of this adaptor authentication service is to deliver a time-limited secret key from a valid grid proxy. The delivered secret key has an associated validity period that does not exceed the duration of the grid proxy originally used to generate it. Command-line clients cache the temporary secret key in the file system of the worker node and use that key to sign subsequent requests issued by the same job.

We implemented a proof-of-concept of the authentication server using pyGSI [22], a Python library for building a client-server authentication mechanism based on grid proxies, extensively used by the DIRAC community grid software [23].

5.4. Front-end server
The front-end server is responsible for exposing Amazon's S3 REST API to both GUI- and CLI-based clients. It validates incoming requests and processes them by using the services provided by the metadata store and the file contents store, the two core services of Mucura (see below).

We implemented a prototype of this service exposing a minimal subset of S3 REST API. It is implemented on top of Tornado, a web framework for building web applications in Python [24].

5.5. Metadata store
This component is responsible for storing the data associated to the files in the repository, such as file owner or creation date. The metadata store serves all requests that do not require access to the actual contents of the files, such as listing the repository contents or retrieving the creation date of one or more files.

We developed a prototype of this service in Python on top of Redis, a key-value data store [25]. The reason Redis was selected for this prototype is because it is a mature product, with a dynamic community of users and is used in production for supporting high-traffic services. In our preliminary experience in the context of Mucura, we measured the service is able to serve 12.000 metadata operations per second using an average server and 3 simultaneous clients.

Redis requires that the entire dataset fit in RAM on a single server. At this stage, this constraint is not a limitation for Mucura, as the volume of metadata is relatively small. From our measurements, to store the metadata of 1 million files requires approximately 400 MB of RAM, so one server typically found in data centers with 20GB of memory usable for Redis can accommodate metadata information for 50 million files.

Mucura is designed in a modular way so we can replace the backend of the metadata store if needed, without impact to the end-user. We would consider alternative backend implementations for storing the metadata, should Redis become a limitation for the use-cases Mucura addresses.

5.6. File contents store
As its name implies, this component is responsible for storing the contents of the files in the repository. Those contents are considered to be an opaque sequence of binary bytes: Mucura makes no interpretation of them. The front-end server uses the services provided by the file contents store to satisfy file storage and retrieval requests.

It is important to note that S3 servers store contents associated to object keys. There is no such concept of directory, as opposed to traditional file systems. In order to provide an approximation of the functionality of directories in an S3-compatible object store, clients implement different strategies. One of them is to create zero-length objects and associate client-specific metadata (or tags) that the client interprets and presents to the end user as a directory.

The implemented prototype of this component uses a networked file system as the backend for storing file contents. All the files belonging to a particular user are stored in the underlying file system under a directory hierarchy that does not match the organization of the user's files hierarchy. Instead, the system uses some simple techniques to distribute the stored files under several intermediate directories. This separation between the user's view of file's location and the actual physical location within the file system mitigates the risk of storing the contents of several thousands user's files in a single directory, which may cause performance penalties for some file systems.

Our end goal is to implement the file contents store on top of a third party, highly available data store. We are currently prototyping this component on top of Riak, a distributed persistent key-value store. In addition, we plan to explore OpenStack's Swift as a backend for Mucura's file contents store.

However, using a networked file system as the backend of the file contents store has proven useful for developing and testing the system. In addition some data centers expressed potential interest in deploying instances of Mucura on top of large installations of networked file systems they are already familiar with and know well how to operate.

6. Future Work

Even if we have not yet thoroughly tested this system in the field, we are convinced of the potential of the concept so we intend to continue developing it.

We plan to incrementally add features to the core components of the system, such as support for access control lists (ACLs) and multipart file upload and download. In addition, we scheduled an alpha-test period with end-users that will allow us to better understand how the system is actually used and evaluate its performance in a real world deployment.

Another important aspect we intend to work on is to improve the project's web presence, to use appropriate channels for distributing the software and eventually to create a community of end-users, services providers and developers.

7. Conclusions

We presented an exploratory work aiming at building an open source system for operating cloud-based file repositories intended for individual users. The motivation, goals, requirements, design decisions and implementation status were also presented.

Persistent distributed data stores have properties that make them attractive to serve as a backend of scalable file repositories. The prototype system we developed exploits those technologies for storing user's files and exposes a widely supported S3-compatible API. Although it has not been fully field-tested yet, initial feedback from users and potential service providers is very encouraging.

Acknowledgements

This work was supported in part by the French Ministry of Foreign Affairs and the Embassy of France in China, in the framework of a bi-lateral scientific collaboration between France and China.

References

[1] Worldwide LHC Computing Grid: http://lcg.web.cern.ch
[2] D. Thain, 2005 Chirp: An architecture for cooperative storage, Workshop on Adaptive Grid Middleware
[3] Amazon Simple Storage Service (Amazon S3): http://aws.amazon.com/s3
[4] Dropbox: https://www.dropbox.com
[5] GoogleDrive: https://drive.google.com
[6] Microsoft SkyDrive: https://skydrive.live.com
[7] Rackspace CloudFiles: http://www.rackspace.com/cloud/cloud_hosting_products/files
[8] SugarSync: https://www.sugarsync.com
[9] J. Barr, J. Varia, M. Woord and R. Buzescu, 2012, Amazon S3 – 905 Billion Objects and 650,000 Requests/Second, http://aws.typepad.com/aws/2012/04/amazon-s3-905-billion-

objects-and-650000-requestssecond.html
[10] Dropbox fact sheet. http://www.dropbox.com/static/docs/DropboxFactSheet.pdf
[11] G. DeCandia et al, 2007 Dynamo: Amazon's highly available key-value store, SIGOPS Oper. Syst. Rev.
[12] Apache Cassandra project: http://cassandra.apache.org
[13] Project Voldemort: http://project-voldemort.com
[14] Riak: http://basho.com/products/riak-overview
[15] OpenStack Swift: http://swift.openstack.org
[16] J. Hughes, 2010 Towards and Exabyte File System, http://indico.cern.ch/conferenceDisplay.py?confId=107217
[17] RiakCS: http://basho.com/products/riakcs
[18] SNIA’s Cloud Data Management Interface: http://www.snia.org/cdmi
[19] Cyberduck: http://cyberduck.ch
[20] Boto: https://github.com/boto/boto
[21] Amazon Simple Storage Service Developer Guide: http://docs.amazonwebservices.com/AmazonS3/latest/dev
[22] pyGSI: https://github.com/acasajus/pyGSI
[23] A. Tsaregorodtsev et al, DIRAC: A Community Grid Solution, Proceedings of the CHEP 2007 Conference
[24] Tornado: http://www.tornadoweb.org
[25] Redis: http://redis.io