Cross-object Compressed Storage System

Shivang Singh
Electrical Engineering Department, San Jose State University
San Jose, California, USA
shivang.singh@sjsu.edu

Abstract*— Cross-object storage is a storage architectural approach in which variety of objects are managed and saved in forms of objects, utilizing both on premise and cloud resources. This storage technology allows huge retention for massive unstructured data present in today's world. The policy factors determine the stored objects in such a way that they are continuously moved to cloud. This project aims at creating a cross-object storage system which uses OpenStack as its backbone and also leverages the services provided by Amazon AWS S3 to create a robust, scalable and cheap storage system which is also prone to on-premise disasters.

Keywords: Cross object storage, OpenStack, Amazon AWS

I. Introduction

In today's fast moving digital world, the rate of data growth is increasing at an alarming rate, leading to issues such as space allocation. Lack of space leads to costly space solutions hence leading to problems all across the globe. Even the devices which are being made today have a limited amount of storage capacities, making users and organizations to move from on board storage solutions to cloud storage systems. There is a slow but steady growth from on premise disks storage solutions to cloud systems. Cloud Systems provides multiple benefits as compared to the old traditional storage services, which includes colossal flexibility, immense availability at a cheaper price, no maintenance cost and many more to describe [1]. The best solution for maintaining all these things in best suited way is to develop a cross-object storage system, which utilises both on-premise as well as cloud services.

Having a cross-object storage system moderates all the advantages which includes higher scalability, huge reliability, higher security, cheap investment and full control over data. As per [2] cloud storage is a cloud computing model in which data is stored at remote locations, which are accessible via Internet. The new concept of cross-object storage system bridges the gap in between high control and high price storage systems.

Our system will act as a supplement to our local on premise storage system. This system is also equipped with the new facility of **Compression**, so that objects are compressed before being sent ahead for putting at on premise and cloud services.

II. Inspiration

The inspiration for functioning towards a cross-object storage system included multiple factors which kept on influencing the decision. Here are some of the motivations:

- Low on-premise storage available with current technology.
- Lack of a disaster recovery in the form of a backup system.
- A cross-object storage is more scalable and redundant than a traditional storage.
- Preserving money with lesser utilization of resources that are required to run a physical storage.
- A hybrid cloud model can enhance responsiveness to new requirements as the need to procure and provision storage transitions away from the IT customer to the service provider who, in theory, is bound by service level agreements [3].
- There are many hardware dependencies in our current storage model combined with the fact that tuning a storage is an expensive and time consuming process.

III. Cross-object Compressed Storage System

Our project, Cross-object Compressed Storage System, is basically a solution which unifies on-premise and cloud storage systems into one via a Python script on CLI. The system is using OpenStack Swift as its on-premise storage and backing up objects to Amazon S3, which is today's cheapest, simplest and highly accessible public cloud. Our framework works in multiple spheres, with the most paramount feature of Compression.

A. Paramount Functionalities of our System

In a nutshell, the project enforces subsequent functionalities :

- COMPRESSION:

Compressing the object before being uploaded and put into OpenStack and further to AWS S3.

- BACKING UP:

Backing up data into 2 parts from OpenStack to Amazon S3 for making storage extremely affordable.

● DISASTER RECOVERY:
Having a copy of the objects and a mechanism to get the object in any sudden case of Disaster.

● REPLICATION:
Making multiple copies across both storage systems for getting copies of data and saving it at any cost.

IV. System's Peripherals

A. OpenStack

OpenStack is a free and open source software that controls large pools of compute, storage and networking resources throughout a datacenter, managed through a dashboard or via the OpenStack API. It is a collaboration between NASA and Rackspace, which started in the year 2010. It is written in Python language and serves at Infrastructure-as-a-Service (IaaS) and allows administrators to efficiently and dynamically manage storage [4].

OpenStack has Swift API for the storage of objects, which we have also utilised. Swift is extremely available, distributed. Objects are always stored in a flat structure and can be accessed using REST APIs [5].

