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. 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, 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 FG services.

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 future directions in Section VI.

II. Background

As previously commented, the images are a key component in cloud technologies. Therefore, any cloud framework providing IaaS or PaaS has an image repository to manage them. In general, IaaS frameworks like Eucalyptus [3], Nimbus [4], OpenNebula [5] or OpenStack [6] provide the possibility to interact with their image repositories. On the other hand, PaaS frameworks like Windows Azure [7] hide all these details to the users.

Another important detail to consider in the development of an image repository is the storage system. Some of the previous frameworks provide interesting storage systems like Cumulus (Nimbus) [4], Walrus (Eucalyptus) [3] or Swift (OpenStack) [6]. However, other applications like NoSQL databases [8] can also be used to store information in distributed systems. These databases typically scale horizontally and are designed to manage huge amounts of data. While they

are oriented to data mining in cloud, some of them also allow to store BLOBS (Binay Large Objects). In this sense, the most active projects are MongoDB [9], CouchDB [10] and Riak [11].

Finally, we would like to mention more traditional approaches used to provide networked and distributed file systems. Here, early examples are NFS and AFS with centralized client-server design. More recent approaches focused on HPC are LUSTRE [12] and PVFS (Parallel Virtual File System) [13]. 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. Next we present the different development phases of the FG image repository namely requirements, design and implementation.

A. Requirements

To specify our requirements for the image repository we have considered mostly the following four user groups: *single users* that create images as part of the experiments they conduct on FG [2]; *group of users* that work together in the same project and need to share the images within the group; *system administrators* that maintain the image repository ensuring backups and preserving space; *FG services and subsystems [2]* like our rain framework which 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. These requirement include a simple, intuitive and user friendly environment; a unified, extensible and integrated system design to manage various types of images for different systems; built in fault tolerance with proper accounting and information tools; and the ability to be integrated with the FG security.

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.

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

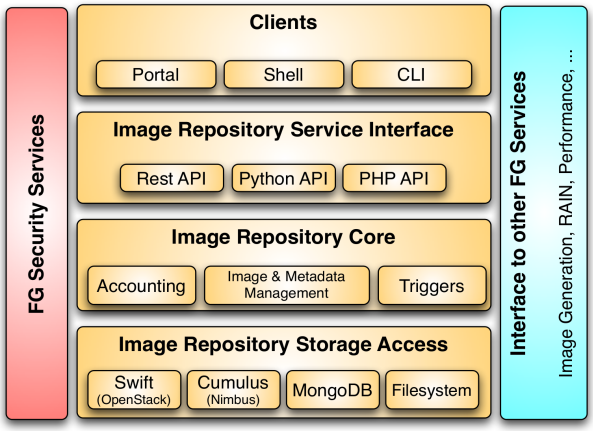

Fig. 1. FutureGrid Image Repository Architecture.

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.

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).

C. Implementation

We are gradually implementing the features that are outlined in our design. The implementation is based on a client-server architecture to target 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.

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 [9], (c) the OpenStack Object Store

TABLE I
INFORMATION ASSOCIATED TO THE IMAGES (METADATA). FIELDS WITH ASTERISKS (*) CAN BE MODIFIED BY USERS

Field Name	Description
imgId	Unique identifier
owner	Image's owner
os*	Operating system
description*	Description of the image
tag*	Image's keywords
vmType*	Virtual machine type
imgType*	Aim of the image
permission*	Access permission to the image
imgStatus*	Status of the image
imgURI	Image location
createdDate	Upload date
lastAccess	Last time the image was accessed
accessCount	# times the image has been accessed
size	Size of the image

(Swift) [6] and (d) Cumulus [14] from the Nimbus project [4]. 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). The current set of metadata is shown in Table I.

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 [9], 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 *. 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 can implement automatic triggers that react upon certain conditions associated with the metadata.

c) Command Shell: We have also developed a command shell for FG to unify the various commands and to provide a structured mechanism to group FG related commands into a single shell. The shell provides the ability to log experiments conducted within the shell for replication. 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 2 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. 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.

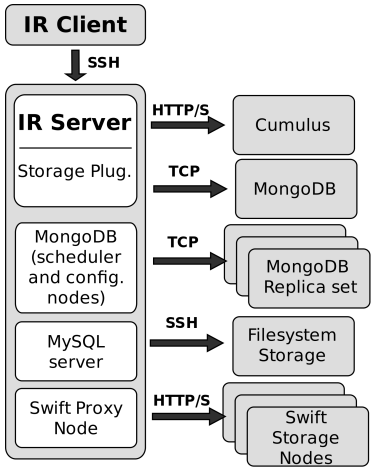

Fig. 2. Test deployment Infrastructure. Each gray box is a different machine.

