OpenStack Swift

Openstack Administration 101

INFN

SWIFT

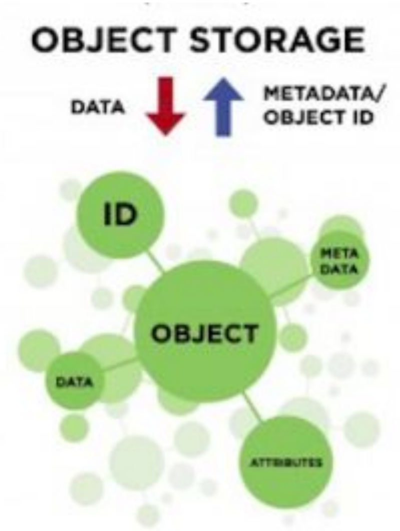
an OpenStack Community Project

Stefano Stalio - LNGS

What is object storage?

Object storage (also known as object-based storage) is a computer data storage architecture that **manages data as objects**, as opposed to other storage architectures like file systems which manages data as a file hierarchy, and block storage which manages data as blocks within sectors and tracks.

Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier.



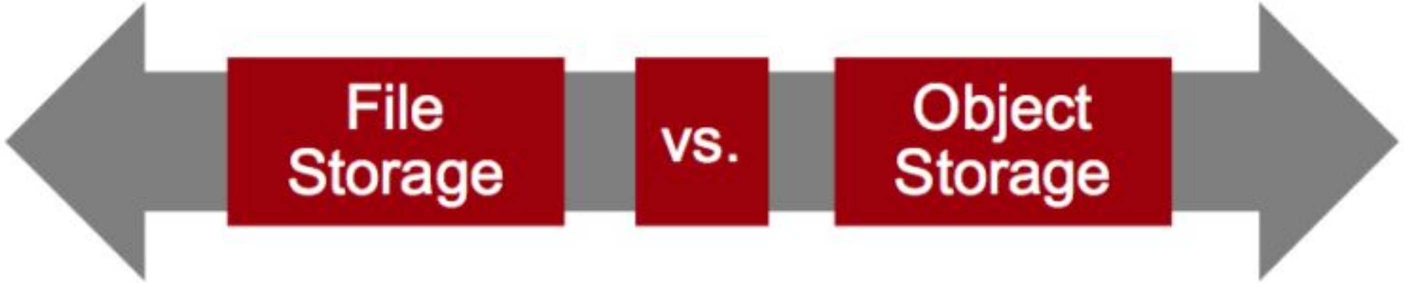
What is object storage?

Object storage can be implemented at multiple levels, including the device level (object-storage device), the system level, and the interface level.

Object storage enables capabilities not addressed by other storage architectures

- interfaces that are directly programmable by the application (**API**s)
- a namespace that can span multiple instances of physical hardware
- data-management functions like data (geo)replication and data distribution at object-level granularity
- authenticated, remote access

File System vs. Object Storage



- Billions of Files
- Amendable Data
  - Locking Mechanisms
  - File System Hierarchy
- Complex to Scale
- Low Storage Efficiency
- **TCO increases exponentially**

- Trillions of Objects
- Immutable Data
  - No Locking Mechanisms
  - One Scalable Storage Pool
  - Scales Uniformly & Simply
  - High Storage Efficiency
- **TCO decreases at scale**

Strong vs eventual consistency

In theoretical computer science, the **CAP theorem**, also named Brewer's theorem after computer scientist Eric Brewer, states that **any distributed data store can only provide two of the following three guarantees**:

- **Consistency**
  - Every read receives the most recent write or an error.
- **Availability**
  - Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition tolerance**
  - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

Strong vs eventual consistency

Storage systems generally use one of two different architectural approaches to provide the scalability, performance and resiliency needed: **eventual consistency** or **strong consistency**.

