Investigating Private Cloud Storage Deployment using Cumulus, Walrus, and OpenStack/Swift

Prakashan Korambath*
Institute for Digital Research and Education
(IDRE)
5308 Math Sciences
University of California, Los Angeles, CA 90095
ppk@ats.ucla.edu

Narcis Madern
Institute for Digital Research and Education
(IDRE)
5308 Math Sciences
University of California, Los Angeles, CA 90095
narcismadern@gmail.com

ABSTRACT

In this paper we describe our experience in investigating private cloud storage solutions and architectures for long-term data archival storage purpose. The storage solution needs in a campus research environment requires reliability and scalability up to petabytes of data at the minimum. The key requirements for a reliable and scalable cloud storage architecture is discussed below along with performance benchmarks on three different cloud storage tools. We deployed three open source cloud storage tools based on REST APIs namely Cumulus, Walrus, and OpenStack/Swift for this investigation in our data center.

Categories and Subject Descriptors

J.0 [**Computer Applications**]: General

General Terms

Management, Performance, Reliability, Storage

Keywords

Storage Cloud, Private Cloud, Data Transfer, Amazon's Simple Storage Service (S3), Cumulus, Walrus, OpenStack/Swift

1. INTRODUCTION

In a typical data center hosting petabytes of storage, the hardware components in the storage machine are expected to fail randomly, but the access to data needs to be persistent through duplication or some other means, meaning users should not be forced to re-run the experiment or calculation due to the loss of data. One of the reasons why continuous failure has to be accounted is because we want to use commodity of the shelf hardware (COTS) to bring down the cost of storage as low as possible. The storage system also needs to be decentralized to avoid single point of failures. Users need to be able to upload or download the data even when some of the hardware in the storage system is failing except for Internet connections. The system needs to handle the failures as a normal operation without impacting the performance to the user side. Manual administration requirement for storage system should be minimal. Users need to be authenticated and authorized in advance of data transfer and any hostile users need to be kept away. Access to the data should be feasible from desktops or servers running any operating systems such as Linux, Windows or Mac and the commands to do data transaction needs to be something as simple as put (object, file) and get (object, file).

We have no need for any relational database based solutions because we do not require complex query to figure out the relevance of the data stored in a folder as most of the data are transferred back and forth from a user home directory on a compute cluster or desktop. Our query model simply needs to list a set of folders and files belonging to an account. Users typically name the folders with some meaningful names or keep a README file inside the folder that describes the contents of the files. So retrieving the README file should be sufficient enough to help the users decide whether to download the entire folder or not. We do not require any protection from accidental overwrite or deletion. We expect all users of this system acknowledge the consequences of their action in advance and do not provide any safety mechanism. During the upload of the data, when there is a conflict of updating from multiple events, the last write wins strategy is quite appropriate for our purpose. The architecture should preferably support access control lists (ACL) so that researchers can grant read or write permissions to their collaborators. We do not expect this data to be available for a real time computation such as mounting through a NFS server for application job runs. Preferably, we would like to have some way of restricting storage capacity per user through a quota system or some sort of metering the resource. Since we expect storage need per user or the need for additional users increase continuously, we should be able to add more disks and servers to the storage system without any interruption in the service. The cloud architecture needs to satisfy the essential characteristics outlined in the NIST definition of cloud computing [1] such as on-demand self-service, broad network access, resource pooling, rapid

*Corresponding author's address: Prakashan Korambath, 5308 Math Sciences, University of California, Los Angeles, CA 90095.

elasticity and measured service.

