FutureGrid Image Repository: A Generic Catalog and Storage System for Heterogeneous Virtual Machine Images

Javier Diaz, Gregor von Laszewski, Fugang Wang, Andrew J. Younge and Geoffrey Fox

Pervasive Technology Institute, Indiana University

2729 E 10th St., Bloomington, IN 47408, U.S.A.

Email: javidiaz@indiana.edu, laszewski@gmail.com

Abstract—FutureGrid (FG) is an experimental, high-performance testbed that supports HPC, cloud and grid computing experiments for both application and computer scientist. FutureGrid includes the use of virtualization technology to allow the support of a wide range of operating systems in order to include a testbed for various cloud computing infrastructure as a service frameworks. Therefore, efficient management of a variety of virtual machine images becomes a key issue. Current cloud frameworks do not provide a way to manage images for different IaaS frameworks. They typically provide their own image repositories, but in general they do not allow us to store the needed metadata to handle other IaaS images. We present a generic catalog and image repository to store images of any type. Our image repository has a convenient interface that distinguishes image types. Therefore, it is not only useful for FutureGrid, but also for any application that needs to manage images.

I. INTRODUCTION

FutureGrid (FG) [1] provides a testbed that makes it possible for researchers to tackle complex research challenges in Computer Science related to the use and security of grids and clouds. These include topics ranging from authentication, authorization, scheduling, virtualization, middleware design, interface design and cybersecurity, to the optimization of grid-enabled and cloud-enabled computational schemes for researchers in Astronomy, Chemistry, Biology, Engineering, Atmospheric Science and Epidemiology. FG provides a new experimental computing grid and cloud test-bed to the research community, together with user support for third-party researchers conducting experiments on FutureGrid.

One of the goals of the project is to understand the behavior and utility of cloud computing approaches. In this sense, FutureGrid provides the ability to compare these frameworks with each other while considering real scientific applications. Hence, researchers will be able to measure the overhead of cloud technology by requesting linked experiments on both virtual and bare-metal systems.

Since we are not only interested in offering pre-installed frameworks exposed through endpoints, we must provide additional functionality to instantiate and deploy them on-demand. Therefore, we need to offer dynamic provisioning within FutureGrid not only within an IaaS framework but allow the provisioning of such frameworks themselves. In this project, we use the term "raining" instead of just dynamic provisioning to indicate that we strive to dynamically provision even the IaaS framework or the PaaS framework [2].

Most of the cloud technologies are based on the virtualization of both resources and software, which makes the image management a key component for them. In fact, each IaaS framework provides its own local image repository specifically designed to interact with such framework. This creates a problem, from the perspective of managing multiple environments as done by FG, because these image repositories are not designed to interact with each other. Tools and services offered by the IaaS frameworks have different requirements and implementations to retrieve or store images. Hence, we present in FG the ability to catalog and store images in a unified repository. This image repository offers a common interface that can distinguish image types for different IaaS frameworks like Nimbus [3], Eucalyptus [4], but also bare metal images that we term distributed raw appliances in support of HPC. This allows us in FG to include a diverse image set not only contributed by the FG development team, but also by the user community that generates such images and wishes to share them. The images can be described with information about the software stack that is installed on them including versions, libraries, and available services. This information is maintained in the catalog and can be searched by users and/or other FutureGrid services. Users looking for a specific image can discover available images fitting their needs, and find their location in the repository using the catalog interface. In addition, users can also register customized images, share them among other users, and choose any of them for the provisioning subsystem [2]. Through this mechanism we expect our image repository to grow through community contributed images.

One of the most important features in our design is that we are not simply storing an image but rather focus on the way an image is created through templating. Thus it is possible at any time to regenerate an image based on the template that is used to install the software stack onto a bare operating system. In this way, the development of a customized image repository not only provides functional advantages, but it also provides structural advantages aimed to increase efficient use of the storage resources. Furthermore, we can maintain specific data that assist in measuring usage and performance. This usage data can be used to purge rarely used images,

while they still can be recreated with the use of templating. This will obviously lead to a significant amount of space saving. Moreover, the use of image templating will allow us to automatically generate images for diverse environments including a variety of hypervisors and hardware platforms. In this process, we will include mechanisms to verify that these requirements are reasonable like for example if the required IaaS is compatible with the requested hypervisor. In general, we can employ simple rules such as (a) if we find the image, we just provide it to the user (b)If not, we generate a new image to provide that to the user and store it in the image repository (c) if an image is rarely used it may get purged and we only keep the image generation template.

The rest of the paper is organized as follows. In Section II, we present an overview of image repositories provided by different cloud frameworks and storage systems. In Section III, we present the FG Image Repository by focusing on its requirements, design and implementation details. Section IV describes the tests performed to compare the different storage systems supported by the image repository and Section V collects the results of these tests. Finally, we present the conclusions in Section VI and future directions in Section VII.