B. AWS S3

AWS are the most prominent cloud provider, offering computing, storing and delivery platforms to its users [6]. Amazon S3 is a web service which is provided by Amazon Web Services (AWS). It has a simple interface which can be used for storing and retrieving any amount of data at any time and from anywhere across the globe. It is the cheapest and fastest growing storage system available today in the world [7].

Objects are the most fundamental entities in Amazon S3 [8]. An object consists of Metadata and the object data. In S3, an object is identified within a bucket by a key and a version ID of it.

C. Python

Python is an object oriented programming language, which incorporates several modules, dynamic typing exceptions and many classes. Python is extremely portable in nature and can run over multiple OS. It can also be used as an extension language for applications which require any interface to run [9].

V. Architectural Movement

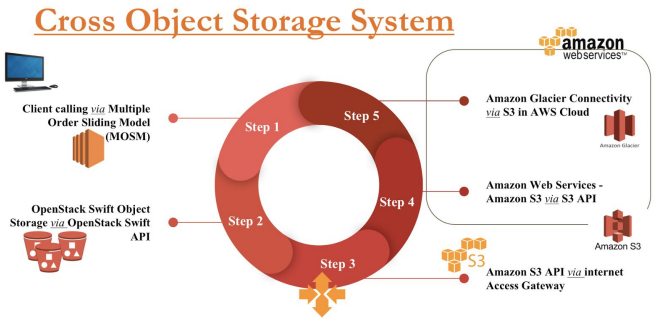

Figure 1: The architectural movement showing the steps in which the system is going to perform

The architectural movement how the storage system works and moves is via Command line interface (CLI), which system utilises to interact with the applications. The system is deployed on OpenStack and AWS, which provides the storage solution.

- Our system is unfolded on AWS's Storage Service named 'S3' communicating with OpenStack, which is the local on-premise storage solution for our system.
- The client calling up to the system is done with the help of Multiple Order Sliding Model (MOSM).
- The further communication between the local system and the OpenStack Swift Object Storage is done with the help of OpenStack Swift API and the Swiftclient.
- The connectivity between the local on-premise storage, which is OpenStack's Object Storage solution is done with the help of Internet Access Gateway and the Amazon's S3 API.
- Communication to AWS is driven using Python boto3 framework.
- We have used Boto3 framework for AWS, for managing and configuring the AWS Access and Secret Keys, which have full access for our AWS account.

VI. Working

The working of our project goes in multiple steps, which includes, Initialization of the connection with OpenStack and AWS simultaneously; Creation of buckets/ containers in AWS and OpenStack respectively; Putting objects along with Compressing them before hand only; Retrieving them and the Quitting the system.

A. Initialization of Connections with OpenStack and AWS

The initiation of the connection is done simultaneously in OpenStack and AWS by using the initialization module. With the initialization of connection, a prompt of successful start is obtained along with multiple functionality available in our system.

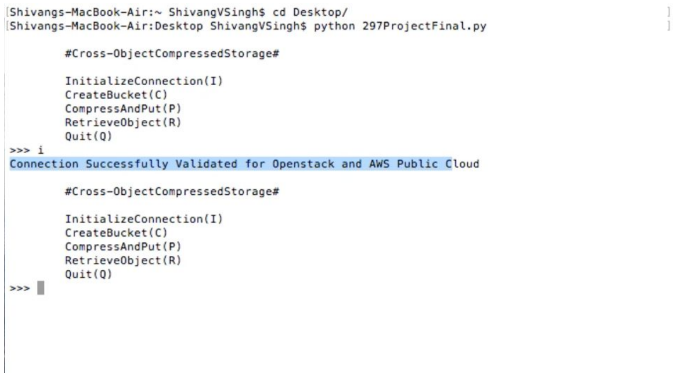

Figure 2: Initialization of connections with prompt showing validation

B. Creation of containers/buckets and replication to OpenStack and AWS

The container/bucket creation and replication is done simultaneously in this step for both on-premise storage and AWS.

