Automatic scalability of the OpenStack object storage

Oleksandr Porunov[1]

[1] National Aerospace University «KhAI», Department of Computer Systems and Networks, 17 Chkalov Str., Kharkiv, Ukraine

alexandr.porunov@gmail.com

Abstract. Modern big web-services should be developed with scalability and high availability. Modern high-load projects must cope with the loss of a server, rack of servers, data center or several data centers. It is not acceptable when a big business stops because of server overload or loss of any of the network elements. Also, small businesses, which aim is to grow in the near future, have to design their architecture to be easily scalable. The most non-trivial task is to construct a data warehouse because it is the stateful service and lots of servers need to be managed to have a storage which is big enough to store all users' data. The paper describes methods which might be taken to scale OpenStack object storage. Particular attention will be drawn to the automation of the object storage scalability. The solutions of scaling OpenStack Swift are suggested.

Keywords. Storage, Swift, OpenStack, Scalable, Automatic

Key Terms. DataWarehousing, DataCloud, ServiceOrchestration, Technology, Object

1 Introduction

Object storage takes a significant part in IT. Data is continuously growing that is why a scalable solution is required to store new data. OpenStack Swift is one of the most popular open source object storages. It has been designed to fit lots of data and to be scalable as much as possible. OpenStack Swift is highly available because it stores replicas of data as far as possible from each other [1]. To describe its nodes Swift uses ring files which contain all the information about devices in the cluster [2].

Data would be in danger without security. Thus OpenStack Swift supports OpenStack Keystone out of the box.

OpenStack Keystone is an OpenStack service that provides API client authentication, service discovery, and distributed authorization by implementing OpenStack Identity API [3]. Keystone is able to use either LDAP or SQL server as a backend.

Almost always OpenStack Keystone is used with OpenStack Swift because it provides convenient authentication and authorization. OpenStack Keystone supports different authorization methods like password based, token based and ec2 based methods and so on. Also custom authorization methods can be added to the keystone.

Projects that use OpenStack object storage as a big object storage need to have automated storage scaling because it's almost impossible to scale big storages manually.

The goal of the article is to develop an algorithm which can be used to automatically scale both OpenStack object storage and OpenStack identity service.

2 The problem of scalability

2.1 Keystone scalability problem

The most popular and secured method to authorize users in Keystone is the token based method. The problem is that it doesn't easily scale out of the box. It has several techniques to generate tokens: UUID, PKIZ, PKI and FERNET [4].

UUID method generates a random string in a database and returns it as a token. When validation is performed it searches the token in the database and compares them. The main problem is all the tokens need to be stored in the database in order to be validated. Of course there are several layers of cache which can be enabled but still the database is the one to work with.

PKI method generates tokens which are signed documents that contain the authentication context, as well as the service catalog. The Identity service uses public/private key pairs and certificates in order to create and validate PKI tokens [4]. The problem with such type of tokens is a very long length depending on the size of the OpenStack deployment.

PKIZ tokens are the same as PKI tokens. The only difference is that PKIZ tokens are compressed to help mitigate the size issue of PKI. PKI and PKIZ tokens are deprecated and not supported in Ocata release.

FERNET tokens are extremely lightweight and aren't stored anywhere (same as with PKI and PKIZ tokens). To generate a fernet token the keystone uses a key which is stored in a keystone machine. For security reasons it is recommended to update a key after some usage because if a malefactor finds out the key they will be able to generate their own tokens and use all services as an administrator. To verify a token on any machine the same key must be stored on all keystone machines. It would be hard to update a key on all servers in one moment. That is why the fernet method provides different types of keys.

- Primary key is the key which is used to generate and verify tokens. After key rotation a primary key becomes a secondary key [5].
- Secondary keys are used only to verify tokens. A limit can set for the count of these keys. After secondary keys reach the limit the oldest key will be deleted [5].
- Staged key is the key which will be used as a next primary key. After key rotation a staged key becomes a primary key and new staged key is generated [5].

Because of these key types it is possible to rotate a key on any keystone machine and then distribute new keys to all other machines. Still there are automation and distribution problems. Keystone doesn't have a tool to update and distribute keys on all machines.

2.2 Swift scalability problem

OpenStack Swift scales to enormous sizes but it has some complicity with the scaling. To manage all drives it stores all information about devices of the cluster in special files called rings. Data is distributed around the cluster with a modified consistent hash ring algorithm. When a device is added, removed and device weight is changed new ring files must be distributed to all swift nodes. After the node has received new ring files it starts a replication process.

Of course it would be hard to distribute new ring files to all nodes and move replicas to new locations in one moment. That is why a special parameter "min part hours" was created. It is responsible for two actions. Firstly it is time after which a ring can be rebalanced again. Secondly when Swift moves replicas it moves only one replica and other stays locked at the same place until either replica movement is done or "min part hours" have passed.