Eventually Consistent Storage Systems	**Strongly Consistent Storage Systems**
Amazon S3	Block storage
OpenStack Swift	Filesystems

Block storage systems and filesystems are strongly consistent, which is required for databases and other real-time data, but limits their scalability and may reduce availability to data when hardware failures occur.

OpenStack Swift, and other object storage systems, are **eventually consistent**.

What is Object Storage used for?

- **Storage for Unstructured Data** such as music, media files, or text documents. Any type of data that doesn't have a distinct structure to it, has metadata (ex. a song's artist, album title, etc.), and likely won't be manipulaed often is a great fit for Object Storage. Some popular products that use object storage in this category that you might recognize are Netflix and Spotify who use is to store their media files.

- **Backup and Recovery** of critical business applications and workloads. With the rise of digital products, mobile devices and the internet, consumers and enterprises expect applications to always be on and functioning. Due to the highly resilient nature and low cost of Object Storage, many businesses use it to backup their data and workloads to ensure business continuity and to prevent data loss in the event of a disaster.

- **Archived data for long term retention**. Sometimes customers specifically in the financial services industry and healthcare industry have requirements to keep data under retention or records for a certain time period (x number of years). Since this data will persist and not be manipulated frequently, object storage is a perfect cost-effective solution for this use case.

- **Cloud Native Applications** for a persistent data store. As businesses look to modernize their approach to application development in an effort to minimize the time to bring their solutions to market, they need a data store that will scale and not cause costs to sky rocket. Object Storage is a great solution for this as applications can connect directly to the object store and will allow for data to scale simply effectively as the business grows with its number of users and locations.

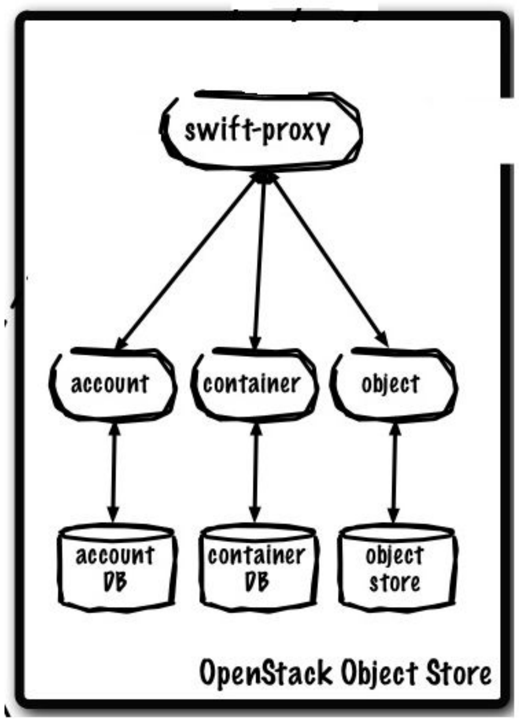
- **Data Lake for Analytics**. With the acceleration of the number of devices generating data (Smartphones, smart devices, IOT sensors etc.), there will be lots of data circulating around that can be processed for intelligent insights. Current storage solutions such as NAS and others are just not effective enough to support this vast growth of data being produced through these various sources. Object Storage can be a great solution for storing all types of data (structured, semi-structured, and unstructured data) that will give businesses a place to dump data before processing and analyzing in order to enable critical insights.

What is OpenStack Swift?

