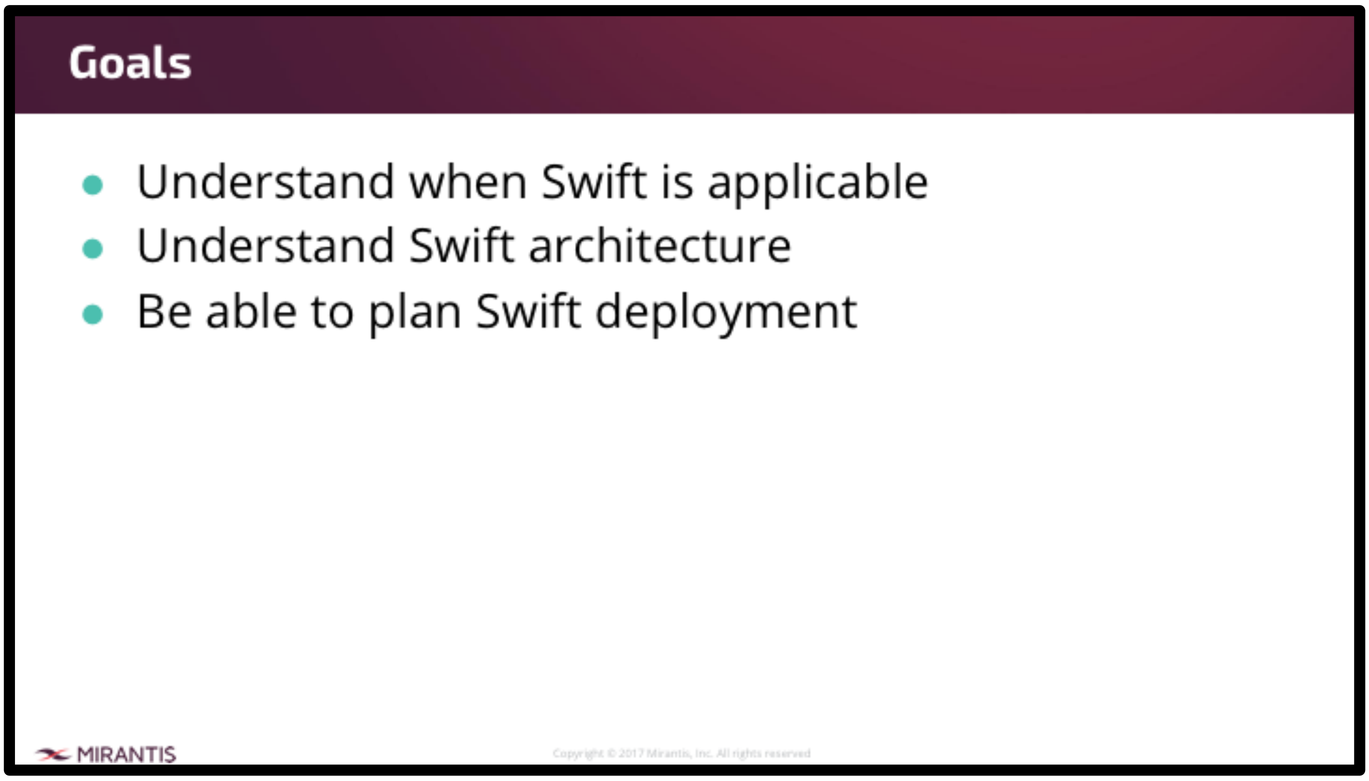
MIRANTIS

Object Storage

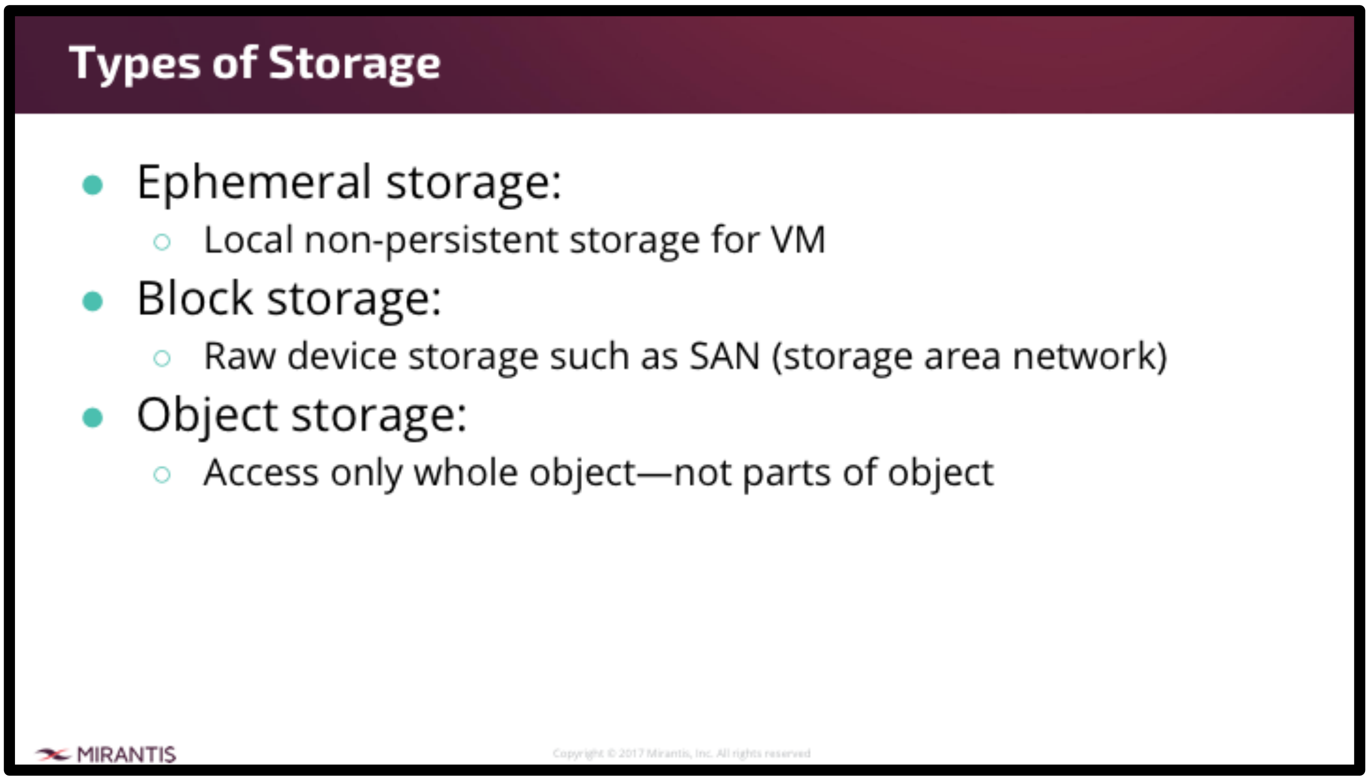
training.mirantis.com



Goals

- Understand when Swift is applicable
- Understand Swift architecture
- Be able to plan Swift deployment