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 [15], Rackspace cloudfiles 1.7.9.2 for Swift [16], Pymongo 1.10.1 for MongoDB [17], and pymysql 0.4 to access MySQL [18].

V. Results

First, we uploaded images to the repository to study the write performance of each storage system. The results are shown in Figure 3. 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.

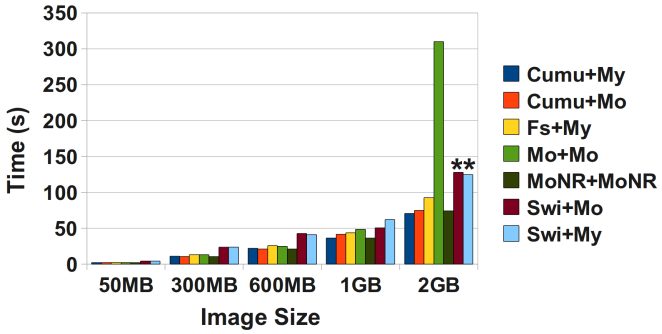

Fig. 3. Upload Images to the Repository. Asterisks mean that those tests were done using the command line tool instead of the Python API.

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.

Figure 4 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.

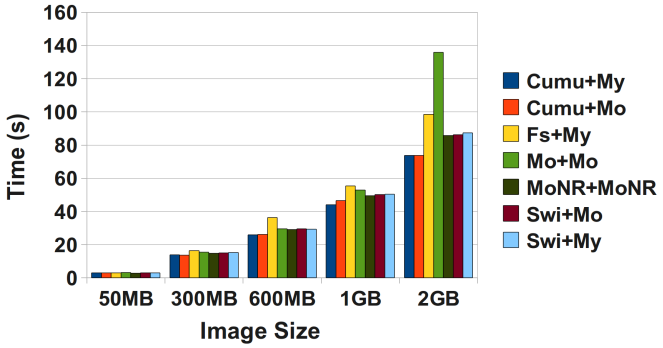

Fig. 4. Retrieve Images from the Repository.

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 5. 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.

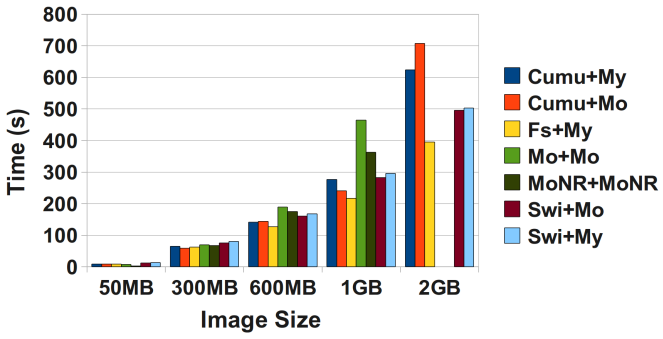

Fig. 5. Retrieve Images from the Repository using 16 client concurrently.

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.

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

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant No. 0910812.

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] "Open Source Eucalyptus," Webpage. [Online]. Available: http://open.eucalyptus.com/
[4] "Nimbus Project," Webpage. [Online]. Available: http://www.nimbusproject.org
[5] "OpenNebula," Webpage. [Online]. Available: http://opennebula.org/
[6] "OpenStack," Webpage. [Online]. Available: http://openstack.org/
[7] D. Chappell, "Introducing windows azure," *David Chappell & Associates White Paper*, 2010.
[8] "NoSQL Databases," Webpage. [Online]. Available: http://nosql-database.org/
[9] "MongoDB," Webpage. [Online]. Available: http://www.mongodb.org/
[10] "Apache CouchDB Project," Webpage. [Online]. Available: http://couchdb.apache.org/index.html
[11] "Basho Riak," Webpage. [Online]. Available: http://www.basho.com/Riak.html
[12] "LUSTRE ," Webpage. [Online]. Available: http://www.lustre.org/
[13] "PVFS," Webpage. [Online]. Available: http://www.pvfs.org/
[14] J. Bresnahan, K. Keahey, T. Freeman, and D. LaBissoniere, "Cumulus: Open source storage cloud for science," *SC10 Poster*, 2010.
[15] "Boto: python interface to Amazon Web Services," Webpage. [Online]. Available: http://code.google.com/p/boto/
[16] "Rackspace interface for Swift," Webpage. [Online]. Available: https://github.com/rackspace/python-cloudfiles
[17] "MongoDB python API," Webpage. [Online]. Available: http://api.mongodb.org/python/
[18] "Pymysql: Pure Python MySQL client," Webpage. [Online]. Available: http://code.google.com/p/pymysql/