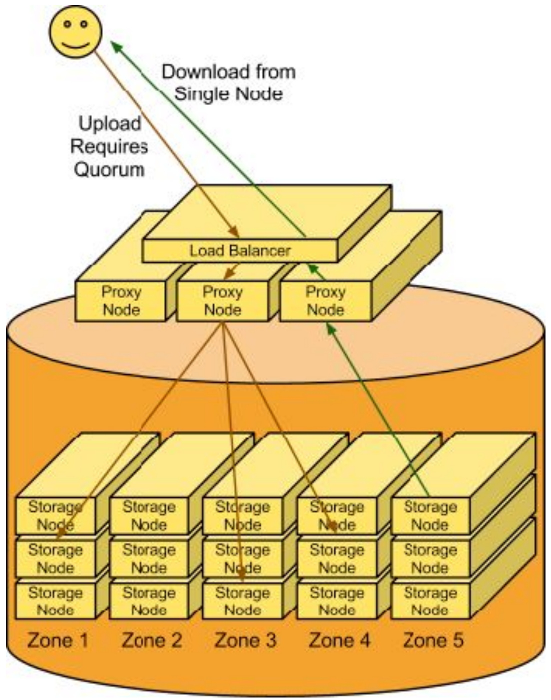
- **Swift is a distributed, eventually consistent object/blob store**. The OpenStack Object Store project, known as Swift, offers cloud storage software so that you can store and retrieve lots of data with a simple API. It's built for scale and optimized for durability, availability, and concurrency across the entire data set. Swift is ideal for storing unstructured data that can grow without bound.
- In August 2009, Rackspace started the development of the precursor to OpenStack Object Storage, as a complete replacement for the *Cloud Files* product. The initial development team consisted of nine developers.
- SwiftStack, an object storage software company, is currently the leading developer for Swift with significant contributions from Intel, Red Hat, NTT, HP, IBM, and more.
-

OpenStack Swift Architecture

Conceptual model



Actual implementation



Proxy Server

The **Proxy server** processes are the public face of Swift as they are the only ones that **communicate with external clients**. As a result they are the first and last to handle an API request. All requests to and responses from the proxy use standard HTTP verbs and response codes.

Proxy servers use a shared-nothing architecture and **can be scaled as needed** based on projected workloads. A minimum of two proxy servers should be deployed for redundancy. Should one proxy server fail, the others will take over.

For example, if a valid request is sent to Swift then the proxy server will verify the request, determine the correct storage nodes responsible for the data (based on a hash of the object name) and send the request to those servers concurrently. If one of the primary storage nodes is unavailable, the proxy will choose an appropriate hand-off node to send the request to. The nodes will return a response and the proxy will in turn return the first response (and data if it was requested) to the requester.

The proxy server process is looking up multiple locations because Swift provides data durability by writing multiple–typically three complete copies of the data and storing them in the system.

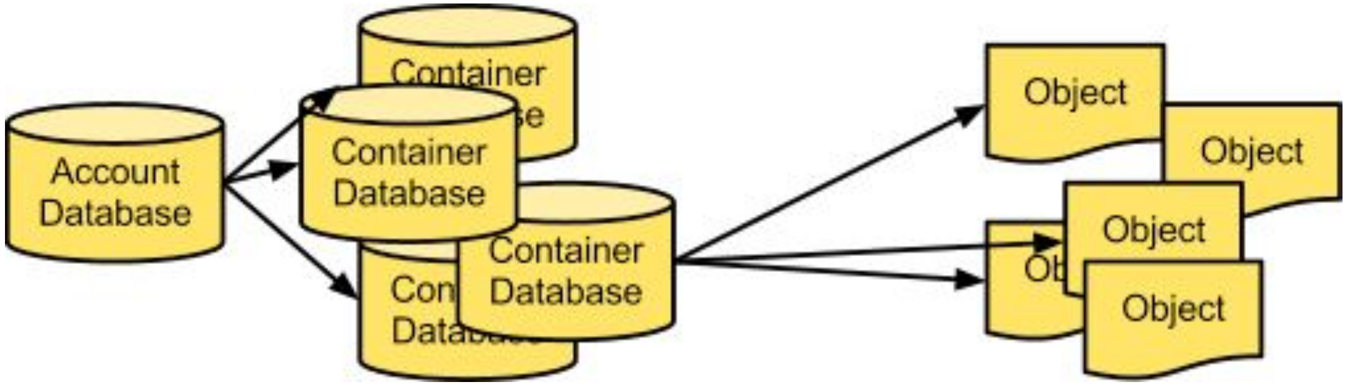
Proxy Server

The **Proxy server** also takes care of the following tasks:

- handle authentication/authorization by interacting with keystone
- expose S3 APIs beside the native APIs, if configured that way
- manage account and container quotas
- manage upload/download of large files that are broken into smaller pieces

Account Server

The account server process handles requests regarding metadata for the individual accounts or the list of the containers within each account. This information is stored by the account server process in SQLite databases on disk.