MIRANTIS

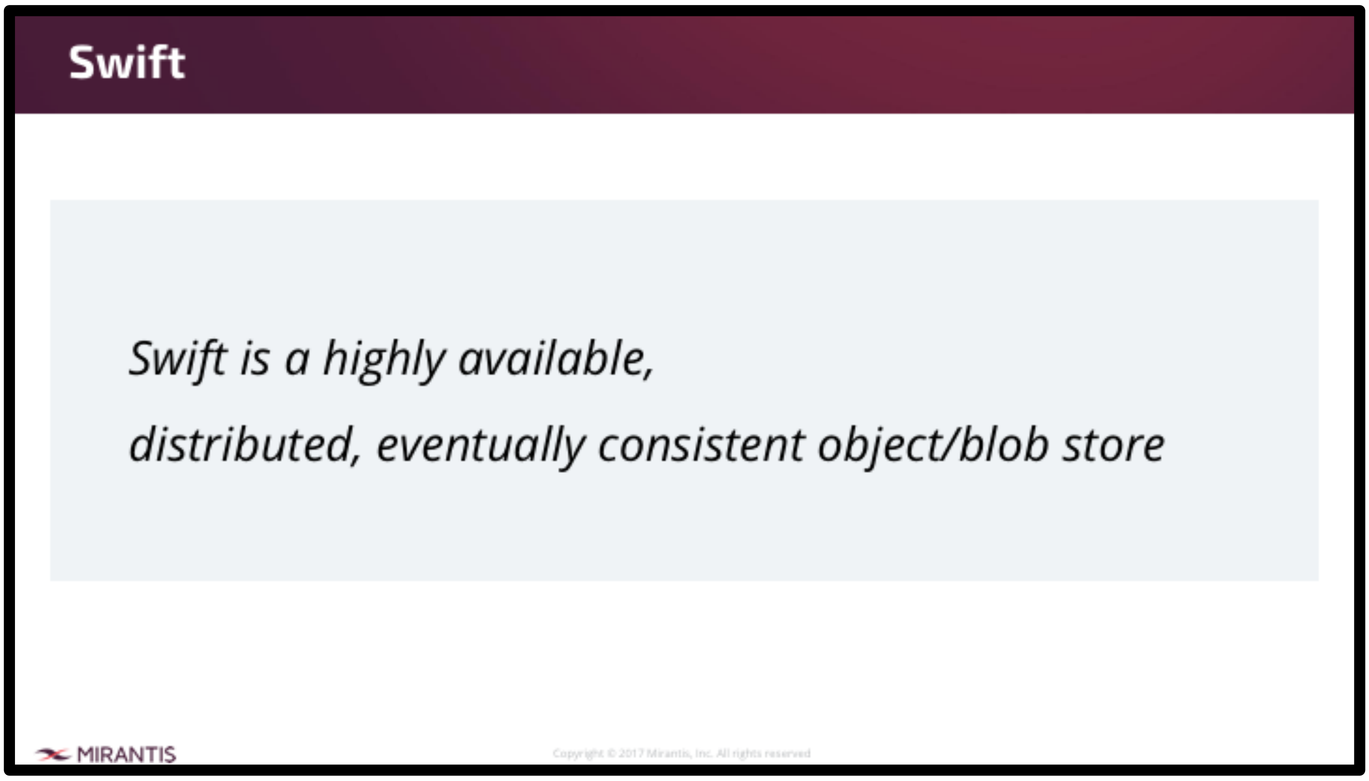


Types of Storage

- Ephemeral storage:
  - Local non-persistent storage for VM
- Block storage:
  - Raw device storage such as SAN (storage area network)
- Object storage:
  - Access only whole object—not parts of object

MIRANTIS

Many people keep mixing up cinder and swift. It’s worthwhile to review different storage types once again. Answer any questions.