Still the problem with Swift scalability is adding, removing or changing a weight of a device will cause a movement storm and the cluster will suffer for some period of time. For example when drives are added to the Swift cluster it redistributes its data to the new drives immediately. For instance if a drive with 4 TB is added to a cluster which is 50% full then it causes 2 TB movements to the drive. With 10Gb Ethernet port it would take about 27 minutes at 100% utilization, assuming the source drives have enough capacity to send the data, the new drive can consume it at that pace, and the network switches can support that transfer. In practice, degraded performance from the cluster will take hours [6]. To prevent degraded performance of the cluster, capacity could be added gradually but the problem is that there are no any opened methods which describe how to do it automatically.

3 The solution of scalability

3.1 Keystone scalability solution

As Keystone uses an SQL database to store users' information, keys information could be stored in the same database. It is advisable to have a highly available database. MariaDB Galera Cluster or Percona Xtradb Cluster or something similar can be a good choice. They both have master-master architecture.

It would be hard to manage crond jobs on all nodes that is why a configuration management software can be used. It is possible to choose any configuration management software but it is advisable to choose software which scales easily and can be highly available. SaltStack is a good choice as it supports different backends and can work in a multi-master mode with a failover option.

The problem with the key rotation is that it must be rotated only on one node in one moment. If keys are rotated on different hosts at one time, then different hosts will have different keys. Several such rotations will result in inconsistent keys on different hosts. To prevent this situation, global locks should be used across the cluster to identify which node will rotate keys in particular time. It is possible to use leader election technique but it consumes a little bit more CPU and bandwidth so global locks are the best choice in this situation. De facto standard for such tasks is Zookeeper because it is lightweight and has all the functionality required.

The final algorithm to update keys will be as follows:

1. SaltStack will send a job to right nodes to update keys if they differ. The job will be sent for example two times per day.
2. If keys differ from the keys in database then keys will be updated.

The final algorithm to rotate keys will be as follows:

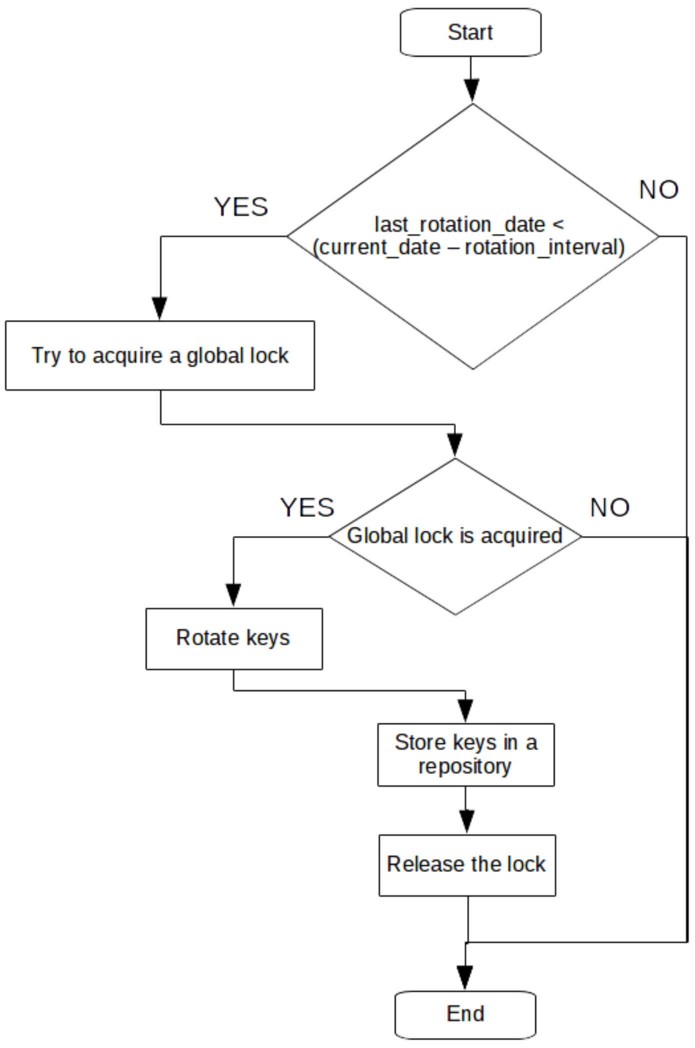
Start
YES
NO
last_rotation_date < (current_date – rotation_interval)
Try to acquire a global lock
YES
NO
Global lock is acquired
Rotate keys
Store keys in a repository
Release the lock
End

Fig. 1. Flowchart of keys rotation

- last_rotation_date is the latest date of keys rotation.
- rotation_interval is the interval of keys rotation.

