Intercloud Object Storage Service: Colony

Shigetoshi Yokoyama, Nobukazu Yoshioka

GRACE Center, National Institute of Informatics, Tokyo, Japan
{yoko, nobukazu} @nii.ac.jp

Motonobu Ichimura

NTT DATA Intellilink, Tokyo, Japan
ichimuram@intellilink.co.jp

Abstract— **Intercloud object storage services are crucial for inter-organization research collaborations that need huge amounts of remotely stored data and machine image. This study introduces a prototype implementation of wide-area distributed object storage services, called colony, and describes a trial of its cloud storage architecture and intercloud storage services for academic clouds.**

Keywords-Cloud computing; Object storage service; OpenStack; Intercloud; Cloud federation.

I. INTRODUCTION

Cloud computing has the potential to dramatically change software engineering. It allows us to manage and use large-scale computing resources efficiently and easily. Moreover, it makes it possible to develop new software by using these resources for scalability and lowering costs.

For example, users can prepare machine images of standard education environments on Infrastructure as a Service to manage the environments efficiently. We have developed edubase Cloud [1], a cloud platform based on open-source software and using a multi-cloud architecture.

We are now developing a research cloud based in part on our experience in managing the edubase Cloud service during the disaster recovery efforts after the Tohoku earthquake and tsunami in March, 2011. Intercloud object storage services that can store machine images and research data remotely are crucial for such a development. Furthermore, if academic clouds are independently deployed and managed, there would be no way for users to continue working within clouds affected by disasters or other outages. By using intercloud object storage services, users can utilize machine images in other clouds operating normally.

We have developed an intercloud storage service architecture and a working prototype called colony [2]. This paper describes this development. Section 2 describes user scenarios on how to use intercloud object storage services. Section 3 presents a comparison with other storage services. We discuss the design and prototype of the intercloud object storage architecture in section 4 and 5, and conclude in Section 6.

II. USER SCENARIOS

The following are academic–cloud-user scenarios for intercloud storage services. In the scene depicted in Figure 1, there are two academic clouds, A and B, providing the intercloud storage service. The users of these clouds can store objects in local storage, i.e., storage-A or storage-B, or in the remote object storage, storage-I. Users just have to change the container attribute from local to remote or vice versa.

Storage-I should be geographically distributed for the sake of availability.

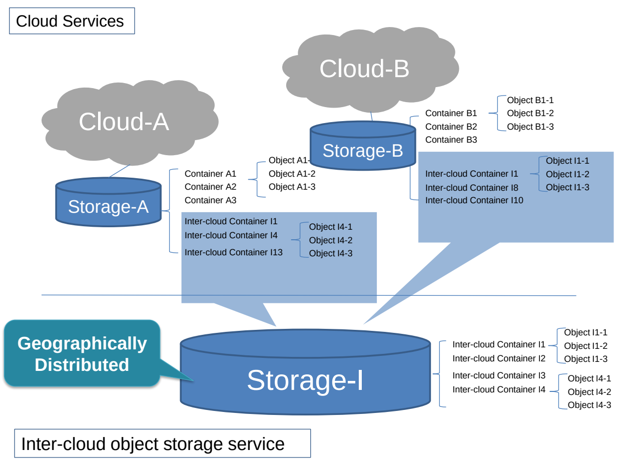


Figure 1. Intercloud object storage service.

A. *Access one's own objects from remote clouds*

Academic cloud users can access their own containers and objects from clouds that are remote from the one they usually use. The machine images stored as objects in storage-I can be used to launch virtual machines in these remote clouds. Machine image conversion might be needed before the launch, depending on the heterogeneity of the source and destination clouds.

B. *Access objects of other users*

Academic cloud users can share containers and objects with other users who may access them from remote clouds. The objects could be, for example, machine images or research data.

C. *Single sign-on to object storage services*

Each object storage service manages its own users but if each manages its users independently, users would have to login to a service every time they want to receive it. To deal with this problem, we support single sign-on among services by using a standardized identity management service such as shibboleth [2].