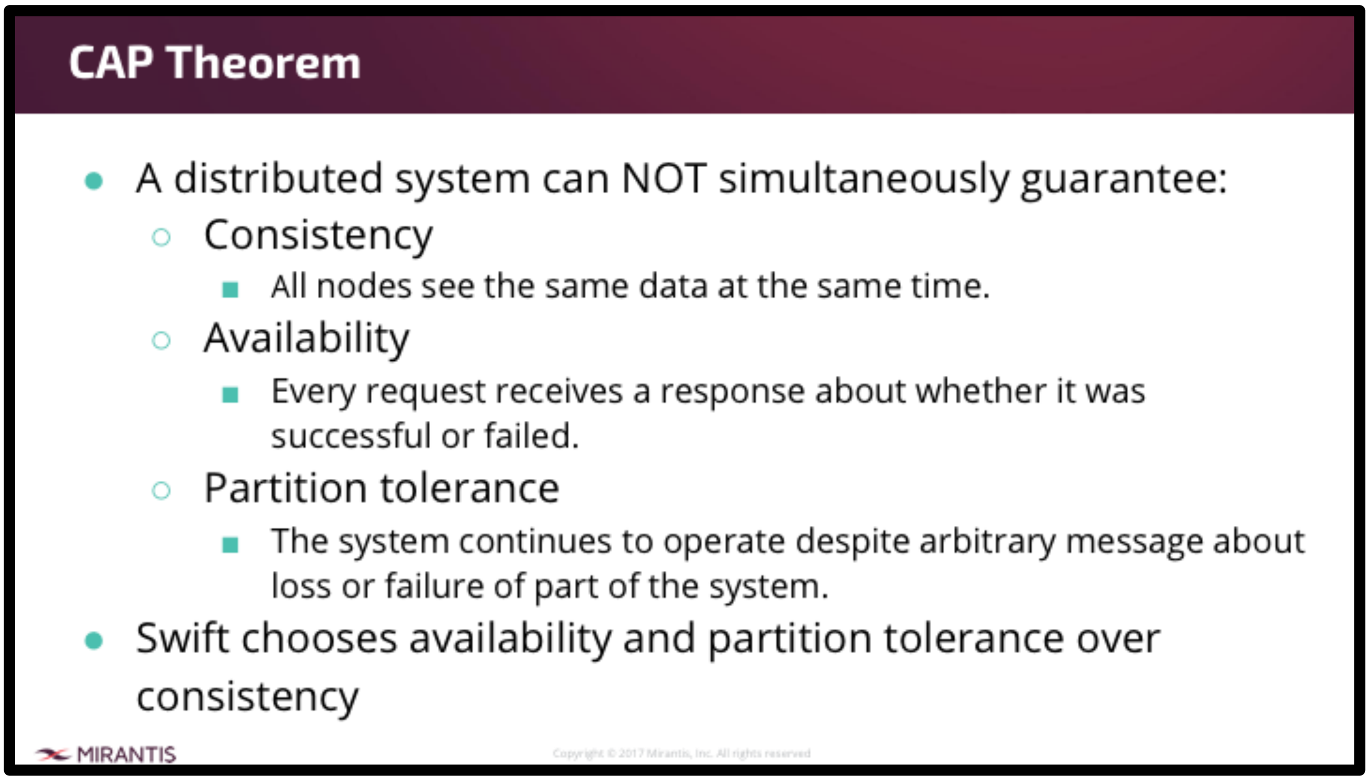


Swift

Swift is a highly available,

distributed, eventually consistent object/blob store

MIRANTIS

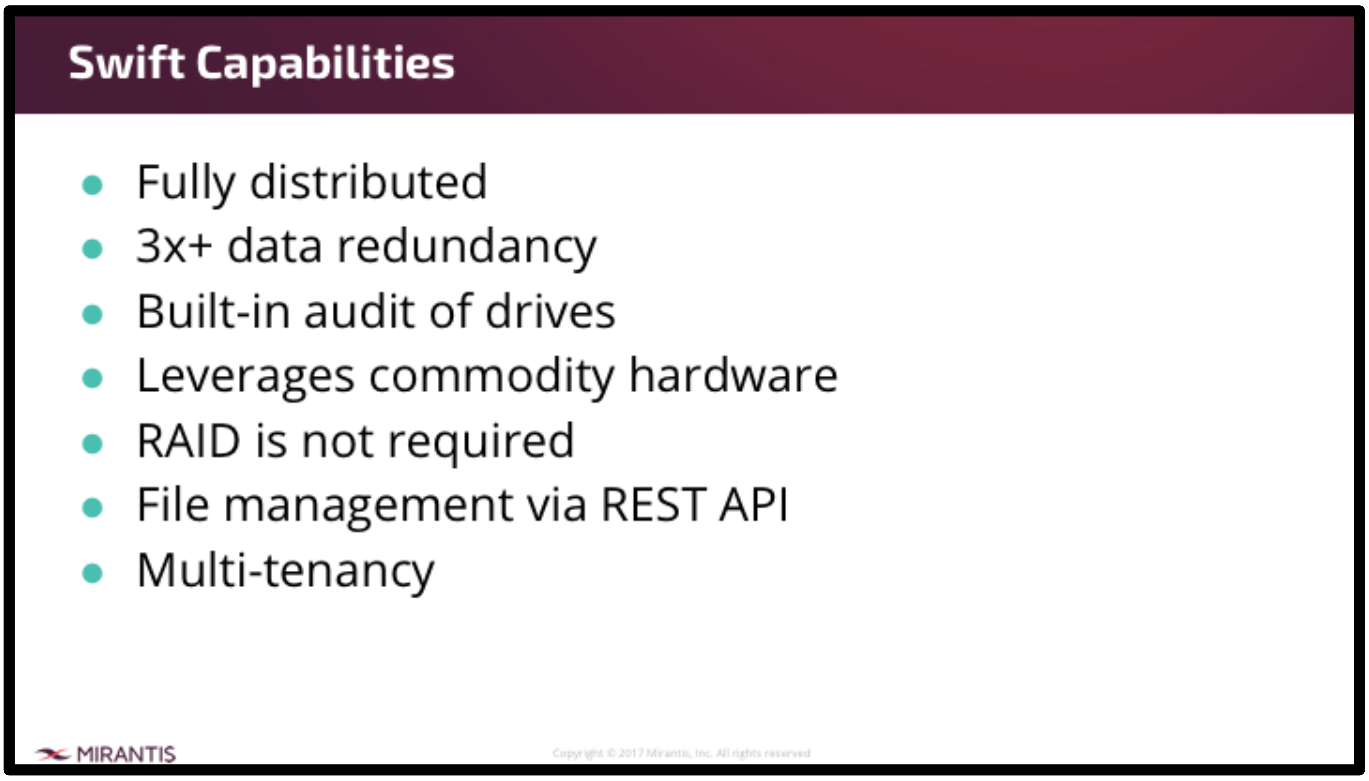


CAP Theorem

- A distributed system can NOT simultaneously guarantee:
  - Consistency
    - All nodes see the same data at the same time.
  - Availability
    - Every request receives a response about whether it was successful or failed.
  - Partition tolerance
    - The system continues to operate despite arbitrary message about loss or failure of part of the system.
- Swift chooses availability and partition tolerance over consistency

MIRANTIS

Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. (http://en.wikipedia.org/wiki/Eventual_consistency)



Swift Capabilities

- Fully distributed
- 3x+ data redundancy
- Built-in audit of drives
- Leverages commodity hardware
- RAID is not required
- File management via REST API
- Multi-tenancy

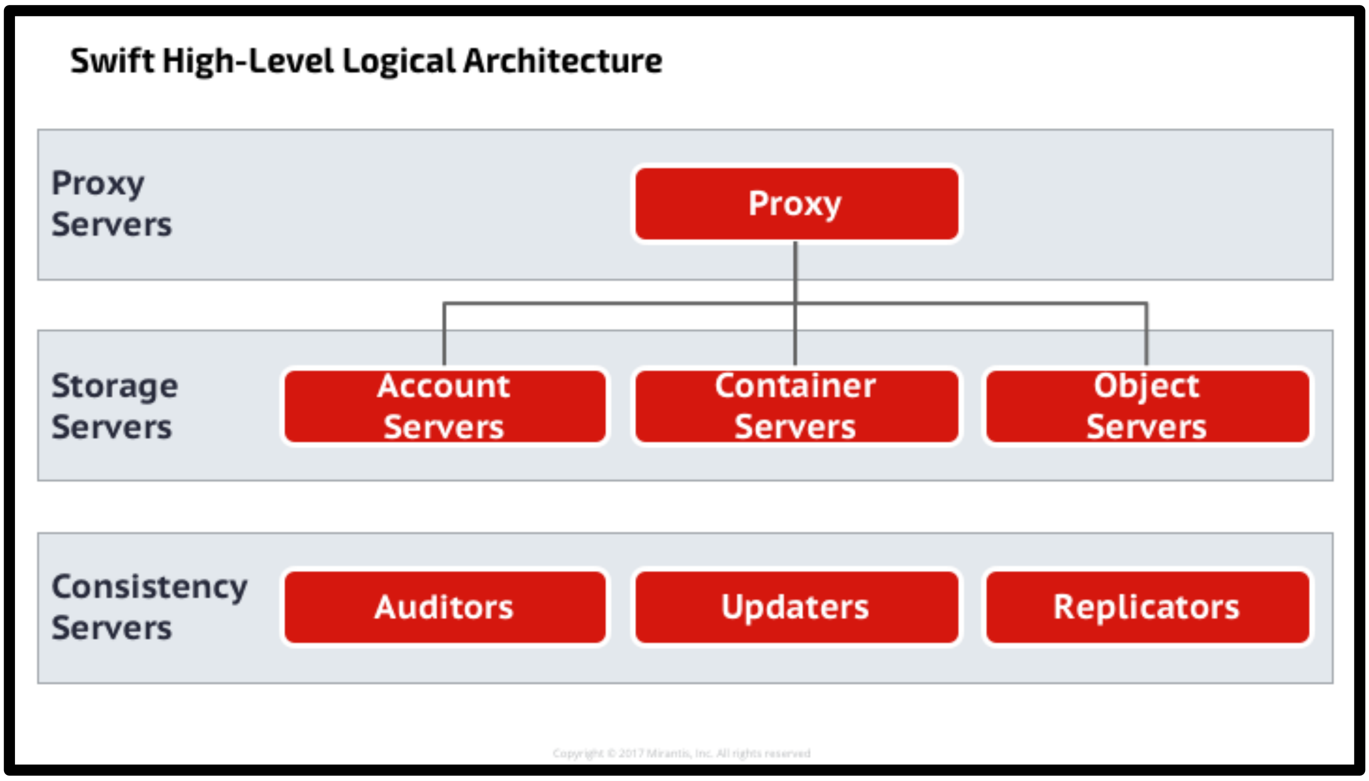
MIRANTIS

Talk about CAP theorem here (http://en.wikipedia.org/wiki/CAP_theorem):
In theoretical computer science, the CAP theorem, also known as Brewer's theorem, states that **it is impossible for a distributed computer system to simultaneously provide all three** of the following guarantees:
- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response about whether it was **successful or failed**) – no guarantee of success
- **Partition tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)

Here we talk about Partition tolerance and Availability.
Swift sacrifices Consistency in some cases - **Eventually Consistent**
Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
(http://en.wikipedia.org/wiki/Eventual_consistency)

http://programmerthoughts.com/openstack/swift-tech-overview/
Swift achieves high scalability by relaxing constraints on consistency. While swift provides read-your-writes consistency for new objects, **listings and aggregate metadata (like usage information) may not be immediately accurate**. Similarly,

reading an object that has been overwritten with new data may return an older version of the object data. However, swift provides the ability for the client to request the most up-to-date version at the cost of request latency.