II. BACKGROUND

As previously commented, the images are a key component in cloud technologies. Therefore, any cloud framework providing IaaS or PaaS has to manage them. Due to the particularities of each framework, each one has developed its own image repository adapted and optimized to its particular software. In general, IaaS frameworks provide the possibility to interact with the image repository, while PaaS frameworks hide all these details to the users. Next, we present an overview of the image repositories implemented by the most important frameworks to manage their images.

Nimbus [3], [5] is a set of open source tools that together provide an Infrastructure as a Service (IaaS) cloud computing solution. Since version 2.5, Nimbus introduced a storage cloud implementation called Cumulus [6] as part of its image repository solution. Cumulus is compatible with the Amazon Web Service S3 REST API [7] and can be used standalone as cloud storage. Currently, it is implemented using the POSIX file system as storage back-end, but the plan to provide access to distributed file systems.

Eucalyptus [4], [8] is open source software to deploy IaaS private and hybrid clouds. Eucalyptus provides a distributed storage system called Walrus which implements Amazon's S3-compatible SOAP and REST interface. It is designed to be modular such that the authentication, streaming and back-end storage subsystems can be customized. Walrus is used as storage for user data and images.

OpenNebula [9]–[11] is an open source toolkit which allows to transform existing infrastructure into an IaaS cloud with cloud-like interfaces. OpenNebula implements an image repository with catalog and functionality for image management. The image repository relies in the POSIX file system to store the images, and includes compatibility with Network File System (NFS) and Logical Volume Manager(LVM).

Amazon Web Services (AWS) [12] is a commercial platform to provide infrastructure web services in the cloud. Amazon maintains an abundant number of images covering popular OSs and architectures. In addition, users can generate and store their own customized images which then can be used by themselves or shared with others. Amazon's image repository uses its storage systems called S3 (Simple Storage Service) and EBS (Elastic Block Storage).

OpenStack [13] is a collection of open source technologies to deliver public and private clouds. These technologies are OpenStack Compute (called Nova), OpenStack Object Storage (called Swift), and the recently presented OpenStack Imaging Service (called Glance). The last one, Glance, is a lookup and retrieval system for virtual machine images. It supports different back-end configurations: using OpenStack Object Store, using Amazon S3 or using Amazon S3 with OpenStack Object Store as intermediate.

Windows Azure [14] provides developers with cloud capabilities through Microsoft datacenters. Since Windows Azure is a platform as a service (PaaS), it keeps the image repository hidden behind the scene and invisible to end users. However, they introduced the possibility to manage an image repository through Azure applications and the virtual machine (VM) role, recently. The main difference between both repositories is that in the first one, Microsoft patch and update the operating system for you, but with the VM role it is up to you.

Abiquo [15] is an open source infrastructure software for the creation and integral management of clouds based on heterogeneous environments. It maintains a public repository where users can download images. It also provides appliance repositories that can be defined by users. This repository is a simple NFS shared folder that is mounted by the Abiquo platform and all the nodes that compose the cloud infrastructure.

On the other hand, a very important detail to consider in the development of an image repository is the storage system because it is essential to provide scalable and fault tolerant applications. Some of the previous frameworks provide interesting storage systems like Cumulus [3], Walrus [4] or Swift [13]. However, there are other tools that can also be used to store information in distributed systems. In this sense, we have Google File System (GFS) [16] that was the first storage system to operate at cloud-scale. This is an scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance and delivers high aggregate performance to a large number of clients. GFS started a new technology trend that was quite well accepted by the community. In fact, Hadoop Distributed File system (HDFS) [17], [18] is popular open source distributed file system inspired in GFS. HDFS is highly fault-tolerant and provides high throughput access to application data and is suitable for applications that have large data sets.

We can also consider, as storage systems, various NoSQL databases [19]. These databases that may not require fixed table schemas, typically scale horizontally and are designed to manage huge amounts of data. In [19] we can find a list of the most important NoSQL databases ordered by type. While this databases are oriented to data mining in cloud, some of them also allow to store BLOBS (Binay Large Objects). The most

active projects are MongoDB [20], CouchDB [21] and Riak [22]. MongoDB is a document-oriented database that manages collections of JSON-like documents [23]. MongoDB includes an special component, called GridFS, designed to store files of any size in BSON format [24]. CouchDB is another document-oriented database that store the information in JSON and the large files are managed as attachments encoded in base64. On the other hand, Riak is a Dynamo-inspired key/value store [25]. It can manage large files using Luwak, an application built on top of Riak, which is bundle with Riak but disabled by default.