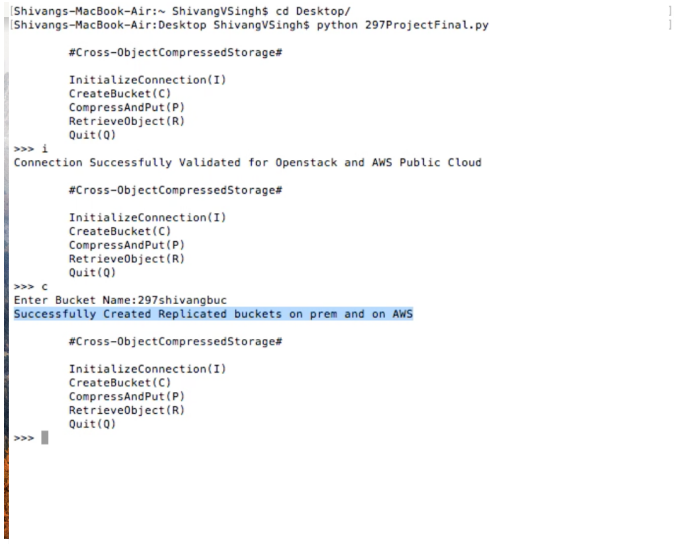

Figure 3: Creation and Replication of containers/buckets on OpenStack and AWS S3

C. Compression and Putting the objects in buckets/containers

The most important milestone of our is "**Compression"** of objects before uploading them to the OpenStack, which is our on-premise storage system. In this step, the API compresses and puts the object on OpenStack and replicates it in the Amazon S3 Storage.

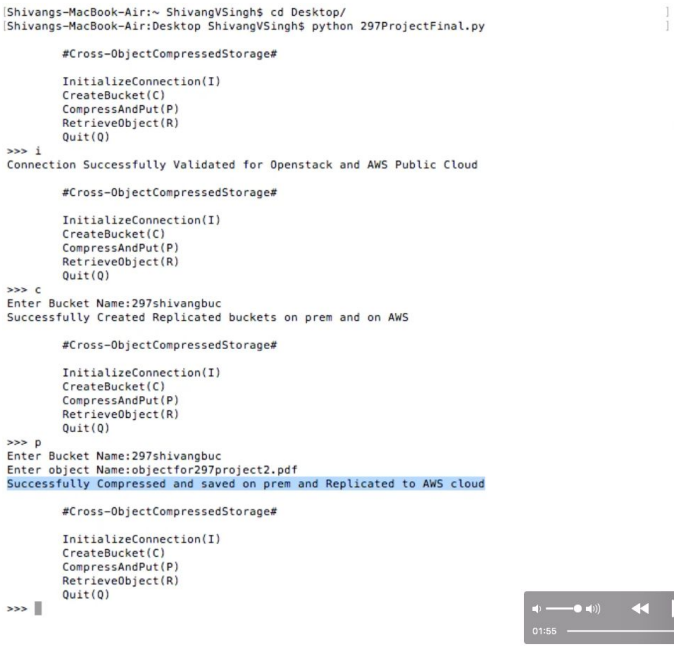

Figure 4: Successful compression, putting and replication on local and on cloud storage systems

In the following image, we can see that the object has been placed and the sie has been decreased to 2.55 MB instead of its original size which is 3.7 MB, as uploaded to the Amazon S3.

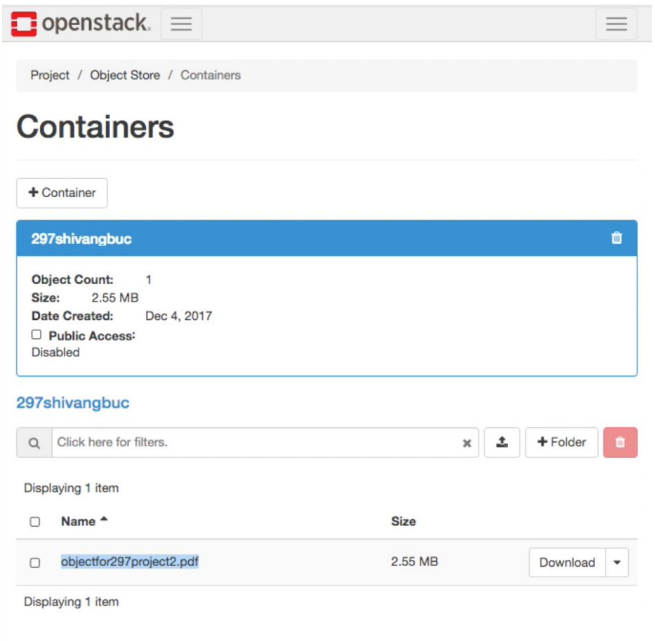

Figure 5: Size of object in OpenStack (2.55MB), after being compressed

Here the file is replicated and further uploaded to the Amazon S3 storage system with its original size of 3.5 MB.