2. POTENTIAL OPEN-SOURCE CLOUD STORAGE TOOLKITS

Based on the the limited requirements described above, we could use Amazon Simple Storage Service, popularly known as Amazon S3 [2], for our reliable storage need. There are several advantages in using Amazon S3 services, one of most appealing is its scalability to any amount of storage that user is willing to pay for. Stored data can be encrypted to be safe from any hostile activities. Also, one doesn't need to operate expensive data centers or have expertise in installing and maintaining a storage array. In this paper we were intended upon building private cloud storage for storing the data that needs to be within the campus boundaries. We already have space in our data center and have expertise in operating Linux based servers. Also, having private storage inside the data center will be cost effective when overall cost including payment for network bandwidth is taken into consideration. But the scalability of private cloud storage is not going to be as good as that of public cloud storage. So we looked at other S3 like cloud storage software that are open source and compatible with S3. They include Cumulus (IaaS:Nimbus) [3], Walrus (IaaS:Eucalyptus) [4] and OpenStack/Swift (IaaS:OpenStack/nova from Rackspace) [5]. All of them use REST (Representational state transfer) [6] based API to transfer data in and out of the storage hardware. REST is an architecture style described in the context of HTTP protocol for client server transactions with scalability as one of the many goals. Clients initiate a request such as put or get and server returns appropriate responses. GET, PUT, POST and DELETE are the main HTTP methods that the RESTful web-services typically use. Cumulus and Walrus can store the data in a regular POSIX file system where as OpenStack/Swift only supports block level storage, however Swift architecture is highly scalable and provides configurable replication of files. Walrus also supports Elastic Block level storage [7].

2.1 Cumulus

Cumulus is an open source implementation of Amazon S3 REST API distributed along with Nimbus IaaS toolkit. Python programming language is used to implement REST services in Cumulus. In particular, Twisted Web [8] is used for HTTP and HTTPS calls. Cumulus can be used as a standalone service to store data in and out of a storage machine or for the data transfer needs during the deployment of virtual computing images using Nimbus services. Although Cumulus supports a variety of storage systems such as PVFS, GPFS and HDFS, for this testing purpose we deployed cumulus on a POSIX file system and measured the performance using the open source REST client s3cmd [9]. Other clients for transferring files are Boto [10] and jets3t [11]. The S3 like interface in cumulus allows client to write, read and delete objects (objects here are equivalent of files) or organize them into buckets (equivalent of directories). The authentication mechanism in Cumulus is based on symmetric key to ensure the data is securely accessed. There is a local database (sqLite [12]) to store the access keys and ID to authenticate and authorize the users who are accessing the data. Cumulus also provides access control list (ACL) to allow a group of users to share data for read or write.

Table 1: Data transfer bandwidth using Cumulus cloud storage tools

Size (MB)	Upload (MB/s)	Download (MB/s)
100	25	21
200	20	21
500	26	28
1000	26	31
2000	25	32
4000	26	33
8000	26	33

Finally, it supports the storage quota, which is one of the ways resource usage per user can be controlled.

2.1.1 Cumulus Benchmarks

We have installed Cumulus distributed along with Nimbus version 2.7 on a server with AMD Opteron 2.0 GHz dual core dual CPU and 8.0 GB RAM running CentOS Linux OS. We carried out data uploads and downloads from a user desktop to a cumulus storage system connected through a public GigE network. The results of our measurement are given in Table 1. The data range we tested is between 100 and 8000 MB. The rate of upload is between 25 and 27 MB/s. The data download in the same range is between 21 and 33 MB/s. We used the same open-source s3cmd tool to perform the upload and download operation. The performance of Cumulus is expected to be comparable to that of 'scp' like services. The scp bandwidth on the same server is of the order of 30 MB/s. So, the results in Table 1. are consistent with our expectations. More data on Cumulus benchmark can be found in the paper by Bresnahan *et. al.* [13]. Maximum theoretical bandwidth in a GigE network is 125 MB/s. Typically one cannot expect more than 60-90 MB/s I/O bandwidth on a single hard-drive in a server connected to a GigE network. The data bandwidth can be increased by using parallel file systems such as GPFS or Lustre.

2.2 Walrus

Walrus is an Amazon S3 like cloud storage API distributed along with Eucalyptus IaaS toolkit. Eucalyptus is an open source implementation of Amazon EC2 to allow users to configure and deploy virtual machines in a private cloud. In order to run Walrus (Storage Controller) as a storage service alone, Eucalyptus Cloud Controller is necessary at the minimum. Cloud Controller is a top level API, which manages resource allocation, user accounts and the web interface for cloud management services. Walrus also make use of Elastic Block Level storage (EBS). Like S3, Walrus provides bucket-based object storage and has interfaces that are compatible with S3. REST based tools such as s3cmd or s3curl can be used for transferring data into and out of Walrus storage system. Walrus uses a JAVA based database for managing user account information with cloud controller and accessing meta-data of S3 buckets and objects stored in the database. If there are multiple concurrent writes to same object, Walrus uses the last write wins strategy. This is because Walrus doesn't impose file locking to achieve scalability.