Finally, we would like to mention more traditional approaches used to provide networked and distributed file systems. Here, early examples are NFS [26] and AFS [27] with centralized client-server design. More recent approaches focused on HPC are LUSTRE [28] and PVFS (Parallel Virtual File System) [29], [30]. Both are parallel distributed file system, generally used for large scale cluster computing.

III. FUTUREGRID IMAGE REPOSITORY

The image repository is one of two important services within our image management. The other component is our image generation tool [2] which deals with the generation of template images that can be rained onto FG. We have applied the typical development life cycle to the FG image repository. Thus, in the following subsections we will talk about the different phases namely requirements, design and implementation.

A. Requirements

To specify our requirements for the image repository we have considered mostly the following four user groups:

- *A single user.* Users create images that are part of experiments they conduct on FG [2]. An image repository helps to manage their images, to share them or to create new images from existing ones adding additional packages configurations through scripts as part of the experiment environment.
- *A group of users.* An additional typical use case includes a group of scientific collaborators that work together in the same project and images are shared within the group instead of each collaborator creating an identical image.
- *System administrators.* They maintain the image repository ensuring backups and preserving space. They also may use it for the distribution of the HPC image that is accessible by default.
- *FG services and subsystems [2].* Different FG services and subsystems like our rain framework will make use of the image repository to integrate access and deployment of the images as part of the rain workflow.

Based on our consideration for the target audience we have identified a number of essential requirements that we need to consider in our design:

- *Diverse access.* The image repository must be accessible through a variety of access mechanisms such as a command line, a portal, an API, and a REST service.

- *Simple.* The image repository must be simple and intuitive to use by users that are not experts in virtualization technologies and distributed systems.
- *Unifying and integrated.* We must provide a unifying interface to manage various types of image for different systems. These systems may have their own repositories and we must be able to integrate with them in some fashion.
- *Extensible.* The image repository subsystem must be extensible. One aspect is to be able to include different back-end storage systems. Another aspect is to provide an API to offer the image repository to those cloud frameworks which allow the use of external image repositories.
- *Informative.* We must provide easy and meaningful access of information managed through the repository. Users should be able to query and report the status and attributes of the stored images.
- *Accountable.* We need to keep track of the usage of the images in order to optimize the space needed by removing unused images.
- *Secure.* The image repository security must be integrated in the FG security architecture. Moreover, in case a security issue is detected in an image, all cloned images from this one can be easily identified. Information associated with the repository and its images are protected through authorized access.
- *Fault tolerant.* The storage mechanism should be distributed to avoid single point of failures and performance bottlenecks.

B. Design

The FutureGrid image repository provides a service to query, store, and update images through a unique and common interface. In Figure 1 we present its architecture.
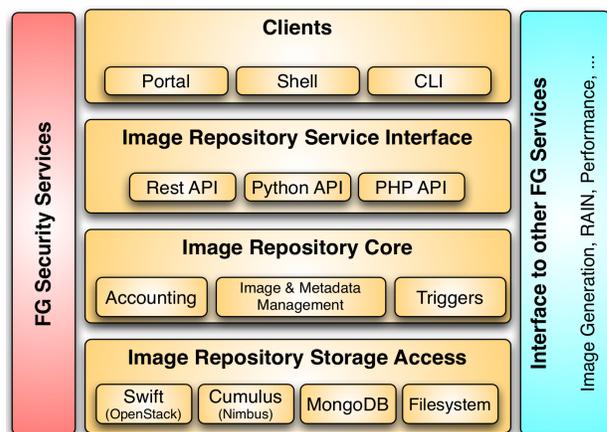
Fig. 1. FutureGrid Image Repository Architecture.

To address extensibility in a flexible and modular way, we have integrated a framework independent *Storage Access* layer. This layer defines an interface to create transparent plugins in support of different storage systems. Hence, a bridge between the storage systems and the image repository core functionality is provided. The *Image Repository Core*

contains the solutions to accounting including usage and quota management, image management and metadata management. The image management is focused on managing the image files and the associated information (metadata) in order to provide a consistent, meaningful and up to date image catalog. The separation of this information is done on purpose in order to support a variety of different storage systems that may be chosen by the site administrator due to functionality or integration requirements. Important to note is that the core also registers the image usage and access. This allows the repository to record information such as how many times an image was accessed and by whom. Internally this data may be used by a *trigger service* that cleanses the repository from faulty or less frequently used images. It also allows us to generate images from templates in case an image is requested with certain functionality that does not yet exist. Thus, instead of having a passive image repository, we move towards an active image repository that can be augmented with a number of triggers that get invoked dependent on the data that is collected within the repository. In this way, we can trigger events such as enforcing quota, automatically updating, or even distributing images based on advanced reservation events forwarded to us by the rain service. To access this functionality, we provide a variety of service interfaces such as an API, a command line interface, and REST services. These interfaces are part of the *Image Repository Service Interface* layer. Through these interfaces we can easily create higher-level image repository clients. In particular, we are immediately interested in provide access through the FG portal and the FG command line tool. Through the FG command line tool we can also integrate the repository commands as workflow scripts. The integration with a Web portal will be facilitated through REST services. An API library is available in python, and we intend to provide an API for PHP via the rest services. We are also designing plugins to allow other FG services to use the image repository and expose its use through such integration efforts to authorized users as part of raining images, IaaS, and PaaS onto the FG resources. Other cloud frameworks could integrate with this image repository by accessing it through an standard API such as S3 or the development of a back-end interface that directly accesses the images within our repository without knowledge to the users of these frameworks.