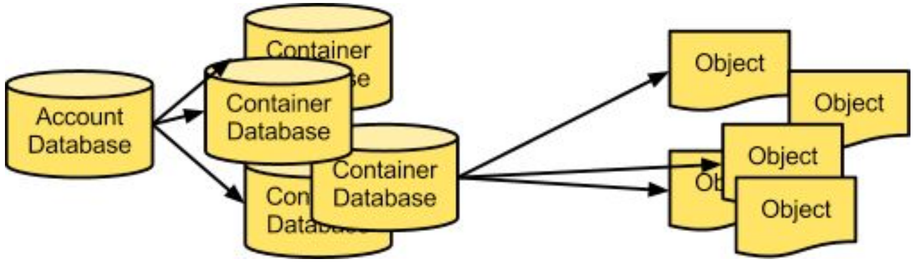


Container Server

The container server process handles requests regarding container metadata or the list of objects within each container.

It’s important to note that the list of objects doesn’t contain information about the location of the object, simply that it belong to a specific container.

Like accounts, the container information is stored as SQLite databases.



Object Server

The object server process is responsible for the actual storage of objects on the drives of its node.

Objects are stored as binary files on the drive using a path that is made up in part of its associated partition and the operation's timestamp.

The timestamp is important as it allows the object server to store multiple versions of an object.

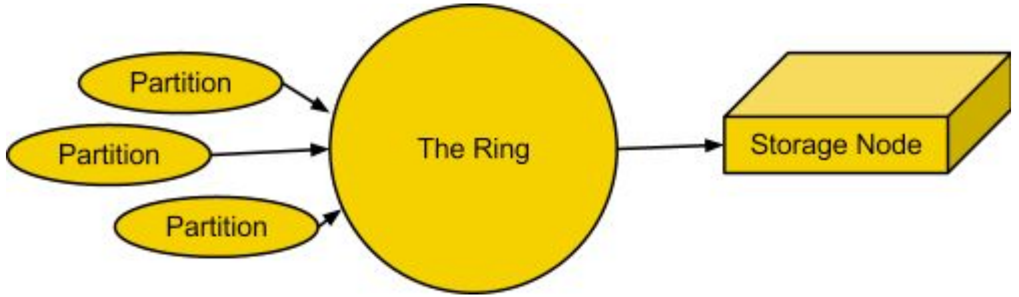
The object’s metadata (standard and custom) is stored in the file’s extended attributes (xattrs) which means the data and metadata are stored together and copied as a single unit.