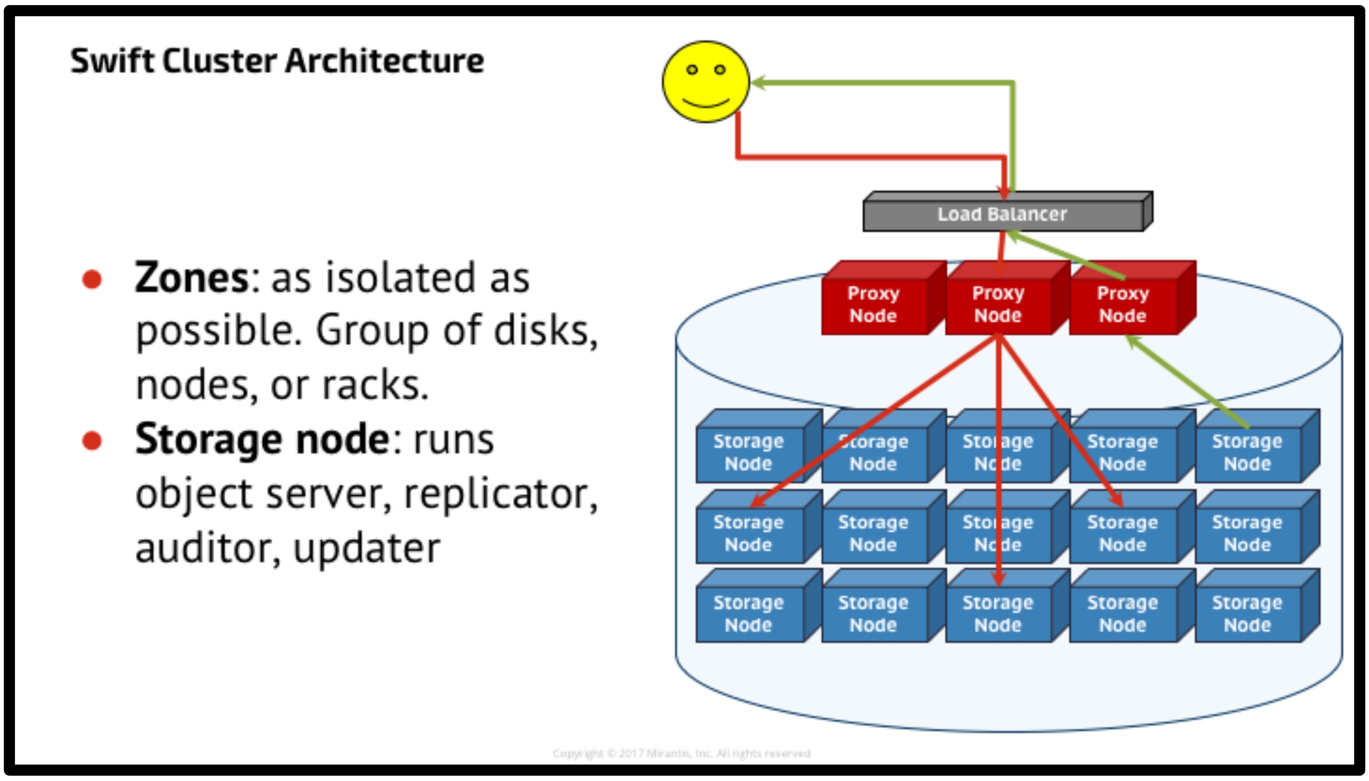
The **proxy server** is a server process that provides the swift API.
The swift **storage servers** provide the on-disk storage for the cluster. There are three types of storage servers in swift: account, container, and object. Each of these servers provide an internal REST-full API.
Swift's **consistency servers** are responsible for finding and correcting errors caused by both data corruption and hardware failures.

The Container Server's primary job is to handle listings of objects. It doesn't know where those objects are, just what objects are in a specific container. The listings are stored as sqlite database files, and replicated across the cluster similar to how objects are. Statistics are also tracked that include the total number of objects, and total storage usage for that container.

The Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes (xattrs). This requires that the underlying filesystem choice for object servers support xattrs on files. Some filesystems, like ext3, have xattrs turned off by default. Each object is stored using a path derived from the object name's hash and the operation's timestamp. Last write always wins, and ensures that the latest object version will be served. A deletion is also treated as a version of the file (a 0 byte file ending with ".ts", which stands for tombstone). This ensures that deleted files are replicated correctly and older versions don't magically reappear due to failure scenarios.

The Account Server's primary job is to handle listings of containers. The listings are stored as sqlite database files, and replicated across the cluster similar to how objects are.

The account/container/object auditor crawls the local account/container/object system checking the integrity of account/container/object objects. If corruption is found (in the case of bit rot, for example), the file is quarantined, and replication will replace the bad file from another replica.



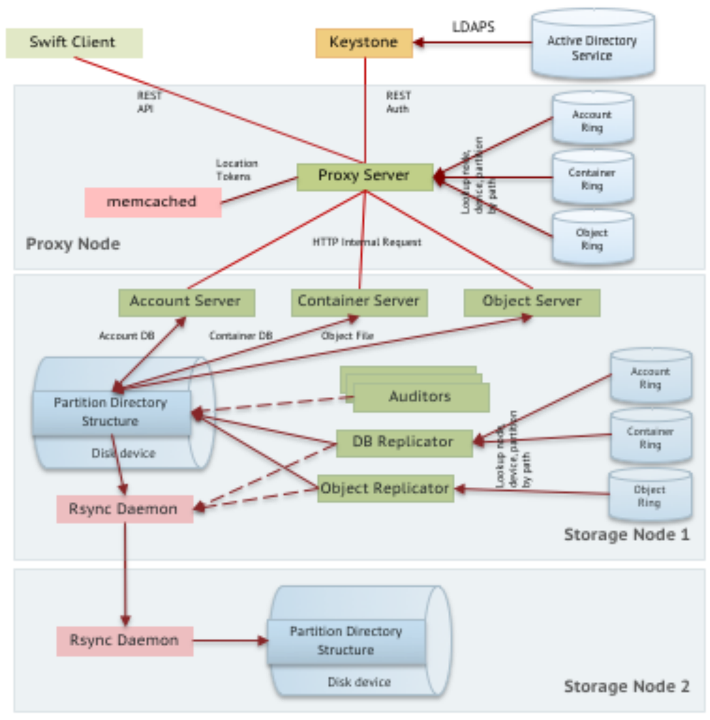
Swift Cluster Architecture

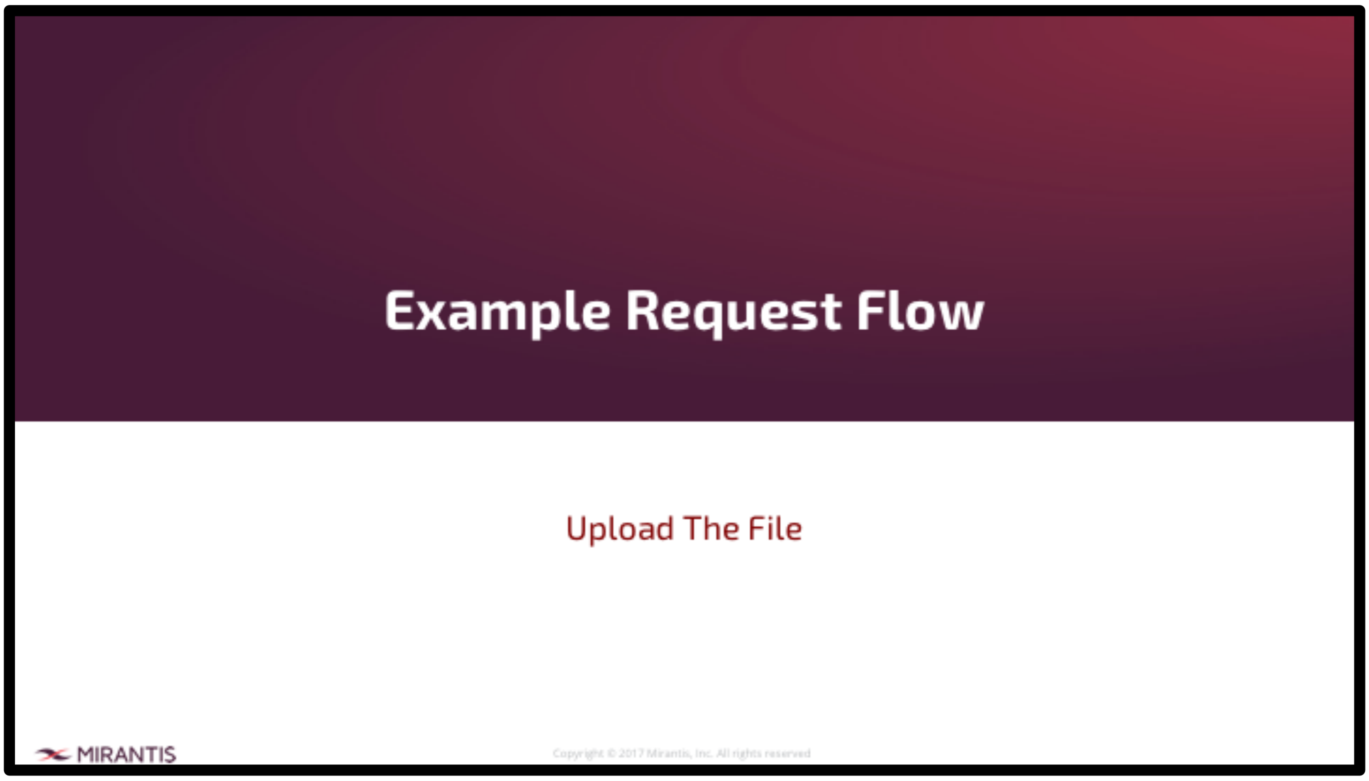
- **Zones**: as isolated as possible. Group of disks, nodes, or racks.
- **Storage node**: runs object server, replicator, auditor, updater