Finally, the security aspect is an essential component to be considered in the design. Thus, the image repository will provide the security functionality needed to integrate the authentication and authorization with the FG ones (based on LDAP). Using this approach we reach two important objectives. On the one hand, we increase the security, because the FG security is being developed by a group of experts in the field. On the other hand, we contribute to maintain a single sign on system for FG, avoiding the duplication of services and user databases.

C. Implementation

We are gradually implementing the features that are outlined in our design. The implementation is based on a client-server architecture like the one shown in Figure 2. This implementation targets a variety of different user communities including end users, developers, administrators via web interfaces, APIs, and command line tools. In addition, the functionality of the repository is exposed through a REST interface, which enables the integration with Web-based services such as the FutureGrid portal.
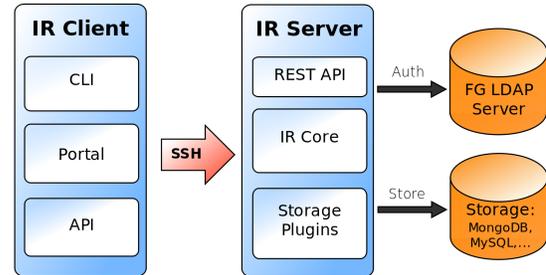
Fig. 2. Image Repository Client-Server Architecture.

Currently, our repository includes several plugins to support up to four different storage systems including (a) MySQL where the image files are stored directly in the POSIX file system, (b) MongoDB where both data and files are stored in the NoSQL database [20], (c) the OpenStack Object Store (Swift) [13] and (d) Cumulus [6] from the Nimbus project [3]. For (c) and (d) the data can be stored in either MySQL or in MongoDB. These storage plugins not only increase the interoperability of the image repository, but they can also be used by the community as templates to create their own plugins to support other storage systems.

We have already created a Command Line Interface (CLI) to manage the image repository. Next, we illustrate the image repository functionality.

a) User Management and Authentication: First, users will have to authenticate to access the image repository. This is not completed yet, but the access is going to be based on roles and project/group memberships. Since FG provides much of this information as part of an integrated portal and LDAP server, we can utilize it to provide authorization to access the repository while querying the FG account management services for the needed metadata on project memberships and roles.

As part of the user management, we currently maintain information related with users such as the quota determining the amount of disk space available for a particular user, the user status (pending, activated, deactivated) and the user role (admin or user). Repository administrators are the only ones with the ability to add, remove and list users as well as update the user's quota, role and status. Thus, we have detailed user-based and role-based access control to implement the previously mentioned authentication mechanism.

b) Image Management: To manage the images we maintain a rich set of information associated with each image (metadata). This includes the operating system, architecture, or image type. The current set of metadata information is shown in Table I including predefined values where applicable. It also shows which fields can be modified by users.

TABLE I
INFORMATION ASSOCIATED TO THE IMAGES (METADATA).

Field Name	Type	Predefined Values	Description	Access
imgId	string		Unique identifier	Read-Only
owner	string		Image's owner	Read-Only
os	string		Operating system	Read-Write
description	string		Description of the image	Read-Write
tag	string list		Image's keywords	Read-Write
vmType	string	none, xen, kvm, virtualbox, vmware	Virtual machine type	Read-Write
imgType	string	machine, kernel, eucalyptus, nimbus, opennebula, openstack	Aim of the image	Read-Write
permission	string	public, private	Access permission to the image	Read-Write
imgStatus	string	available, locked	Status of the image	Read-Write
imgURI	string		Image location	Read-Only
createdDate	date		Upload date	Read-Only
lastAccess	date		Last time the image was accessed	Read-Only
accessCount	long		# times the image has been accessed	Read-Only
ttl	date		Date when the image will be deleted	Read-Write
ttg	date		Date when the image will be replaced by its generation description	
size	long		Size of the image	Read-Only

We provide the ability to upload an image by specifying its location and its associated metadata. Defaults are provided in case some metadata values are not defined. The metadata includes also information about access permissions by users. In this way, we can define if an image is *private* to the user uploading the image, or shared with the public. Additionally, we are going to implement the ability to share an image with a selected number of users or a *group/project* as defined through the FutureGrid portal.