2.2.1 Walrus Benchmarks

Table 2: Data transfer bandwidth using Walrus cloud storage tools

Size (MB)	Upload (MB/s)	Download (MB/s)
100	52	31
200	50	32
500	42	42
1000	41	41
2000	35	42
4000	30	36
8000	32	36

We installed Eucalyptus version 2.0 on a server with Intel Xeon 2.3 GHz Quad core dual CPU and 8.0 GB RAM running CentOS Linux OS. As in the case of Cumulus we have measured the data transfer bandwidth for Walrus service running on a POSIX file system from a desktop. We do not use Elastic block storage system (EBS) for this measurement. We used open-source s3cmd client version 0.9.8.3 along with a patch file provided at Eucalyptus website [14] to upload and download files from our desktop to the server. All the machines involved in this benchmark are connected through a public GigE network and the data is directly written to a local hard-drive. Before accessing Walrus each user needs to obtain a pair of keys called access-key and secret-key. Usually, this is obtained through a request on a web interface. There are some default configurations which limit the number of buckets and storage space per user. The command line interface tool s3cmd usually does not indicate the exact cause of error if one of those limits are exceeded. So, users should be aware of crossing such limitations. Also, the errors in such situation will lead to instability in file upload process.

We listed our results for uploading and downloading files from user desktop to Walrus storage server in Table 2. The upload bandwidth is between 52 MB/s to 30 MB/s. The download bandwidth is between 31 and 42 MB/s. In the case of Walrus upload seems to be faster than that of download.

2.3 OpenStack/Swift

Swift is a highly scalable open source cloud storage solution from Rackspace using REST based APIs. The APIs are written in Python programming language and is deployable on servers that are running any Linux OS although Ubuntu Linux 10.04 is the officially supported platform. Swift is suitable for archival storage purpose where static data can be uploaded for long term storage, retrieved for analysis or writing over with new data. Swift is not suitable for real time access as in mounting it through a NFS like service for real time computation or for accessing data like in a SQL database. Swift is designed to be scalable to Petabytes of storage with tens of thousands of hard drives. OpenStack also has a compute software API called nova for IaaS, but the storage API can be installed standalone. The public API for Swift is exposed through a proxy server. The proxy server is responsible for finding the location of an account, container or object in the 'ring' and route the request accordingly. The ring represents a mapping between the names of entities stored on the disk and their physical location. There are three types of rings called account, object and container to perform each kind of operation. Ring maintains a mapping of zones, devices, partitions and replicas. Each replica usually up to 3 copies is guaranteed to reside in different zones. A zone can be a drive, server, cabinet or a data center. The user data are stored as binary files (blobs) on the file system with meta-data stored in the file's extended attributes (XATTRS). An object server is responsible for facilitating the storage and retrieval of the data files. Last write always wins when multiple put operations are performed on the same object and upon deletion all replicated copies are removed as well. Swift runs a container server whose primary object is to handle the listing of objects. Containers are similar to directory names or file folders. The listings are stored in a SqLite database. The account server is responsible for listing the containers.

The Swift architecture uses a replication process ensuring the data is protected in the event of disk failures and network outages. It also make sure all copies are up to date to the latest version and deleted data is removed from all the replicated locations. The replication process is a push based rsync operation. There is also an updater process which will take care of updating the data in a queue during periods of high load. Auditors check for the integrity of objects, containers and accounts and corrupted files. They will be quarantined and replaced from another replica.

The ring-builder utility helps with the configuration of rings. Typical configuration parameters are number of *replicas, zones, ports* and *devices*. A zone is usually a group of devices which are in the same physical location and network address. Using the ring-builder, the storage can be reconfigured at any time. However, depending upon the complexities and rebuilding process, the immediate data access may be delayed for already saved data. If there are servers with different capacity, appropriate weights can be added to let Swift know the imbalance in the capacity of servers.

Although file size for downloading is unlimited, by default, Swift limits the size of a single upload object to 5 GB. The Swift script used for uploading the data has an option for segmenting objects larger than 5 GB into smaller pieces and upload them in segments. Swift will create a manifest file so that all segments can be downloaded at once.