How swift nodes are deployed in the real life.

Storage nodes run all the servers we mentioned in the previous slide, so contrary to what one may think, they are actually cpu and network io intensive.

Swift Detailed Architecture

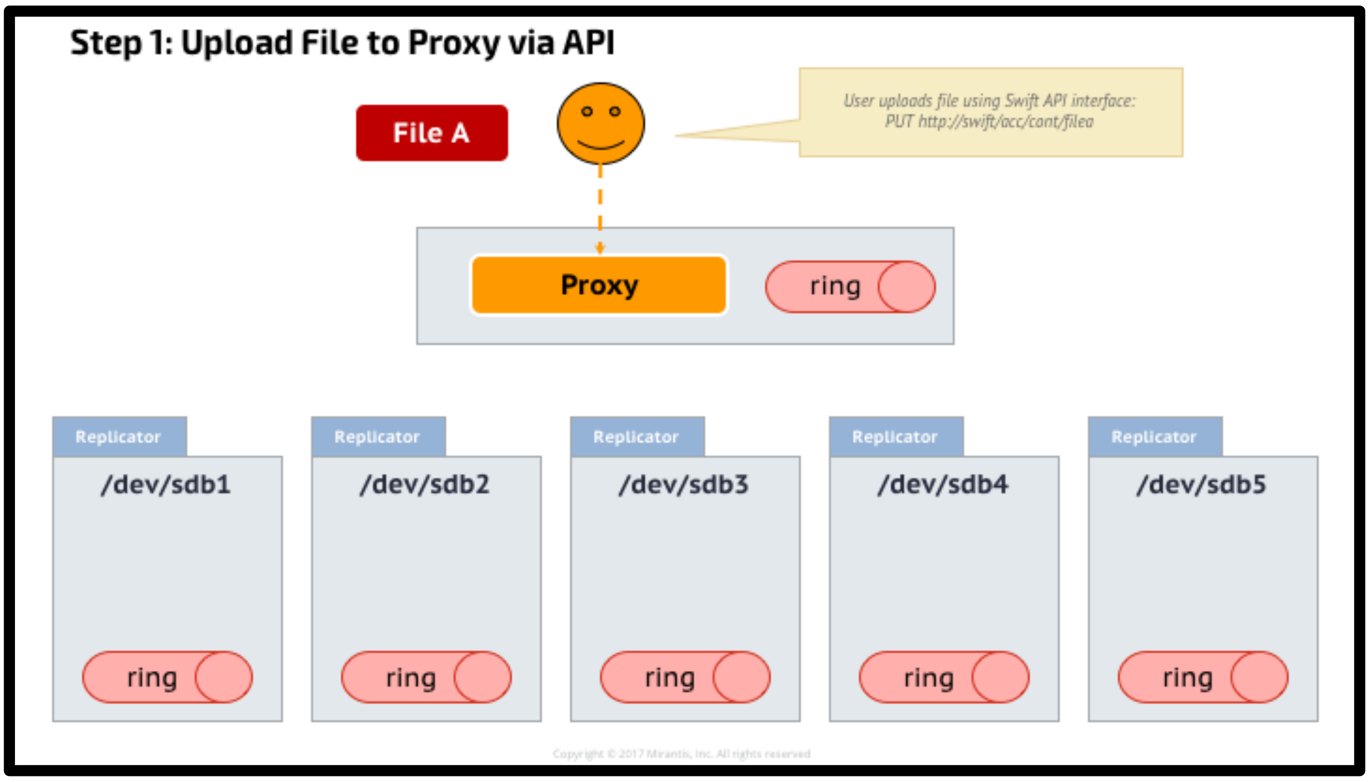




Example Request Flow

Upload The File

MIRANTIS


Step 1: Upload File to Proxy via API
File A
User uploads file using Swift API interface:
PUT http://swift/acc/cont/filea
Proxy
ring
Replicator
/dev/sdb1
ring
Replicator
/dev/sdb2
ring
Replicator
/dev/sdb3
ring
Replicator
/dev/sdb4
ring
Replicator
/dev/sdb5
ring
Copyright © 2017 Mirantis, Inc. All rights reserved

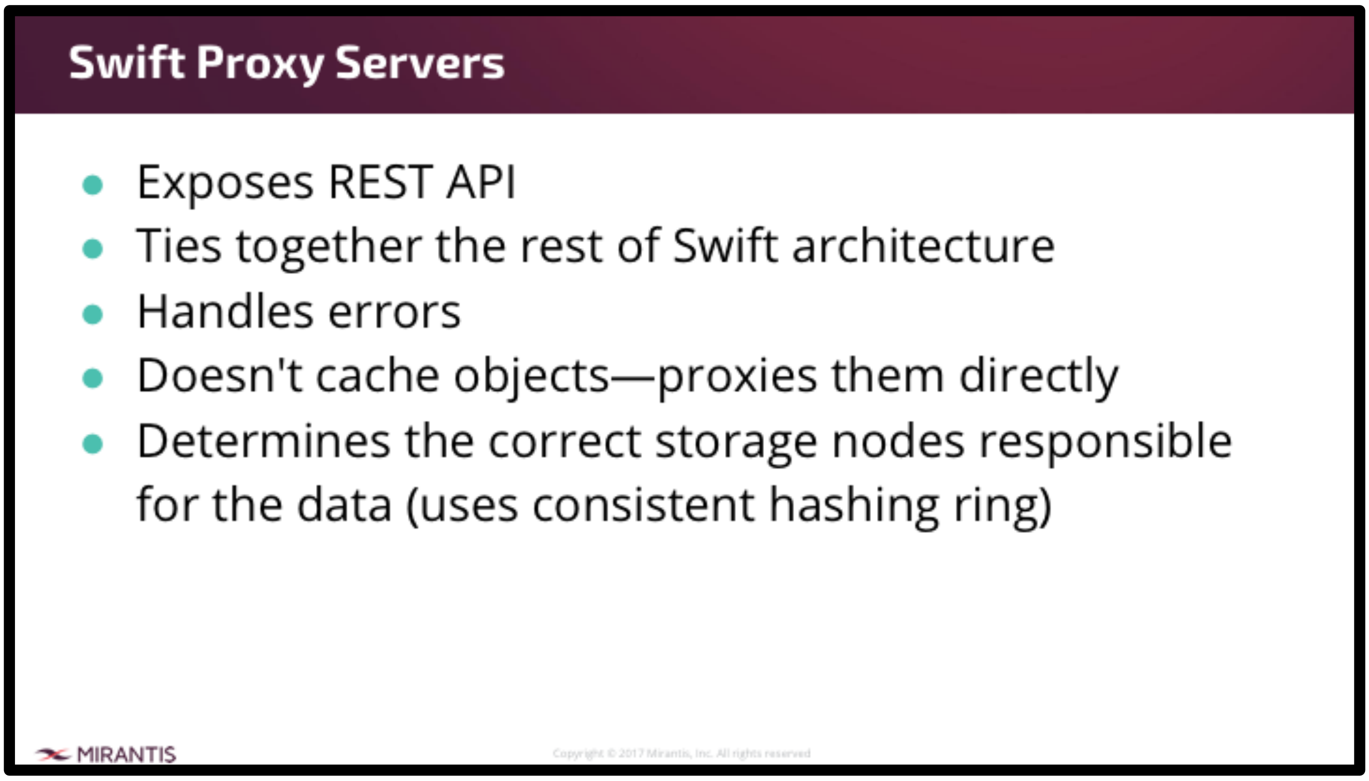
http://docs.openstack.org/trunk/openstack-object-storage/admin/content/installing-and-configuring-storage-nodes.html