III. RELATED WORK

We thought that we should not start developing our intercloud storage service from scratch and that it would be better to utilize existing open source object storage service software. Figure 2 compares the various candidates that we examined in focusing on AWS S3 type Web API base object storage open source projects. S3 is a de-facto standard among object storage services, and there is a software eco system around it.

.

	baltic-avenue	board walk	fs3	Rado sgw	sinatra -s3	swift	Walrus
Redundancy mechanism	△	△	-	△	-	×	-
Max data size	1M	∞	∞	∞	∞	5G	5G
Max number of data	1000	∞	∞	∞	∞	∞	finite
Error correcting	×	×	-	×	-	×	-
ACL	×	-	×	×	×	×	×
Cache mechanism	-	-	-	-	-	-	-

×: OK, - : NG, △: redundacy mechanism with S3

Figure 2. Object storage service projects comparison.

Baltic-avenue [3], boardwalk [4], fs3 [5], sinatra-s3 [6] are effectively development test beds for S3, because they are not designed to have redundancy mechanisms. Because of this limitation, they cannot support huge intercloud object storage services.

Radosgw [7] is a web API front-end of the ceph distributed file system [8]. Walrus is a component of Eucalyptus [9], and although it is compatible with S3, it does not have a redundancy mechanism either.

Swift [10] supports large object storage services in commercial public clouds.

The above considerations led us to study OpenStack swift and modify it for our intercloud object storage service.

IV. DESIGN

A. OpenStack swift

OpenStack Object Storage (code-named Swift) is open source software for creating redundant, scalable data storage using clusters of standardized servers to store peta-bytes of accessible data. It is not a file system or real-time data system, but rather a long-term storage system for large amounts of static data that can be retrieved, leveraged, and updated. Object Storage uses a distributed architecture with no central point of control, providing greater scalability, redundancy and permanence.

Objects are written to multiple hardware devices, with the OpenStack software responsible for ensuring data replication and integrity across the cluster. Storage clusters scale horizontally by adding new nodes. Should a node fail, OpenStack works to replicate its content from other active nodes. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used in lieu of more expensive equipment.

Swift has proxy nodes and auth nodes acting as the front-end and storage nodes acting as the back-end for accounts, containers, and object storage.

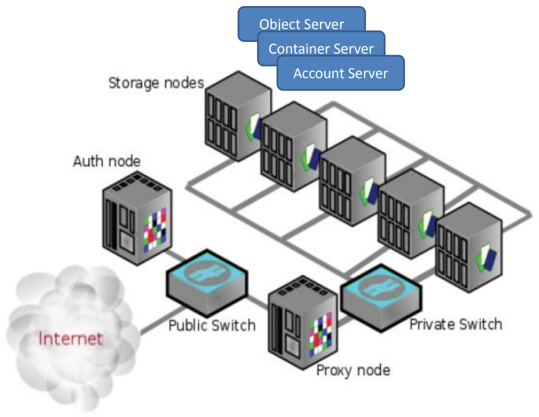


Figure 3. OpenStack swift.

B. Intercloud object storage architecture

Let us begin by discussing the intercloud object storage service architecture by categorizing how to allocate swift components such as proxy nodes, auth nodes, and storage nodes. The proxy nodes and auth nodes categorized as front-end. The storage nodes are categorized as back-end. We examined the suitability of the following architectures.

1. All-in-one architecture
   The front-end and back-end nodes are all on one site.
2. Fan architecture
   One front-end node is on the central site, and the back-end nodes are on each site.
3. Peer-to-peer architecture
   Each site has its own front-end nodes and back-end nodes. The front-end nodes communicate to synchronize the swift rings.
4. Zone architecture
   The front-end nodes have a hierarchical structure similar to the DNS hierarchy and use it to locate storage nodes.
5. Dispatcher add-on architecture
   Dispatchers that can recognize the destination front-end nodes are deployed as an add-on to the front-end.