2.3.1 Swift Benchmarks:

We deployed a Swift 1.3.0 based storage system with one proxy server and 5 storage servers configured to store up to 3 replicas of every uploaded objects. Our storage servers have an additional hard drive to run operating system in addition to 1 TB storage drive in each of the 5 servers. The proxy server has only a single hard-drive to run operating system. All servers are running Ubuntu Linux 10.04 64 bit Linux OS. The hardware is made up of AMD Opteron 2.0 GHz dual core dual CPU and 4.0 GB RAM server with slots for four SATA hard drives. Swift can be directly installed through apt-get command in Ubuntu Linux from the Swift-core repository. However, the repository usually has older versions and Swift is a fast developing software tool (initially, we started with Swift 1.2.0 using this automated process). So, a better solution is to download the tar files and configure manually to use the specific version that we are interested. The storage server hard drives are formatted with XFS file system. Since Swift requires extended

Table 3: Data transfer bandwidth using OpenStack/Swift cloud storage tools

Size (MB)	Upload (MB/s)	Download (MB/s)
100	17	27
200	18	17
500	21	30
1000	23	30
2000	27	18
4000	34	29
8000	34	23

attributes (XATTRS) to store the meta-data and XFS is the only file system thouroughly tested by Rackspace, it is the recommended one for using on the storage disks. The operating system hard drives are formatted with ext4 file system. Because installing RAID would degrade the performance very quickly, it is not recommended on the storage device. Insted of RAID, JBOD (just a box of disks) using several SATA drives (connected through SATA cables) is the preferred hardware configuration for Swift architecture and it is the one we actually deployed.

The proxy server is exposed to public IP address and the storage servers are all in a private subnet connected to the proxy server through its private IP address on the same subnet. The entire network is connected through a GigE switch. It is advisable to use multiple proxy servers with 10 GigE network cards in a protection system.

The results of our measurement from a user desktop are given in Table 3 for upload and download. The bandwidth for upload varies from 17 MB/s to 34 MB/s and download varies from 17 MB/s to 30 MB/s in the range of 100 MB to 8000 MB. As mentioned previously, when ever the data size is larger than 5 GB, the upload process is done in segments. In the case of 8 GB data we requested the upload script to segment it into 1 GB pieces. The script, called "Swift" (or st), needs to be copied over to the desktops for upload process to begin. In addition to this command line script there are few GUIs like Cyberduck [15] which can be used to run upload and download operations from a MAC or Windows desktop.

3. CONCLUSIONS

We investigated three cloud storage tools, Cumulus, Walrus and OpenStack/Swift. Figures 1 and 2 gives a comparison of the upload and download performance of all three. The results are average of 10 measurements. Their upload and download performance are comparable, Walrus being little faster than the other two for small data size. We are generally interested in the data size between 1000 MB and 4000 MB and, in that region, all three seem to have similar performance. Normally, for small size of data that can be fit into memory cache, I/O benchmarks appear fast because data is committed to the disk much later. This may explain much larger deviation in the observed bandwidth in 100 to 1000 MB region seen in the figures.

Cumulus offers little in terms of scalability unless one can use expensive parallel file systems. We have not investigated the elastic block storage (EBS) feature in Walrus yet. This