Use a single partition per drive. For example, in a server with 12 disks you may use one or two disks for the operating system which should not be touched in this step. The other 10 or 11 disks should be partitioned with a single partition, then formatted in XFS.

Each drive is a single physical partition:

```
mkfs.xfs -i size=1024 /dev/sdb1
echo "/dev/sdb1 /srv/node/sdb1 xfs noatime,nodiratime,nobarrier,logbufs=8 0 0" >>
/etc/fstab
```

We show only 1 drive per server, but the number of them might and probably will be more.



Swift Proxy Servers

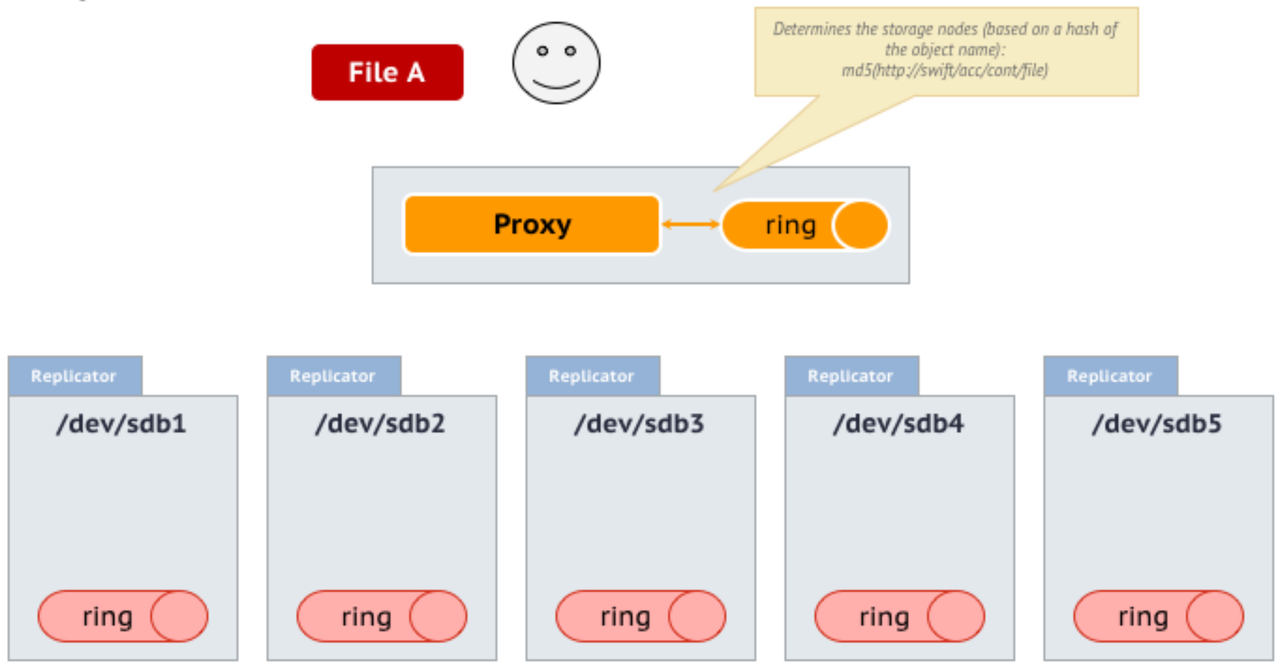
- Exposes REST API
- Ties together the rest of Swift architecture
- Handles errors
- Doesn't cache objects—proxies them directly
- Determines the correct storage nodes responsible for the data (uses consistent hashing ring)

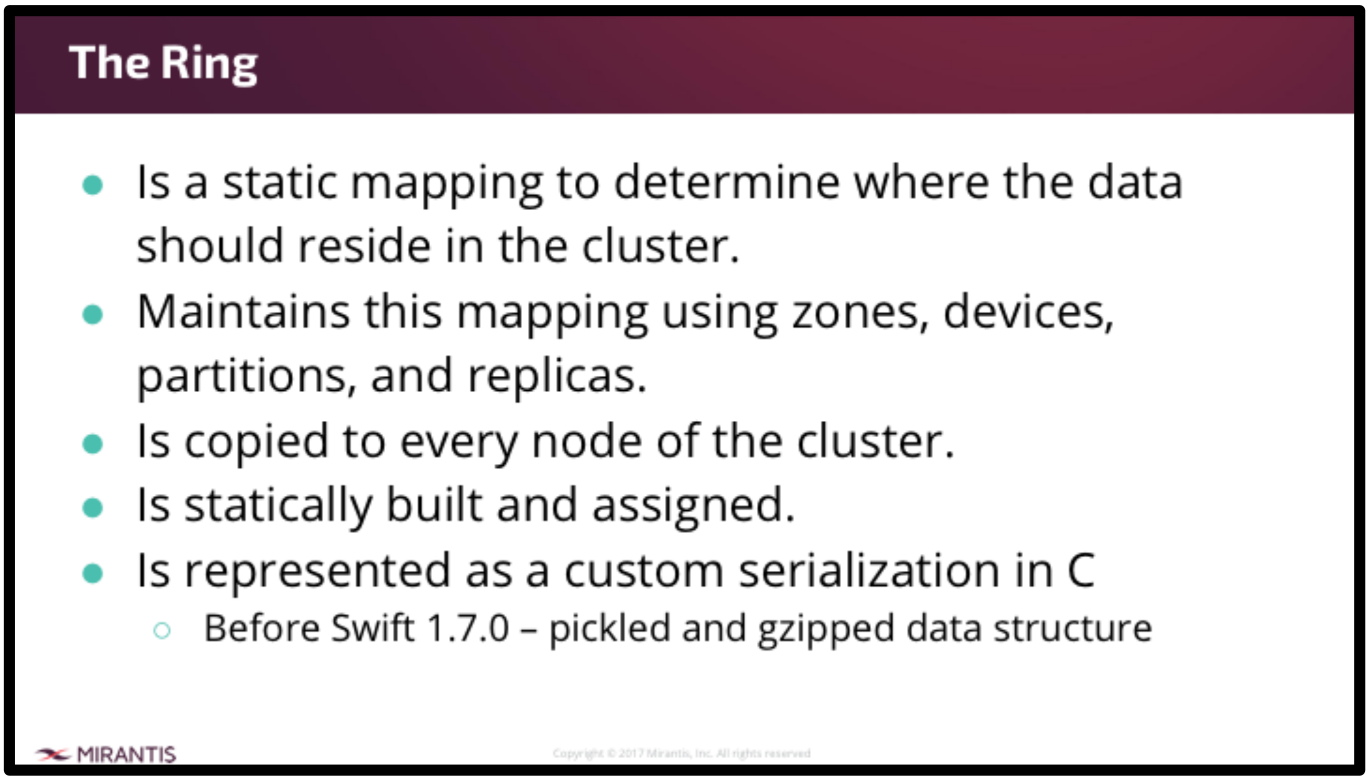
MIRANTIS

As the only system in the swift cluster that communicates with clients, the proxy is responsible for coordinating with the storage servers and replying to the client with appropriate messages (Availability).
The proxy is an HTTP server that implements swift's REST-ful API. All messages to and from the proxy use standard HTTP verbs and response codes. This allows developers building clients to interact with swift in a simple, familiar way.