All-in one, fan, and zone architectures have a single point of failure. The dispatcher add-on architecture is better than a peer –to-peer one because it require fewer servers at each site. Some sites only need to have the dispatcher. These considerations led us to choose the dispatcher add-on architecture.

This architecture has the following advantages:

- Easy to modify swift codes with it
- Easy to extend to more than two swift federations

V. PROTOTYPING

We are now prototyping intercloud object storage service and make the code public as the colony project in github [11] by using the dispatcher add-on architecture which is described in the previous section.

Figure 4 shows an overview of the colony architecture. New components such as swift dispatcher, VM info converter, and caching module were developed by analyzing this prototype. The dispatcher calls the local swift or intercloud swift depending on the container attributes. The VM info converter is used to convert the virtual machine image metadata for one cloud to metadata for another cloud in order to launch the machine image in the other cloud. The content cache helps to make the data transfer efficient.

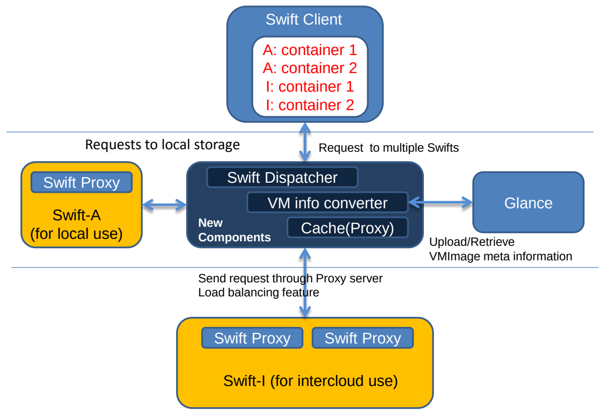


Figure 4. Colony overview.

The swift client can send requests to swift-A and swift-I through the swift dispatcher. In the prototype, the dispatcher can find the destination swift by looking at the prefix string in the container names. In the example in Figure 5, the prefix 'A:' indicates that the container resides in the local cloud, which is 'cloud-A.' The prefix 'I:' specifies that the containers having this prefix are located in the intercloud, which is 'cloud-I.'

When swift sends responses to the client, it merges the response from each swift, as described in Figure 5.

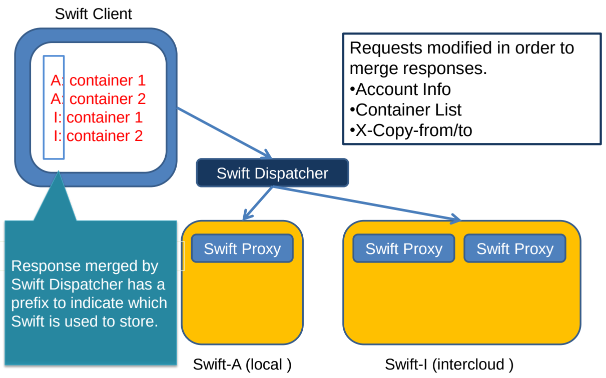


Figure 5. Swift dispatcher.

Swift dispatcher can use a cache proxy per swift proxy to retrieve objects from remote swifts (Fig. 6). In the prototype, the cache is implemented using a squid content proxy cache mechanism [12]. This sort of simple caching mechanism works because the swift proxies in the swift-I are located remotely from the swift client.