1. SaltStack will check the last rotation date.
2. If the last rotation date is later than rotation interval time ago then SaltStack will send a job to right nodes to rotate keys if needed.
3. The node will try to acquire a lock in the Zookeeper cluster.
4. If the lock was acquired then keys are rotated.
5. If keys were rotated then they are stored in a keys repository.
6. The lock is released.

The simplest way to develop 3rd, 4th and 5th cases in the above algorithm is to use Apache Curator to manage Zookeeper. It has all common algorithms which are used with Zookeeper. These algorithms allow having Keystone instances which can be scaled without worrying about consistency.

3.2 Swift scalability solution

To automate adding capacity gradually it is possible to use a global lock again. Zookeeper is helpful in this case. As same rings have to be around all nodes, ring files are required to be stored in global storage. For this purpose database can be used but it would be an additional unnecessary technology and this technology isn’t designed for such type of tasks. In fact, Swift itself can be used to store its rings. The only thing left to do before the first rings update is to store empty rings in the OpenStack Swift. Thus Swift will use itself as storage for its rings. Again, SaltStack will be used for both ring updates and gradual capacity changes on all servers.

The above technologies are not a requirement. It's just practical recommendations. It is possible to develop any of those technologies or use other technologies instead of them. Also, it's possible to use a distributed file system or synchronization processes instead of object storage to synchronize rings among the cluster.

The main algorithm is based on global locks and distributed, highly available storage. It is important to update rings on a single node in a particular moment of time. Global locks can guarantee that in a particular moment of time only one node modifies rings. Distributed storage can guarantee that all nodes will be eventually consistent.

The final algorithm to update rings will be as follows:

1. SaltStack will send a job to right nodes to update rings if they differ. The job will be sent for example once per hour.
2. If rings differ from the rings which are stored then rings will be updated.

The final algorithm to change rings will be as follows:

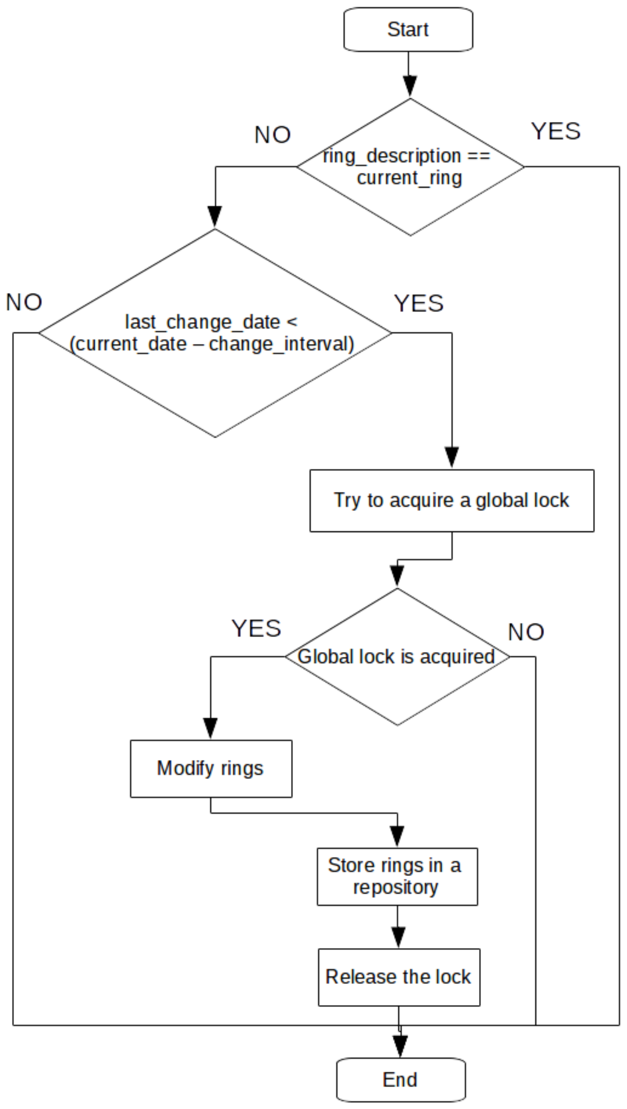
Start
ring_description == current_ring
NO
YES
last_change_date < (current_date – change_interval)
NO
YES
Try to acquire a global lock
Global lock is acquired
YES
NO
Modify rings
Store rings in a repository
Release the lock
End

Fig. 2. Flowchart of rings modification

— ring_description is the full description of expected ring.
— current_ring is the current ring description.
— last_change_date is the latest date of rings modifying.
— change_interval is the interval of rings modifying.

1. SaltStack will compare the description of the ring with the current ring.
2. If there are any differences between ring description and current ring then Salt-Stack will check the last change date.