DHT – Distributed Hash Tables

Step 2: Calculate Where the File Should Go




The Ring

- Is a static mapping to determine where the data should reside in the cluster.
- Maintains this mapping using zones, devices, partitions, and replicas.
- Is copied to every node of the cluster.
- Is statically built and assigned.
- Is represented as a custom serialization in C
  - Before Swift 1.7.0 – pickled and gzipped data structure

MIRANTIS

Can specify number of replicas when creating the ring
Checksum – md5sum of the bytes – when get/put the objects

128 bits

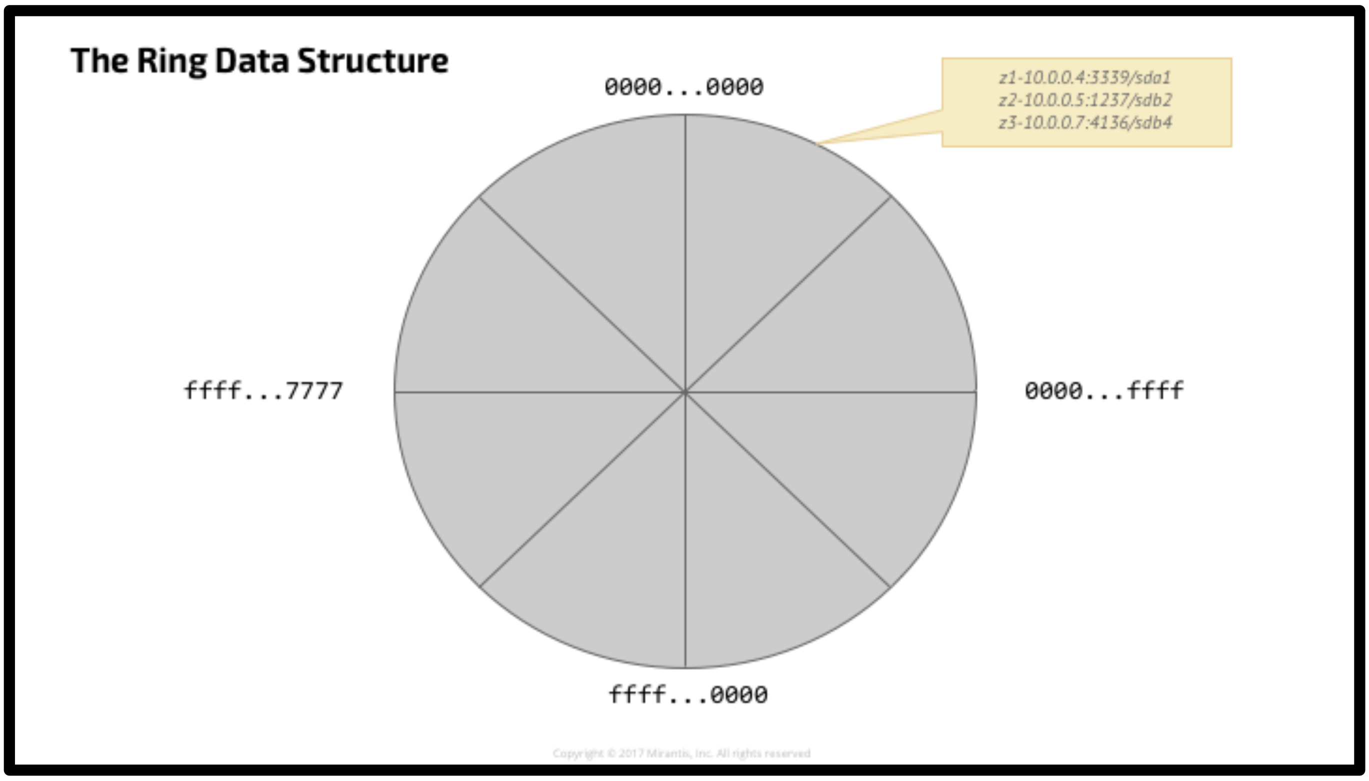
20 megs is avg size of ring
Err on the side of too big and use more memory

Md5 hash <- hash path

Ring has a static mapping

Ring files are endian aware of the machine that generated it and will appropriately byteswap if necessary

on the machine that will deserialize it.



http://en.wikipedia.org/wiki/Hash_function
A hash (hash value, hash code, hash sum, checksum) - the value returned by hash function.
A hash function is any algorithm that maps data of variable length (like name (URL) in Swift case) to data of a fixed length.
Md5sum is used as a hash function in Swift (128-bit (16byte) length – 32 hex digits)

http://greg.brim.net/page/building_a_consistent_hashing_ring.html
Probably the simplest hash is just a modulus of the hash. For instance, if all hashes are numbers (which they are for md5sum) and you have two machines you wish to distribute data to, you could just put all odd numbered hashes on one machine and even numbered hashes on the other. Assuming you have a balanced number of odd and even numbered hashes, and a balanced data size per hash, your data would be balanced between the two machines.
Another benefit of using an MD5 is that the resulting hashes have a known even distribution, meaning your hashes will be evenly distributed without worrying about keeping the hash values themselves evenly distributed.
Const – a lot of reshuffling of data when adding a node

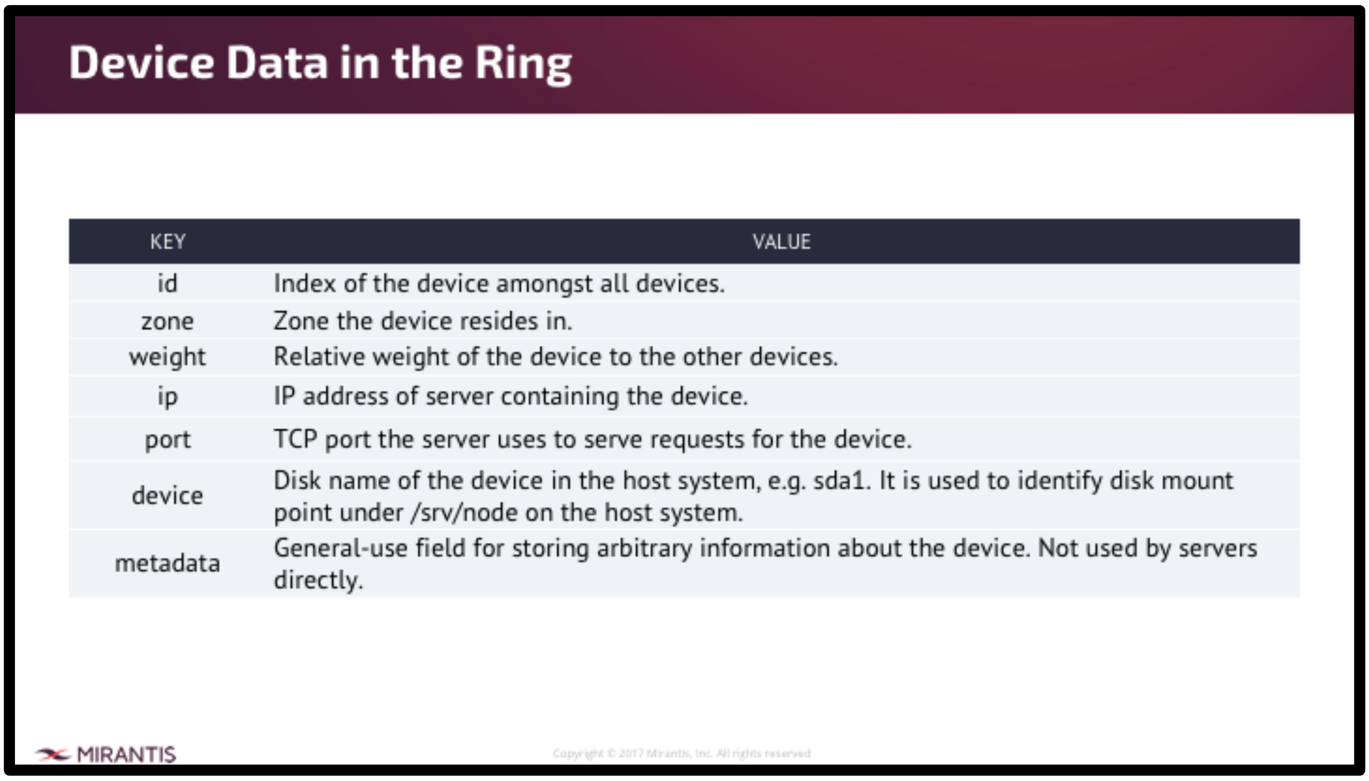
This is where the DHT, consistent hashing and the "ring" really come in. We can assign **ranges of hashes** directly to nodes and then use an algorithm that minimizes the