Modifications to the metadata can be accomplished by the owner of an image. However, some metadata cannot be changed, such as the last time an image was accessed, modified, or used.

We can retrieve images from the repository by name or by Uniform Resource Identifier (URI). Nevertheless, as some of our back-ends may not support URI's, such as MongoDB [20], the URI based access is not supported uniformly.

To remove images from the repository, users must own such images. Admin users can remove any image, though.

Users can also query the image repository. It uses SQL style queries to retrieve a list of images matching the query. Currently, we provide a very simple interface that allows us to conduct searches on the user exposed metadata using regular expressions. For example, to retrieve a list of images that match the OS to be Redhat and it is tagged with hadoop, we can use the query string * *where os=redhat, tag=hadoop*. Additionally, we can restrict the attributes of the returned metadata by using queries such as *field1,field2 where field3=value*, which returns only field1 and field2 of all images where field3 equals to the *value*. To return all information, users can simply pass a *, which is also the default in case no search string is provided. The use of this query language allows us to abstract the back-end system delivering a uniform search query across the different systems.

One additional very important property is the ability to support an accounting services while monitoring image repository usage. Important information provide by this service relates to the number of times that an image has been requested, the last time that an image was accessed, number of images registered by each user, disk space used by each user. Using this information we are going to implement automatic triggers that react upon certain conditions associated with the metadata. This includes the time to live (*ttl*) and the time to regenerate (*ttg*). The ttl specifies a time that allows users to automatically remove the image from the repository entirely. The ttg specifies when the image should be removed, but the metadata and the way the image is generated is preserved in the repository so that it can be recreated upon request. This feature will be helpful to manage many images by lots of users.

c) Command Shell: We have also developed a command shell for FutureGrid to unify the various commands and to provide a structured mechanism to group FG related commands into a single shell. Shells are very important as part of the scientific program development and have been popular with tools such as R, matlab, and mathematica. Hence, in addition to just exposing a single command line tool, we also provide a shell. The shell provides the advantage of defining more easily an script with a set of commands that need to be invoked to manage images and other FG related activities. It also provides the ability to log experiments conducted within the shell for replication. Thus, users will obtain a convenient mechanism to manage their own experiments and share them with other users through the FutureGrid shell. As scripts, pipes and command

line arguments can be used to pass commands into the shell, it provides a very convenient way to organize simple workflows as part of experiments within FutureGrid.

IV. Methodology

Since the image repository supports different storage systems, we need to know the expected performance of each system while working with the image repository. Therefore, we have conducted several performance tests to evaluate all these storage back-ends for the image repository. The back-ends include MongoDB, Swift, Cumulus, MySQL and an ext4 file system. To distinguish the setup in our Results' Section, each configuration is labeled as image storage+metadata storage. With this convention we have seven configurations: Cumulus+MongoDB (Cumu+Mo), Cumulus+MySQL (Cumu+My), Filesystem+MySQL (Fs+My), MongoDB with Replication (Mo+Mo), MongoDB with No Replication (MoNR+MoNR), Swift+MongoDB (Swi+Mo) and Swift+MySQL (Swi+My).

Figure 3 shows how we have deployed the image repository (IR) and the storage systems for our experiments. Within the experiments we have used 16 machines that are equipped with the image repository client tools. The image repository has been configured on a separate machine containing services such as the IR server, the Swift proxy, MySQL server and the MongoDB scheduler and configuration services (only used by MongoDB with replication). We have also used three additional machines to store the images and to create a replication mechanism. However, only Swift and MongoDB made use of the three machines, because they are the only ones that support replica service. In the case of Cumulus and the normal file system, we have only used one machine to store the images. Moreover, to allow comparison, we have also deployed MongoDB using a single machine without the replication service and therefore without the scheduler and configuration services. This deployment is labeled with MoNR+MoNR. However, in the case of Swift we could not avoid the use of replication since it needs a minimum of three replicas.
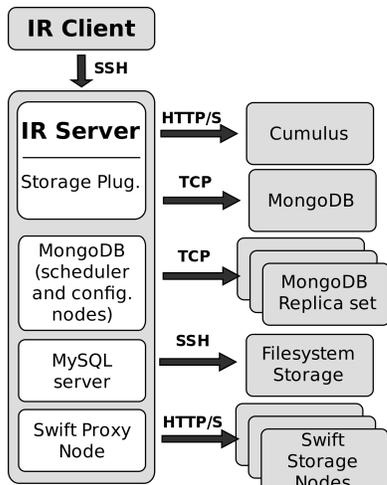
Fig. 3. Test deployment Infrastructure. Each gray box is a different machine.