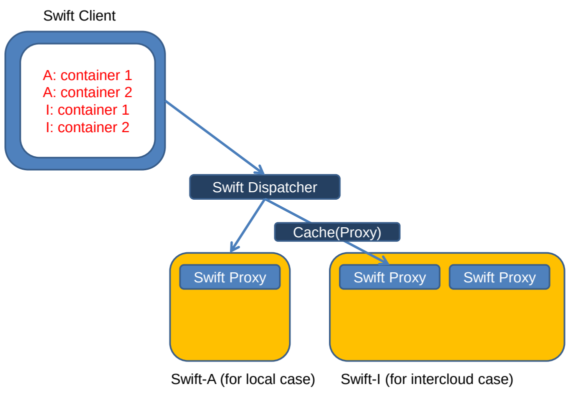
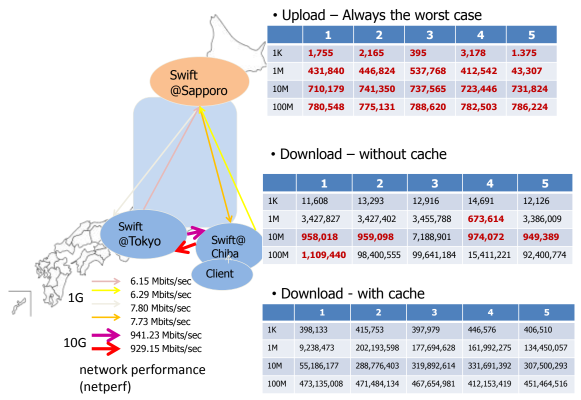


Figure 6. Colony cache.

We implemented a prototype of our intercloud storage service using colony and have started evaluating the performance and usability in three geographically distributed sites. So far, we can say that the colony load balancing seems to contribute to the performance of the intercloud object storage service. We located inter-region swift between three regions, i.e., Tokyo, Chiba, and Hokkaido, and investigated its performance in relation uploading/downloading objects. Throughputs between Tokyo and Chiba were about 1 Gbits/s while throughputs between Hokkaido and Tokyo/Chiba were about 7 Mbits/s.

In this case, uploading of objects is always the worst case because swift proxy puts objects in three zones, sets replication to default, and waits until all objects are uploaded. In contrast, the worst case of downloading objects is one-third of all transactions because the swift proxy randomly chooses one of three object servers. When downloading objects through web cache proxy l, the first download will likely be the worst case, but the results nonetheless show the cache proxy is effective (see Fig. 7).



• Upload – Always the worst case

	1	2	3	4	5
1K	1,755	2,165	395	3,178	1.375
1M	431,840	446,824	537,768	412,542	43,307
10M	710,179	741,350	737,565	723,446	731,824
100M	780,548	775,131	788,620	782,503	786,224

• Download – without cache

	1	2	3	4	5
1K	11,608	13,293	12,916	14,691	12,126
1M	3,427,827	3,427,402	3,455,788	673,614	3,386,009
10M	958,018	959,098	7,188,901	974,072	949,389
100M	1,109,440	98,400,555	99,641,184	15,411,221	92,400,774

• Download - with cache

	1	2	3	4	5
1K	398,133	415,753	397,979	446,576	406,510
1M	9,238,473	202,193,598	177,694,628	161,992,275	134,450,057
10M	55,186,177	288,776,403	319,892,614	331,691,392	307,500,293
100M	473,135,008	471,484,134	467,654,981	412,153,419	451,464,516

Figure 7. Uploading and downloading objects performance.

Swift should be zone-aware for geographically distributed use. For example, swift dispatcher can choose the best swift proxy to transfer a request to if it knows the network latency (see Fig. 8).

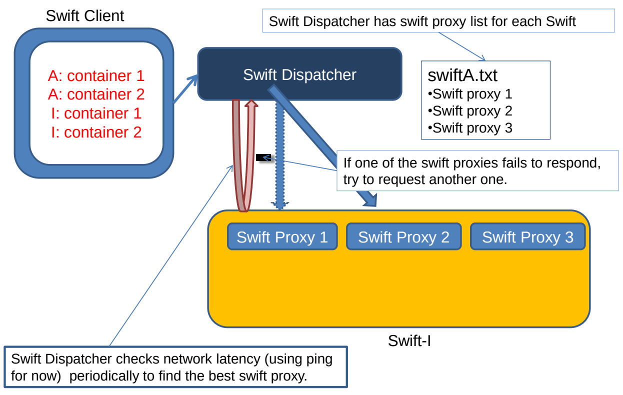

Swift Client
A: container 1
A: container 2
I: container 1
I: container 2
Swift Dispatcher has swift proxy list for each Swift
Swift Dispatcher
swiftA.txt
•Swift proxy 1
•Swift proxy 2
•Swift proxy 3
If one of the swift proxies fails to respond, try to request another one.
Swift Proxy 1
Swift Proxy 2
Swift Proxy 3
Swift-I
Swift Dispatcher checks network latency (using ping for now) periodically to find the best swift proxy.