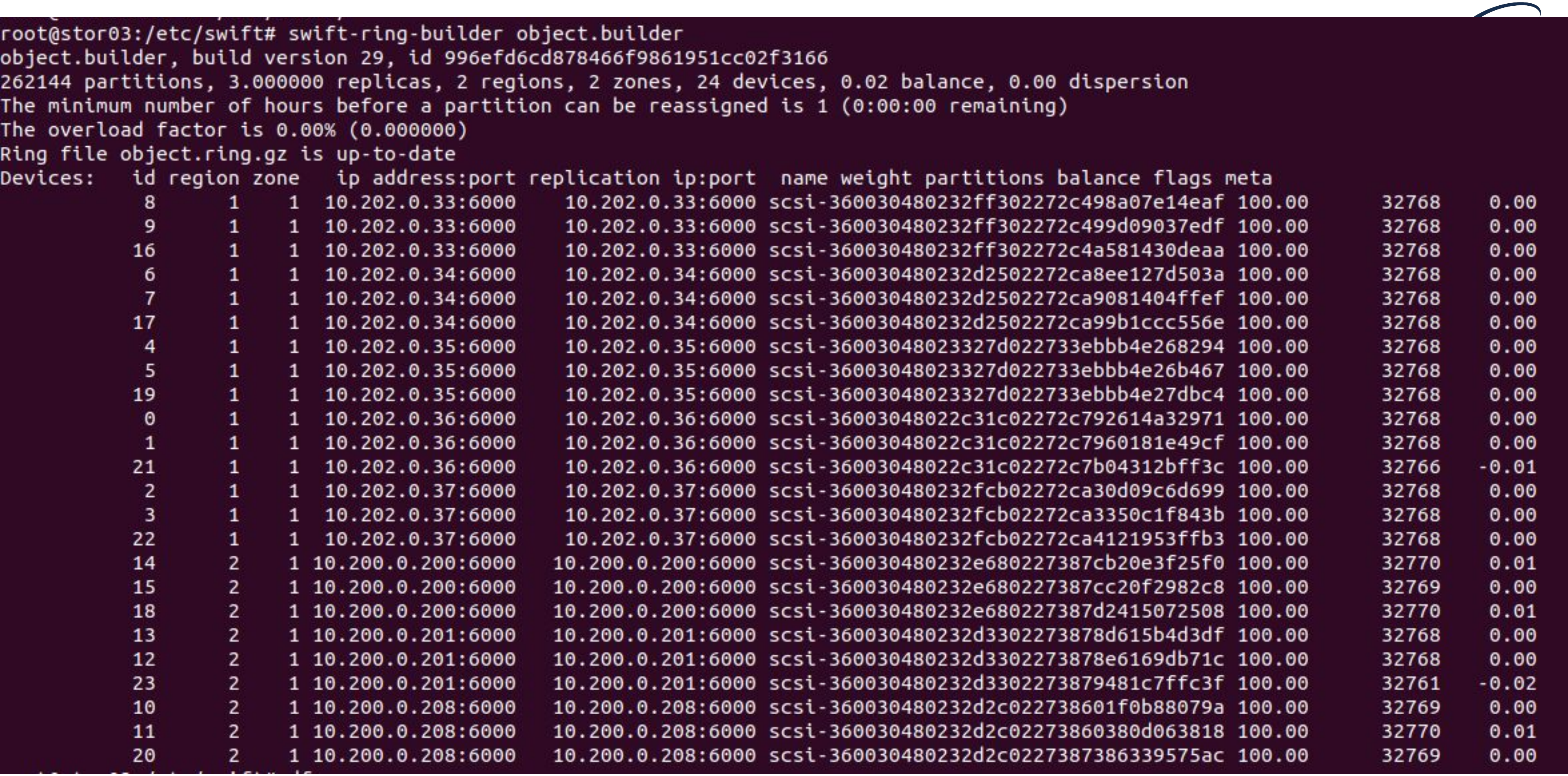
Rings

A ring represents a mapping between the names of entities stored in the cluster and their physical locations on disks.

There are separate rings for accounts, containers, and objects. When components of the system need to perform an operation on an object, container, or account, they need to interact with the corresponding ring to determine the appropriate location in the cluster.