We have considered five different image sizes: 50MB, 300MB, 500MB, 1GB and 2GB in order to covers realistic image sizes in use by FutureGrid users. We have compared both read and write performance for each storage system by uploading and retrieving images using a single client. In addition, we have tested a distributed scenario that involves 16 clients retrieving images concurrently. We have measured the average time that the clients need to retrieve or upload their images while running the test five times.

Tests have been carried out on FutureGrid while using the FG Sierra supercomputer at UCSD (University of California, San Diego). This cluster is composed by 84 machines with quad-core Intel Xeon processors and 32GB of memory. The cluster is connected using Infiniband DDR and 1 Gb Ethernet networks. The operating system is RHEL 6 and the file system format is ext4. The software used is Cumulus from Nimbus 2.7, Swift 1.4.0 (OpenStack Object Storage), MongoDB 1.8.1, and MySQL 5.1.47. Since the image repository is written in python, we use the corresponding python APIs to access to the storage systems. Thus, we use Boto 2.0b4 to access Cumulus [31], Rackspace cloudfiles 1.7.9.2 for Swift [32], Pymongo 1.10.1 for MongoDB [33], and pymysql 0.4 to access MySQL [34].

V. Results

First, we uploaded images to the repository to study the write performance of each storage system. The results are shown in Figure 4. We observe that the Cumulus configurations offer the best performance, which is up to 4.5% and 54% better than MongoDB with no replication (MoNR+MoNR) and Swift, respectively. Unfortunately, Cumulus does not provide any data-scalability and fault tolerance mechanism, which was in our experiments not a notable drawback. On the other hand, if we use MongoDB with replication (Mo+Mo), its performance degrades significantly resulting in a 70% worse performance for the 2GB case. This is due to two main factors, (a) the needed to send the same file to several machines and (b) the large amount of memory that this software requires. In fact, doing the same tests in machines with only 8GB of memory, the performance started to decrease even in the 300MB case. The reason of this performance degradation is that the memory usage is that MongoDB uses memory-mapped files to access data and is naturally memory bound. Once we hit the memory limitation, performance drastically declines. Finally, we had many problems with Swift due to errors when trying to upload larger files. Indeed, starting with the 600Mb case, the failure rate was more than 50% and for the 2GB case we were not able to upload a single image using the Python API. For this reason, we performed the last two tests by calling directly the command line tool included in Swift called *st*. It demonstrated that the documentation of the API is not yet sufficient and that the utilization of the provided command line tools is at this time a preferred choice for us.

Next, we study the performance of the different storage systems retrieving images. Since this is the most frequent use case for our image repository, we have performed two set of tests involving one or multiple clients.
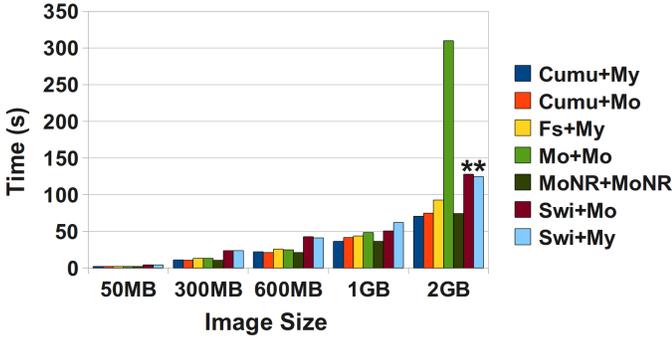
Fig. 4. Upload Images to the Repository. Asterisks mean that those tests were done using the command line tool instead of the Python API.

Figure 5 shows the results of requesting images from a single client. We observe that Cumulus provides us with the best performance. It is up to 13% better than MongoDB with no replication (MoNR+MoNR). Once again, by introducing replication to MongoDB (Mo+Mo), its performance degrades around a 30% due to the higher complexity of the deployed infrastructure. Finally, we can see that Swift performs quite well considering that it has to manage a more complex infrastructure involving replication and it is only 15% worse than Cumulus.
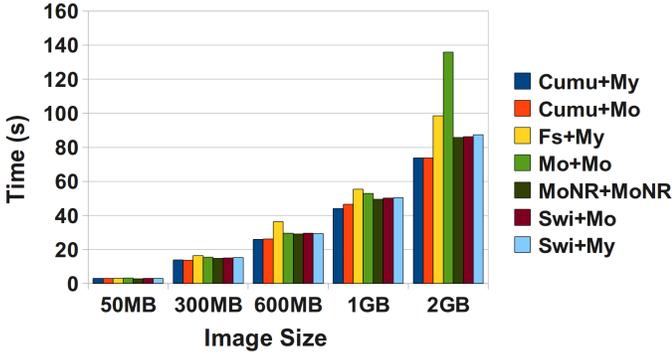
Fig. 5. Retrieve Images from the Repository.