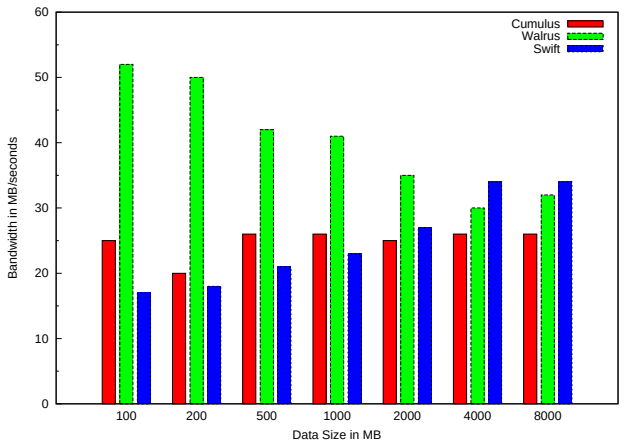

Figure 1: Comparison of upload bandwidth using Cumulus, Walrus and Swift

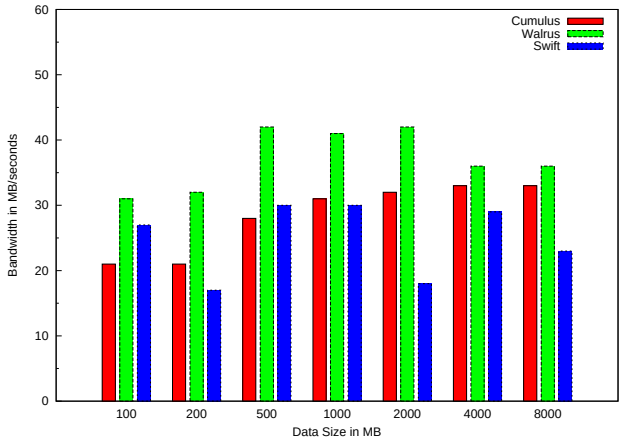

Figure 2: Comparison of download bandwidth using Cumulus, Walrus and Swift

will be one of our future works. However, we do have concerns about the stability of Walrus system. Even though the s3cmd client is very convenient, we observed that it did not return proper error messages when something goes wrong with configuration file or on the server side. So, users will mostly need help from Walrus server administrator to figure out why upload commands are not working correctly. Some of the situations may arise when users exceed the number of buckets or file-size limitations.

Based on the overall performance and scalability, we intend to build a production level cloud storage using OpenStack/Swift in coming days. We also intend to continue our benchmark with Elastic Block Storage on our Walrus server. The benchmarks are run on three different servers. Although they have similar hardware specifications, they are in different networks. Thus we cannot comment on the cause of differences in the benchmark data from three different experiments. In the case of Swift architecture having a single proxy server is a bottleneck when lots of operations are performed at the same time, but that can be alleviated by adding multiple proxy servers. For our investigative purpose the results from those results have given us a good exposure to three architectures. Overall, OpenStack/Swift seems to be stable and scalable for the requirements we outlined.

4. ACKNOWLEDGMENTS

We would like to acknowledge UCLA's Cyberinfrastructure grant for the informatics and computational data development for funding this work, especially Marsha Smith and the reviewers of our storage proposal to perform this study. Additionally, we would like to thank IDRE Research computing group staff for discussions on cloud storage tools, especially Tajendra Vir Singh, Shao-Ching Huang, Qiyang Hu (IDRE HPC group) and Bill Labate. We also thank Kendall N. Houk for being the faculty sponsor for this project. We thank Justin Tea for providing us network support. Finally, we thank our colleagues at SDSC, Doug Weimer, Shava Smallen and Ronald Joyce for sharing the details of their storage architecture (Swift) deployment with us.

5. REFERENCES

[1] NIST:. The nist definition of cloud computing (draft). http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf.

[2] Amazon:. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[3] Nimbus:. Open-source IaaS cloud similar to Amazon EC2. http://www.nimbusproject.org/.

[4] D.Nurmi, R.Wolski, C.Grzegorczyk, G.Obertelli, S.Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid. Washington, DC, USA: IEEE Computer Society, 2009.* CCGRID 09, 2009.

[5] OpenStack/Swift:. Cloud Storage based on Rackspace cloud storage API. http://www.rackspace.com/;http://swift.openstack.org/index.html.

[6] REST:. Representational State Transfer, Fielding's dissertation:. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[7] EBS:. Elastic block storage. http://aws.amazon.com/ebs/.

[8] Twisted Matrix Labs:. Twisted web. http://twistedmatrix.com/trac/wiki.

[9] s3cmd. command line s3 client:. http://s3tools.org/s3cmd.

[10] boto:. Python interface to aws:. http://code.google.com/p/boto/.

[11] Jets3t:. An open source java toolkit for amazon s3:. http://jets3t.s3.amazonaws.com/.

[12] SqLite:. Serverless database:. http://sqlite.org/.

[13] Cumulus:. Papers and tech reports. http://www.nimbusproject.org/papers/.

[14] Walrus:. S3-compatible tools for walrus. http://open.eucalyptus.com/wiki/s3cmd.

[15] Cyberduck:. OpenSource GUI for FTP, SFTP, WebDAV, Cloud Files, Google Docs, Amazon S3, and OpenStack/Swift for Mac and Windows. http://cyberduck.ch/.