The ring maintains this mapping using zones, devices, partitions, and replicas. Each partition in the ring is replicated, by default, three times across the cluster, and partition locations are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used as handoffs in failure scenarios.





```
root@stor03:/etc/swift# swift-ring-builder object.builder
object.builder, build version 29, id 996efd6cd878466f9861951cc02f3166
262144 partitions, 3.000000 replicas, 2 regions, 2 zones, 24 devices, 0.02 balance, 0.00 dispersion
The minimum number of hours before a partition can be reassigned is 1 (0:00:00 remaining)
The overload factor is 0.00% (0.000000)
Ring file object.ring.gz is up-to-date
Devices:   id region zone   ip address:port replication ip:port  name weight partitions balance flags meta
            8      1    1  10.202.0.33:6000    10.202.0.33:6000 scsi-360030480232ff302272c498a07e14eaf 100.00      32768    0.00
            9      1    1  10.202.0.33:6000    10.202.0.33:6000 scsi-360030480232ff302272c499d09037edf 100.00      32768    0.00
           16      1    1  10.202.0.33:6000    10.202.0.33:6000 scsi-360030480232ff302272c4a581430deaa 100.00      32768    0.00
            6      1    1  10.202.0.34:6000    10.202.0.34:6000 scsi-360030480232d2502272ca8ee127d503a 100.00      32768    0.00
            7      1    1  10.202.0.34:6000    10.202.0.34:6000 scsi-360030480232d2502272ca9081404ffef 100.00      32768    0.00
           17      1    1  10.202.0.34:6000    10.202.0.34:6000 scsi-360030480232d2502272ca99b1ccc556e 100.00      32768    0.00
            4      1    1  10.202.0.35:6000    10.202.0.35:6000 scsi-36003048023327d022733ebbb4e268294 100.00      32768    0.00
            5      1    1  10.202.0.35:6000    10.202.0.35:6000 scsi-36003048023327d022733ebbb4e26b467 100.00      32768    0.00
           19      1    1  10.202.0.35:6000    10.202.0.35:6000 scsi-36003048023327d022733ebbb4e27dbc4 100.00      32768    0.00
            0      1    1  10.202.0.36:6000    10.202.0.36:6000 scsi-3600304822c31c02272c792614a32971 100.00      32768    0.00
            1      1    1  10.202.0.36:6000    10.202.0.36:6000 scsi-3600304822c31c02272c7960181e49cf 100.00      32768    0.00
           21      1    1  10.202.0.36:6000    10.202.0.36:6000 scsi-3600304822c31c02272c7b04312bff3c 100.00      32766   -0.01
            2      1    1  10.202.0.37:6000    10.202.0.37:6000 scsi-360030480232fcb02272ca30d09c6d699 100.00      32768    0.00
            3      1    1  10.202.0.37:6000    10.202.0.37:6000 scsi-360030480232fcb02272ca3350c1f843b 100.00      32768    0.00
           22      1    1  10.202.0.37:6000    10.202.0.37:6000 scsi-360030480232fcb02272ca4121953ffb3 100.00      32768    0.00
           14      2    1 10.200.0.200:6000   10.200.0.200:6000 scsi-360030480232e680227387cb20e3f25f0 100.00      32770    0.01
           15      2    1 10.200.0.200:6000   10.200.0.200:6000 scsi-360030480232e680227387cc20f2982c8 100.00      32769    0.00
           18      2    1 10.200.0.200:6000   10.200.0.200:6000 scsi-360030480232e680227387d2415072508 100.00      32770    0.01
           13      2    1 10.200.0.201:6000   10.200.0.201:6000 scsi-360030480232d3302273878d615b4d3df 100.00      32768    0.00
           12      2    1 10.200.0.201:6000   10.200.0.201:6000 scsi-360030480232d3302273878e6169db71c 100.00      32768    0.00
           23      2    1 10.200.0.201:6000   10.200.0.201:6000 scsi-360030480232d33022738794 81c7ffc3f 100.00      32761   -0.02
           10      2    1 10.200.0.208:6000   10.200.0.208:6000 scsi-360030480232d2c022738601f0b88079a 100.00      32769    0.00
           11      2    1 10.200.0.208:6000   10.200.0.208:6000 scsi-360030480232d2c02273860380d063818 100.00      32770    0.01
           20      2    1 10.200.0.208:6000   10.200.0.208:6000 scsi-360030480232d2c0227387386339575ac 100.00      32769    0.00
```

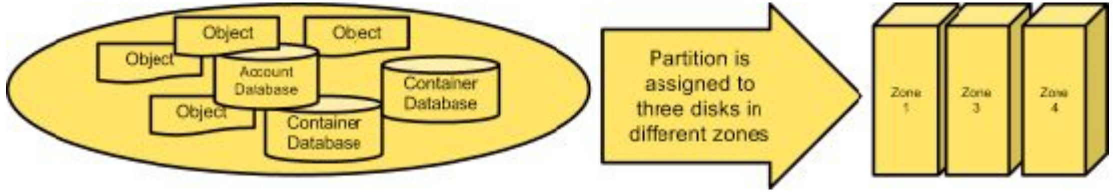
Partitions

A partition is a collection of stored data. This includes account databases, container databases, and objects. Partitions are core to the replication system.