The last set of tests shows the average time that each of the 16 clients spent to retrieve an image from the repository, see Figure 6. In this case, the Fs+My configuration has the best performance which is up to 53% better than any of the others. This is because Fs+My, unlike the other implementations, does not suffer form any performance degradation due to the overhead introduced by the software itself. We observe that the performance of Cumulus degrades when requesting the largest files. Hence, Swift provides a better performance in this case. However, Swift experienced significant reliability problems resulting in 31% and 43% of the clients not to receive their images. With respect to MongoDB, both configuration (MoNR+MoNR and Mo+Mo) had problems to manage the workload and in the 2GB case any client got the requested image due to connection errors. Therefore, only Cumulus and the Filesystem+MySQL configurations were able to handle the workload properly.
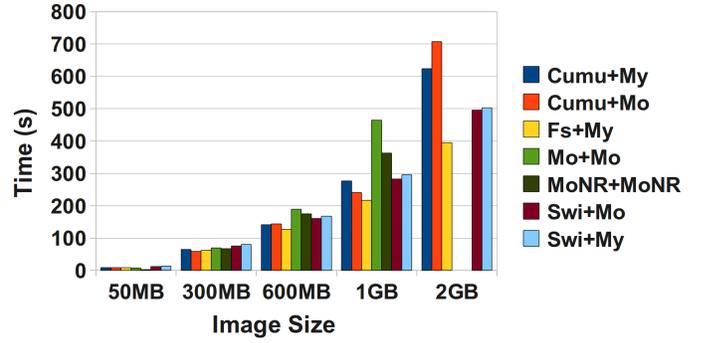
Fig. 6. Retrieve Images from the Repository using 16 client concurrently.

A. Discussion about the implemented Storage Back-ends

We have implemented four different storage systems based on MySQL, MongoDB, Cumulus and Swift, respectively and we discuss the advantages and disadvantages of each approach next.

d) Filesystem+MySQL approach.: MySQL is mature database and provides good security, scalability and reliability. The problem of relational databases is that they offer a rigid data model and we cannot provide solutions where each register has a different number of parameters. Moreover, since this approach uses the file system to store the images, it requires additional effort to explicitly provide mechanisms that ensure replication and fault tolerance. However, using the file system as storage back-end, we can potentially obtain good scalability and performance as we could use one of the HPC storage solution mentioned in Section II.

e) MongoDB approach.: MongoDB has implemented a sharing feature that distributes the database among different machines while maintain replicas in other machines. Through this mechanism, it provides good horizontal storage scalability and fault tolerance, although, in our tests, it degraded the performance due to resource starvation when reaching memory limitations. In addition, since MongoDB is a document-oriented database, it allows us to store documents containing different number of fields. Thus, we could offer users the possibility to enhance the metadata by including their own fields. Another advantage of MongoDB is that we can use a single framework to store both metadata and image files. Nevertheless, MongoDB has some drawbacks. It stores binary files in BSON format and therefore the serialization procedure could be one of the responsible of the low performance offered in some cases. Finally, as we commented in Section V, MongoDB uses large amounts of memory which strongly determines its performance when replication is used.

f) Swift approach.: Swift is a new framework as part of the OpenStack project designed to be highly scalable and fault tolerant. Its design is specifically aimed to object storage in the cloud. However, we observed that the Python API needs to be improved to get a better reliability. Additionally, we need to use an external database to store the metadata associated to the images, because Swift does not allow to store it.

g) Cumulus approach.: The Nimbus Cumulus cloud storage system is a recent development. As we commented

previously, it showed a good performance for our tests. The main problem is that the current version does not yet provide mechanisms to address good scalability and fault tolerance. For this reason, the Nimbus team is working to provide compatibility with other storage systems that will bring these features to Cumulus. Moreover, plans of developing a cloud database to allow data storage are underway. If these features are made available to us the need to use an external databases to store the metadata associated to the images would be eliminated.

VI. Conclusions

In this paper we have introduced the FutureGrid Image Repository. We focused on the requirements and design to establish the important features that we have to support. We present a functional prototype that implements most of the designed features. We consider that a key aspect of this image repository is the ability to provide a unique and common interface to manage any kind of image. Its design is flexible enough to be easily integrated not only with FutureGrid but also with other frameworks. The Image Repository features are enclosed and offered through a command line interface to provide an easy access to them. Furthermore, we provide an API to develop applications on top of the image repository.