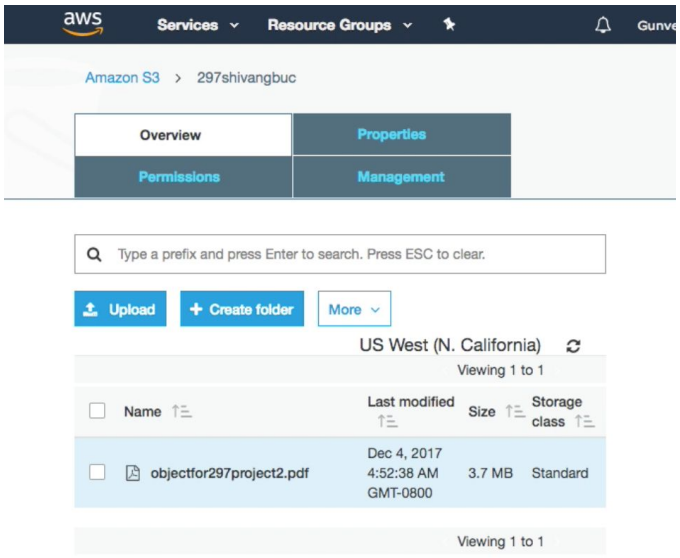

Figure 6: Size of object in Amazon S3 (3.7 MB)

This reduction in size of the uploaded object in OpenStack is due to the feature of Compression, which compresses the objects before hand only for uploading. This is the biggest feature of our system since there is a rapid increase in the storage demands all across the globe and the technological community has to find some solution for this ever-lasting highly increasing demand for storage.

D. Retrieving Objects

The object is fetched from OpenStack firstly, and if in case it is not present there, it will be fetched from Amazon S3, since any Disaster might have affected the storage system of OpenStack.

```
>>> r
Enter Bucket Name:297shivangbuc
Enter object Name:objectfor297project2.pdf
Downloaded the fileobjectfor297project2.pdffrom public Cloud

        #Cross-ObjectCompressedStorage#

        InitializeConnection(I)
        CreateBucket(C)
        CompressAndPut(P)
        RetrieveObject(R)
        Quit(Q)
>>>
```

Figure 7: Successful retrieval of object

Ahead of that the object has been saved in form of zipped file, which can be used for getting the real original objects by opening them as seen below. The object is retrieved and saved at the same place where the directory location is currently present. Here in our case, it is the Desktop, where the file has been downloaded further unzipped by the user.

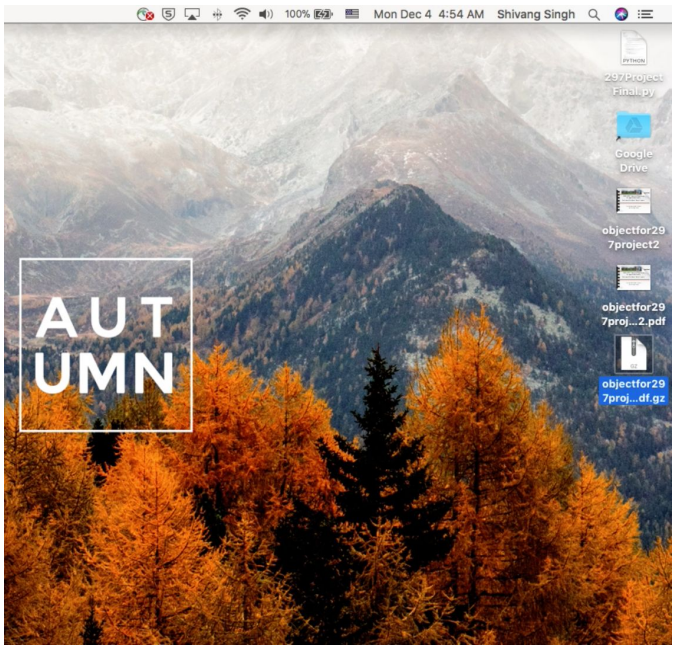

Figure 8: Retrieved object as seen on the device and the original file once it has been unzipped