Think of a partition as a bin moving throughout a fulfillment center warehouse. Individual orders get thrown into the bin. The system treats that bin as a cohesive entity as it moves throughout the system. A bin is easier to deal with than many little things. It makes for fewer moving parts throughout the system.

System replicators and object uploads/downloads operate on partitions. As the system scales up, its behavior continues to be predictable because the number of partitions is a fixed number.

Implementing a partition is conceptually simple: a partition is just a directory sitting on a disk with a corresponding hash table of what it contains.

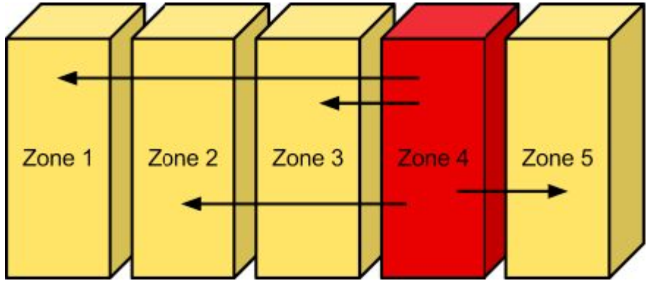


Zones

Object Storage allows configuring zones in order to isolate failure boundaries.

If possible, each data replica resides in a separate zone. At the smallest level, a zone could be a single drive or a grouping of a few drives. If there were five object storage servers, then each server would represent its own zone.

Larger deployments would have an entire rack (or multiple racks) of object servers, each representing a zone. The goal of zones is to allow the cluster to tolerate significant outages of storage servers without losing all replicas of the data.



Unique-as-possible Placement

To ensure the cluster is storing data evenly across its defined spaces (regions, zones, nodes, and disks), **Swift assigns partitions using unique-as-possible placement algorithm**. The algorithm identifies the least-used place in the cluster to place a partitions. First it looks for the least used region, if all the regions contain a partition then it looks for the least used zone, then server (IP:port), and finally the least-used disk and places the partition there. The least-used formula also attempts to place the partition replicas as far from each other as possible.

Once Swift calculates and records the placement of all partitions, the ring can be created.

- One account ring will be generated for a cluster and be used to determine where the account data is located.
- One container ring will be generated for a cluster and be used to determine where the container data is located.
- One object ring will be generated for a cluster and be used to determine where the object data is located.

Storage Policies

Storage Policies allow for some level of segmenting the cluster for various purposes through the creation of multiple object rings. The Storage Policies feature is implemented throughout the entire code base so it is an important concept in understanding Swift architecture.

By supporting multiple object rings, Swift allows the application and/or deployer to essentially segregate the object storage within a single cluster. There are many reasons why this might be desirable:

- Different levels of durability: If a provider wants to offer, for example, 2x replication and 3x replication but doesn’t want to maintain 2 separate clusters, they would setup a 2x and a 3x replication policy and assign the nodes to their respective rings. Furthermore, if a provider wanted to offer a cold storage tier, they could create an erasure coded policy.
- Performance: Just as SSDs can be used as the exclusive members of an account or database ring, an SSD-only object ring can be created as well and used to implement a low-latency/high performance policy.
- Collecting nodes into group: Different object rings may have different physical servers so that objects in specific storage policies are always placed in a particular data center or geography.
- Different Storage implementations: Another example would be to collect together a set of nodes that use a different Diskfile (e.g., Kinetic, GlusterFS) and use a policy to direct traffic just to those nodes.
- Different read and write affinity settings: proxy-servers can be configured to use different read and write affinity options for each policy.