3. If the last change date is older than change interval time ago then SaltStack will send a job to right nodes to change rings.
4. The node will try to acquire a lock in the Zookeeper cluster.
5. If the lock was acquired then rings would be updated (for example if the drives weight shows a capacity in GB then it is possible to add or remove 25 weights in 30 minutes per drive. Notice that it is highly recommended to calculate “min part hours” and set it to the amount of time which replication takes, in this case 1 hour is enough).
6. If rings were updated then they would be stored in the OpenStack Swift cluster.
7. The lock is released.

Again, to manage Zookeeper it is preferable to use Apache Curator because it has all common algorithms which are used with Zookeeper.

As Apache Curator is a Java library it would be better to use a Java library for OpenStack Swift management. Unfortunately right now there is no any Java library (listed on the OpenStack web-site [7]) which works correctly with a broken TCP connection. After the testing of all Java client libraries for the OpenStack Swift it was found out that they don’t have any solutions for the broken TCP. Those Java libraries are waiting till the broken TCP notification is accepted from OS.

The logical solution for such type problems is to use an API which is more popular. S3 API is much more popular that is why it has a powerful Java client library called AWS Java SDK For Amazon S3 (aws-java-sdk-s3).

Before S3 API can be used with OpenStack Swift it is required to install an additional module called Swift3 which expands Swift with S3 API. After that OpenStack Swift proxy servers must be configured to manage S3 API. Then it will be possible to use AWS Java SDK For Amazon S3 directly with the OpenStack Swift cluster.

These algorithms allow scaling Swift cluster without worrying about both replication storms and inconsistency.

4 Scalability estimates

4.1 OpenStack Keystone scalability estimates

OpenStack Keystone scales almost linearly (bandwidth, CPU load, read ops).

OpenStack Keystone keeps users' data in either LDAP or SQL database. That is why it is hard to scale write operation well. But with fernet tokens write operations aren’t needed to create a token because our tokens aren't stored anywhere.

All read operations scales linearly because they will work with either cache or a local database (the rows will not be locked). In addition keystone instances don't depend on other keystone instances or any additional services. Thus all verification processes scales linearly. CPU load scales linearly because more cores are added with more keystone machines. Bandwidth scales linearly because more physical interfaces are added with more keystone machines.

4.2 OpenStack Swift scalability estimates

OpenStack Swift scales linearly (bandwidth, CPU load, read ops, write ops).

Read and write operations scale linearly because added machines don't depend on other machines. CPU load scales linearly because more cores are added with more keystone machines. Bandwidth scales linearly because more physical interfaces are added with more keystone machines.

5 Conclusion

Object storages play an important role in the life of big web projects. Object storages are widely used to store different unstructured data.

It is very important to have automatically scalable object storage if a business grows rapidly because it is very hard to scale big storage manually.

This paper considered a method to automate one of the most popular open source object storage OpenStack Swift.

Object storage requires authorization. OpenStack Keystone is the de facto standard to have authorization in OpenStack Swift. In multi datacenter clusters there are virtually always different Keystone servers to distribute authorization load that is why a method to scale OpenStack Keystone was researched.

Main tools used were Zookeeper and SaltStack. With SaltStack it is possible to automate these algorithms in a big cluster deployment. Zookeeper can guarantee that only one node will provide changes in one time.

References

1. Introduction to Object Storage. http://docs.openstack.org/admin-guide/objectstorage-intro.html (access date: February 2017)
2. SwiftStack, Inc: The OpenStack Object Storage system, February 2012 – pp. 6-28
3. Keystone the OpenStack Identity Service. http://docs.openstack.org/developer/keystone/ (access date: February 2017)
4. Keystone tokens http://docs.openstack.org/admin-guide/identity-tokens.html (access date: February 2017)
5. Fernet - Frequently Asked Questions. http://docs.openstack.org/admin-guide/identity-fernet-token-faq.html (access date: February 2017)
6. Sam Merritt: Swift Capacity Management, April 09, 2012. https://www.swiftstack.com/blog/2012/04/09/swift-capacity-management/ (access date: February 2017)
7. Software Development Kits. https://wiki.openstack.org/wiki/SDKs (access date: February 2017)
8. Markus Huber, Stewart Kowalski, Marcus Nohlberg, Simon Tjoa: Towards Automating Social Engineering Using Social Networking Sites, August 2009 – 8 p.
9. Wei Chen, Yifei Yuan, Li Zhang: Scalable Influence Maximization in Social Networks under the Linear Threshold Model, December 2010 – 10 p.
10. Steffen Lohmann, Sebastian Dietzold, Philipp Heim, Norman Heino: A Web Platform for Social Requirements Engineering, January 2009 – 7 p.