VII. Advantages

The most prominent advantages of our system are listed as below -

- Our system caters an exceptionally simplified storage solution which can deal with both on-premise and public cloud.
- The system is extremely crucial in providing a redundant and scalable architecture for fulfilling the immensely increasing need for storage.
- It serves a long term solution to the increasing problem of storage systems.
- It also exhibits automatic backup and compression of the objects to OpenStack and S3, with 100% transparency.
- It is flexible in nature and can be scaled up or down as per the requirement by any storage providing company.
- It is super economical and saves huge money-related issues.
- It is an uncomplicated straightforward API-accessible platform.

VIII. API Dependencies Utilized

A. *INIT API:* This is the module which is used for initialization of connection to OpenStack, which is our on-premise storage system and to the AWS S3. Once the connection has been started, then the menu with other functionalities of the system are presented to the user, for their selection.

B. *CREATE BUCKET:* This API is used for creating a bucket / container in OpenStack and S3. It is the one inside

which the object is placed and stored after being compressed at the terminal of the operator.

C. *COMPRESS AND PUT OBJECT:* The Compression of objects is done with the gzip API. It first compresses the objects which are to be uploaded to OpenStack and S3 and then they are pushed by used PUT Object API which pushes the compressed objects to the buckets/containers. With the help of PUT object API, the high availability of the objects is managed since it replicates and then pushes the objects to the two storage solutions. The replication is done with PUT Object API only.

D. *GET OBJECT:* This GET Object API ensures successful retrieval of the objects to the device of the user; with trying firstly from the local storage solution (OpenStack) and if in case of any Disaster, then from the Amazon S3.

IX. Performance Analysis

What has previously done in the field of object storage has been a simple uploading of files/objects on a cloud platform. This project is different from all others in the respect that it keeps the data as it is on the on premise storage and compresses the data on the on premise before uploading the data on the cloud platform which is AWS S3 in our case. This tweaking helps us out in a lot of different ways. Firstly, it keeps the data pure and unchanged on the on the cloud storage which ensures there is no change in the data or data loss associated with compressed data. Compressing our object reduces the storage memory utilized by 31.08% and also saves huge time while uploading. Along with this, keeping a compressed version on the premise helps us in our fight against already minimal space for our data without the cloud. The cloud acts as a disaster recovery. Compressing files on-premise and storing the original sized files on cloud is a highly efficient system which gives us a performance unmatched in previous hybrid storage systems.

X. Conclusion

After working on this cross-object storage solution we can conclude some things for certain. What we have here is a multiple platform system. Three platforms to be precise, the device, Openstack and the AWS S3 cloud. We compress our files on the device before storing it on the openstack so that we can store the maximum amount of data. Openstack is the backbone for our project. AWS S3 acts as a disaster recovery. The features we have added help us in uploading objects on the cloud platform and openstack. Also, we can fetch those objects from both the platforms using the python code. This system helps us make a highly reliable and effective cross-object storage system.

XI. Possible Future Scopes

- Usage of Machine Learning while creating the buckets and containers and then putting the objects on behalf of multiple classes.
- An operating system such as Drop Box, totally relies on multiple Virtual Platforms, for obtaining a greater scalability, security and reliability.
- Making system much more speedy for using algorithms utilized in uploading and retrieving the objects in form of objects.
- Creating multiple security clearances so that private data is not stolen and remains secure for longer period of time.