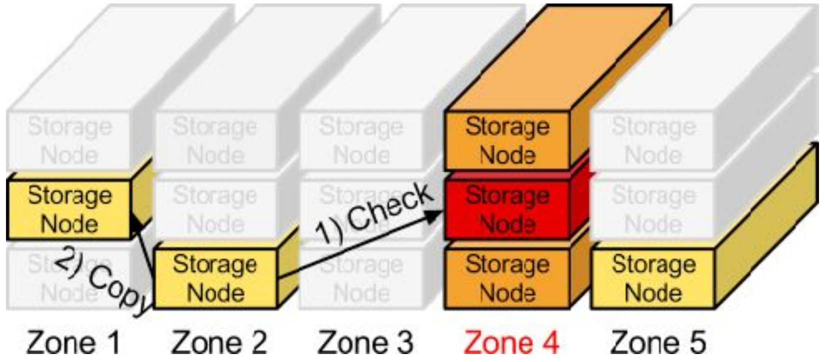
Replicators

In order to ensure that there are three copies of the data everywhere, replicators continuously examine each partition. For each local partition, the replicator compares it against the replicated copies in the other zones to see if there are any differences.

The replicator knows if replication needs to take place by examining hashes. A hash file is created for each partition, which contains hashes of each directory in the partition. For a given partition, the hash files for each of the partition's copies are compared. If the hashes are different, then it is time to replicate, and the directory that needs to be replicated is copied over.

This is where partitions come in handy. With fewer things in the system, larger chunks of data are transferred around (rather than lots of little TCP connections, which is inefficient) and there is a consistent number of hashes to compare.

The cluster has an eventually-consistent behavior where old data may be served from partitions that missed updates, but replication will cause all partitions to converge toward the newest data.



User data management

- OpenStack projects are mapped to Swift accounts
- The only way to give exclusive access to a set of data to a single user is to associate her/him to a personal project
- ACLs allow users to share their data with other users/projects or to allow anonymous access
- ACLs are applied to containers (buckets). ACLs can not restrict access within the account (project)
- Users can set quotas on their containers
- Administrator can set quotas on accounts

Interacting with OpenStack Swift

You use the HTTPS (SSL) protocol to interact with Openstack Swift, and you use standard HTTP calls to perform API operations. You can also use language-specific APIs, which use the RESTful API, that make it easier for you to integrate into your applications.

To assert your right to access and change data in an account, you identify yourself to Openstack Swift by using an authentication token. To get a token, you present your credentials to an authentication service. The authentication service returns a token and the URL for the account. Depending on which authentication service that you use, the URL for the account appears in:

- **OpenStack Identity Service**. The URL is defined in the service catalog.
- Tempauth. The URL is provided in the X-Storage-Url response header.

In both cases, the URL is the full URL and includes the account resource.

Tools for interacting with OpenStack Swift

- Native APIs
- Python SDK
- Openstack Client
- Horizon Dashboard
- High level third party tools

OpenStack Swift S3 APIs

OpenStack Swift was born with its own APIs and with a command line client that is now integrated in the OpenStack client. Both clients use the native Swift APIs.

S3 storage APIs have become the *de facto* standard for accessing Object Storage systems.

In the last few years Swift developed and greatly improved its own S3 API implementation and is now capable of offering a stable and mature S3 interface, although not all features available with its native APIs have been ported to S3.

Swift S3 APIs guarantee better compatibility with many high level Object Storage client tools

Swift/S3 clients

These are just some of the client tools that use the Swift or S3 APIs and that can be exploited on a Swift-based Object Storage deployment:

- rclone
- s3cmd
- aws s3 cli
- cyberduck
- BucketAnywhere (android)
- nextcloud/owncloud

References

- https://en.wikipedia.org/wiki/Object_storage
- https://en.wikipedia.org/wiki/CAP_theorem
- https://www.swiftstack.com
- https://docs.openstack.org