Figure 8. Colony load balancing.

The swift code of the prototype was modified as follows:

- Uploading

Calculate the number of unfinished tasks in the send queue for each area and when one area has much more than the others stop uploading jobs to it.

- Downloading

Check the connection performance of the object servers and try to retrieve an object from the fastest one. Uploading performance improves by utilizing zone awareness (Fig. 9).

object size	1	2	3	4	5
1K	11,356	13,157	13,074	12,758	12,680
1M	9,824,750	11,205,249	7,599,312	10,931,206	11,199,982
10M	52,294,403	51,437,092	51,050,686	52,641,471	52,300,141
100M	97,937,987	101,847,002	102,385,002	102,413,801	101,462,855

Figure 9. Uploading performance with zone awareness.

The VM info converter can be used to share virtual machine image metafiles and is implemented as a swift dispatcher filter (Fig. 10). This implementation enables the shared machine images stored in intercloud storage service to be launched in user specified cloud compute services.

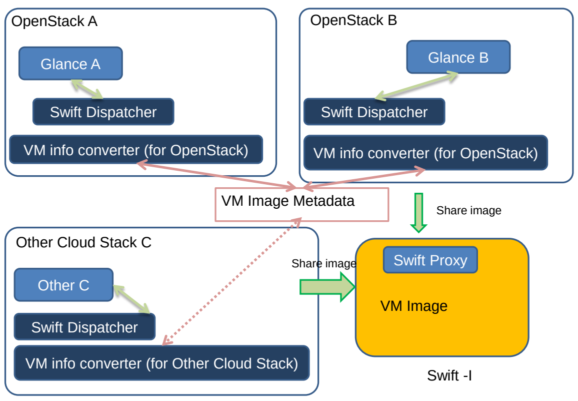


Figure.10. Colony virtual machine image metadata converter.

VI. CONCLUSION

We described an intercloud storage service architecture and prototype using code of the project called colony. The architecture looks feasible, and we will continue to evaluate it in a real environment and enhance the code for better performance.

We already know that there are points in the intercloud object storage service we could tune to get better performance. These points and their evaluations will be reported in the future.

REFERENCES

[1] Nobukazu Yoshioka, Shigetoshi Yokoyama, Yoshionori Tanabe, and Shinichi Honiden , “edubase Cloud: An Open-source Cloud Platform for Cloud Engineers,” SECLOUD '11 Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing, 2011.

[2] Shibboleth: http://www.shibboleth.net/ [retrieved: June, 2012]

[3] Baltic-avenue : http://code.google.com/p/baltic-avenue/[retrieved: June, 2012]

[4] Boardwalk :https://github.com/razerbeans/boardwalk [retrieved: June, 2012]

[5] Fs3: http://fs3.sourceforge.net/ [retrieved: June, 2012]

[6] Sinatra-s3: https://github.com/nricciar/sinatra-s3 [retrieved: June, 2012]

[7] Radosgw: http://ceph.newdream.net/wiki/RADOS_Gateway

[8] Ceph: http://ceph.newdream.net/ [retrieved: June, 2012]

[9] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov, “The Eucalyptus Open-source Cloud-computing System,” 2009 J. Phys.: Conf. Ser. 180 012051.

[10] OpenStack Swift: http://openstack.org/downloads/openstack-object-storage-datasheet.pdf and http://docs.openstack.org/cactus/openstack-object-storage/admin/os-objectstorage-adminguide-cactus.pdf [retrieved: June, 2012]

[11] Colony: https://github.com/nii-cloud/colony [retrieved: June, 2012]

[12] Squid:http://www.squid-cache.org/ [retrieved: June, 2012]