XII. Source Code[1]

The source code for our project is given as below which includes all the dependencies required for making this script work well along with all the additional functionalities.

```
import swiftclient
import boto3
import gzip
import gzip
import shutil


#Creating a connection to S3

class storageCompressor():
#    s3connection
        def __init__(self):
                '''
                Initializes the connection to S3 and
Openstack

                '''
                self.swift_conn =
swiftclient.client.Connection(authurl='Your URL/identity/v3',
                 user='Username', key='Key',
tenant_name='Name', auth_version='Your Version',
                 os_options={'tenant_id': 'Your Tenant ID',
'region_name': ''}
                 )


                self.s3_client = boto3.client( 's3',
aws_access_key_id='Your AWS Access Key',
                        aws_secret_access_key='Your
AWS Secret Key',
                        region_name = 'us-west-1'
                        )
```

[1] Enter your AWS and OpenStack security credentials in the source code, wherever required.

```
                if self.swift_conn and  self.s3_client:
                        print "Connection Successfully
Validated for Openstack and AWS Public Cloud"
                else:
                        print "Connection Failed"


        def createbucket(self,bucket_name):

swift_created_bucket=self.swift_conn.put_container(bucket_n
ame)

s3_created_bucket=self.s3_client.create_bucket(Bucket=buck
et_name,CreateBucketConfiguration={
   'LocationConstraint': 'us-west-1'})

      #Returning Success (0) if buckets are successfully
created else returning 1

                if swift_created_bucket and
s3_created_bucket:
                        return 0
                else:
                        return 1

        def
compressAndPut(self,bucket_name,object_name):


response_s3=self.s3_client.put_object(Body=open(object_na
me, 'rb'),Bucket=bucket_name,Key=object_name)
                with open(object_name, 'rb') as f_in,
gzip.open(object_name+'.gz', 'wb') as f_out:
                        shutil.copyfileobj(f_in, f_out)
                file_post=open(object_name+'.gz', 'rb')



response_swift=self.swift_conn.put_object(bucket_name,
object_name, file_post)

        def retrieveObject(self,bucket_name,object_name):

response=self.s3_client.get_object(Bucket=bucket_name,Key
=object_name)
                with open(object_name, 'w') as f:
                        chunk =
response['Body'].read(1024*8)
                        while chunk:
                                f.write(chunk)
                                chunk =
response['Body'].read(1024*8)
                print "Downloaded the
file"+object_name+"from public Cloud"

if __name__=="__main__":

    while True:
       print("\n\


#Cross-ObjectCompressedStorage#\n\n\tInitializeConnection(
I)\n\tCreateBucket(C)\n\tCompressAndPut(P)\n\tRetrieveObje
ct(R)\n\tQuit(Q)")
       choice = raw_input(">>> ").lower().rstrip()
       if choice=="i":
          sc=storageCompressor()



       elif choice=="c":
            bucketname=raw_input('Enter Bucket
Name:').lower().rstrip()

            if sc.createbucket(bucketname):
                    print "Successfully Created Replicated
buckets on prem and on AWS"



       elif choice=="p":
            bucketname=raw_input('Enter Bucket
Name:').lower().rstrip()
            object_name=raw_input('Enter object
Name:').lower().rstrip()

            sc.compressAndPut(bucketname,object_name)



            print "Successfully Compressed and saved on prem
and Replicated to AWS cloud"


       elif choice=="r":
            bucketname=raw_input('Enter Bucket
Name:').lower().rstrip()
            object_name=raw_input('Enter object
Name:').lower().rstrip()
            sc.retrieveObject(bucketname,object_name)


       elif choice=="q":
          break
       else:
          print("Invalid choice, please choose again\n")
```

REFERENCES

[1] https://www.cloudwards.net/cloud-storage-good-idea/

[2] https://www.allbusiness.com/cloud-storage-business-21059-1.html

[3] http://www.tomsitpro.com/articles/hybrid-storage-solutions,2-719.html

[4] https://www.openstack.org/

[5] https://docs.openstack.org/swift/latest/api/object_api_v1_overview.html

[6] Ignacio Bermudez, Stefano Traverso, Marco Mellia, Maurizio Munafo, “Exploring the cloud from passive measurements: The Amazon AWS case”, INFOCOM, 2013, IEEE.

[7] http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html

[8] http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#BasicsObjects

[9] https://docs.python.org/3/faq/general.html#what-is-python