We have studied the performance of the different storage back-ends supported by the image repository to determine which one is the best for our users in FutureGrid. Although none of them was a perfect match because of performance problems and high memory usage in the case of MongoDB, too many errors in Swift or missing fault tolerance/scalability like in Cumulus. Despite of the previous problems, we think that the candidates to be our default storage system are Cumulus because is still quite fast and reliable and Swift because has a good architecture to provide fault tolerance and scalability. Furthermore, we have an intense relationship with the Cumulus group as they are funded in part by FutureGrid and we can work with them to improve their software. We will have to monitor the development of swift closely due to the rapid evolution of OpenStack as part of a very large open source community. Our work also shows that we have the ability to select different systems based on future developments if needed.

VII. Ongoing Work

We are presently developing a REST API to the image repository and are integrating the automatic image generation. We would also like to provide compatibility with the Open Virtualization Format (OVF) to describe the images using this format.

Acknowledgment

References

[1] "FutureGrid Portal," Webpage. [Online]. Available: http://portal.futuregrid.org

[2] G. von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, K. Keahey, and E. Delman, "Design of the futuregrid experiment management framework," in *GCE2010 at SC10*, IEEE. New Orleans: IEEE, 2010.

[3] "Nimbus Project," Webpage. [Online]. Available: http://www.nimbusproject.org

[4] "Open Source Eucalyptus," Webpage. [Online]. Available: http://open.eucalyptus.com/

[5] K. Keahey, I. Foster, T. Freeman, and X. Zhang, "Virtual workspaces: Achieving quality of service and quality of life in the grid," *Scientific Programming Journal*, vol. 13, no. 4, pp. 265–276, 2005.

[6] J. Bresnahan, K. Keahey, T. Freeman, and D. LaBissoniere, "Cumulus: Open source storage cloud for science," *SC10 Poster*, 2010.

[7] "Amazon Web Services S3 REST API," Webpage. [Online]. Available: http://awsdocs.s3.amazonaws.com/S3/latest/s3-api.pdf

[8] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," *9th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, pp. 124 – 131, 2009.

[9] "OpenNebula," Webpage. [Online]. Available: http://opennebula.org/

[10] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009.

[11] I. M. Llorente, R. Moreno-Vozmediano, and R. S. Montero, "Cloud computing for on-demand grid resource provisioning," *Advances in Parallel Computing*, vol. 18, no. 5, pp. 177–191, 2009.

[12] "Amazon Web Services," Webpage. [Online]. Available: http://aws.amazon.com/

[13] "OpenStack," Webpage. [Online]. Available: http://openstack.org/

[14] D. Chappell, "Introducing windows azure," *David Chappell & Associates White Paper*, 2010.

[15] "Abiquo," Webpage. [Online]. Available: http://www.abiquo.com/

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *19th ACM Symp. on Operating Systems Principles (SOSP03)*, 2003.

[17] "Apache Hadoop!" Webpage. [Online]. Available: http://hadoop.apache.org/

[18] "Hadoop Distributed File System," Webpage. [Online]. Available: http://hadoop.apache.org/hdfs/

[19] "NoSQL Databases," Webpage. [Online]. Available: http://nosql-database.org/

[20] "MongoDB," Webpage. [Online]. Available: http://www.mongodb.org/

[21] "Apache CouchDB Project," Webpage. [Online]. Available: http://couchdb.apache.org/index.html

[22] "Basho Riak," Webpage. [Online]. Available: http://www.basho.com/Riak.html

[23] "Introducing JSON," Web Page, 2009. [Online]. Available: http://www.json.org/

[24] "BSON - Binary JSON," Webpage. [Online]. Available: http://bsonspec.org/

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *Proc. of the 21st ACM Symp. on Operating Systems Principles*, 2007.

[26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation or the sun network filesystem," *Proc. of the Summer 1985 USENIX Conference*, pp. 119–130, 1985.

[27] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[28] "LUSTRE ," Webpage. [Online]. Available: http://www.lustre.org/

[29] W. Ligon and R. Ross, "Implementation and performance of a parallel file system for high performance distributed applications," *Proc. of the 5th IEEE Int. Symp. on High Performance Distributed Computing*, 1996.

[30] "PVFS," Webpage. [Online]. Available: http://www.pvfs.org/

[31] "Boto: python interface to Amazon Web Services," Webpage. [Online]. Available: http://code.google.com/p/boto/

[32] "Rackspace interface for Swift," Webpage. [Online]. Available: https://github.com/rackspace/python-cloudfiles

[33] "MongoDB python API," Webpage. [Online]. Available: http://api.mongodb.org/python/

[34] "Pymysql: Pure Python MySQL client," Webpage. [Online]. Available: http://code.google.com/p/pymysql/