changes to those ranges.
The way of displaying the such DHT is in a form of a circle – “ring”.
Back to our small scale, our hashes range from 0 to 2^128. We have two nodes and we'll assign data ids 0-2^64 to node 0 and 2^64+1-2^128 to node 1. Later, when we add node 2, we can take half the data ids from node 0 and half from node 1, minimizing the amount of data that needs to move.
Okay, that is better. But still, moving 50% of our data to add 1% capacity is not very good.

We can minimize the change to a node's assigned range by assigning several smaller ranges instead of the single broad range we were before. This can be done by creating "virtual nodes" for each node. The picture above each node has 2 “virtual nodes” and there 4 “real nodes” on the system.
Those “virtual nodes” are called partitions.
It’s also easier to always keep the partition count a power of two. This makes it easy to just use bit manipulation on the hash to determine the partition rather than modulus.

http://julien.danjou.info/blog/2012/openstack-swift-consistency-analysis
We can also define a weight for each node, in order for some nodes to get more partitions than others.
Unfortunately, the number of virtual nodes created at the beginning can never change for the life of the cluster without a lot of careful work.
A good rule to follow might be to calculate 100 partitions to each real node (physical drive) at maximum capacity. This would allow you to alter the load on any given node by 1%, even at max capacity, which is pretty fine tuning. It might seem like we'd use up way too much memory, but the only structure this affects is the virtual node to real node mapping.

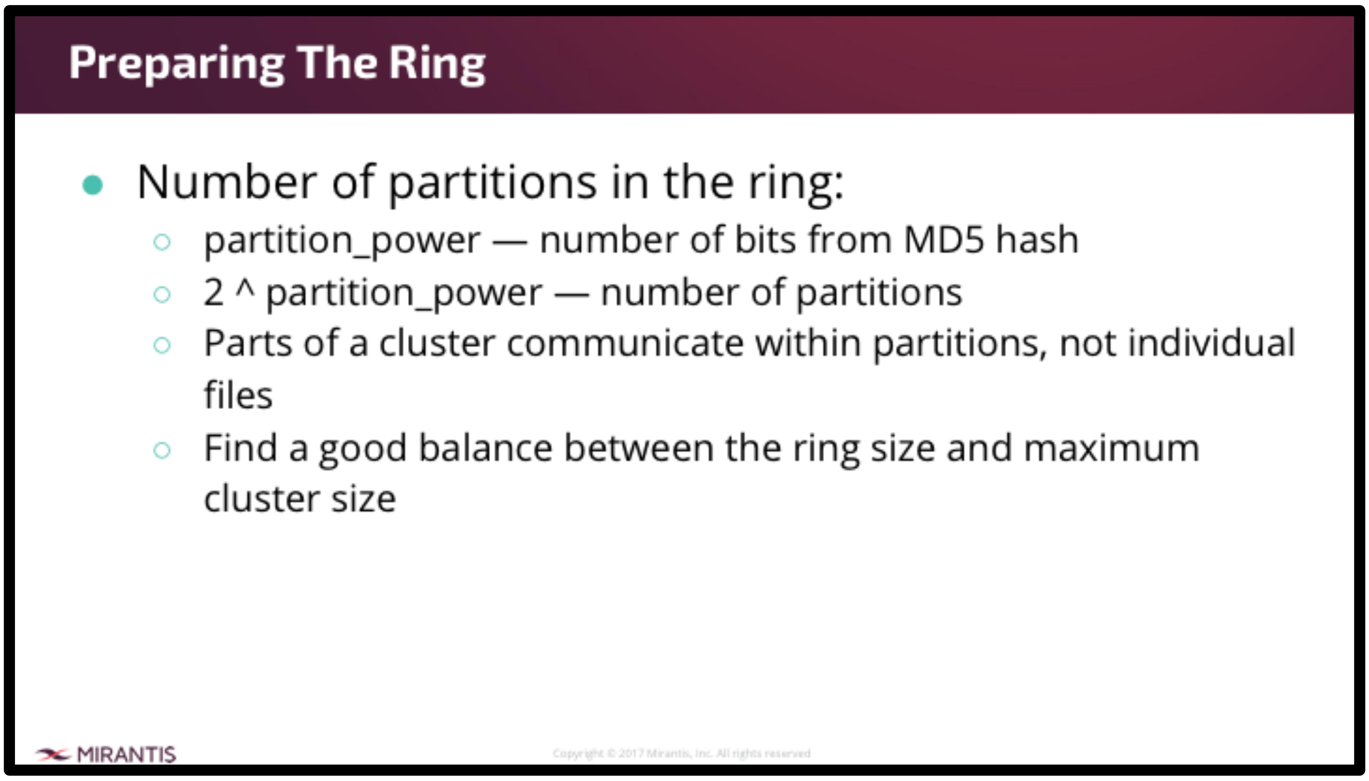
Now, to assure availability and partitioning (as seen in the *CAP theorem*) we also want to store replicas of our objects. By default, Swift stores 3 copies of every objects, but that's configurable.
In that case, we need to store each partition defined above not only on 1 node, but on 2 others. So Swift adds another concept: zones. A zone is an isolated space that does not depends on other zone, so in case of an outage on a zone, the other zones are still available. Concretely, a zone is likely to be a disk, a server, or a whole cabinet, depending on the size of your cluster. It's up to you to chose anyway.
Consequently, each partitions has not to be mapped to 1 host only anymore, but to N hosts. Each node will therefore store this number of partitions:
number of partition stored on one node = number of replicas × total number of partitions ÷ number of node



Device Data in the Ring

KEY	VALUE
id	Index of the device amongst all devices.
zone	Zone the device resides in.
weight	Relative weight of the device to the other devices.
ip	IP address of server containing the device.
port	TCP port the server uses to serve requests for the device.
device	Disk name of the device in the host system, e.g. sda1. It is used to identify disk mount point under /srv/node on the host system.
metadata	General-use field for storing arbitrary information about the device. Not used by servers directly.

MIRANTIS


Preparing The Ring

- Number of partitions in the ring:
  - partition_power — number of bits from MD5 hash
  - 2 ^ partition_power — number of partitions
  - Parts of a cluster communicate within partitions, not individual files
  - Find a good balance between the ring size and maximum cluster size

MIRANTIS

http://docs.openstack.org/trunk/openstack-object-storage/admin/content/preparing-the-ring.html

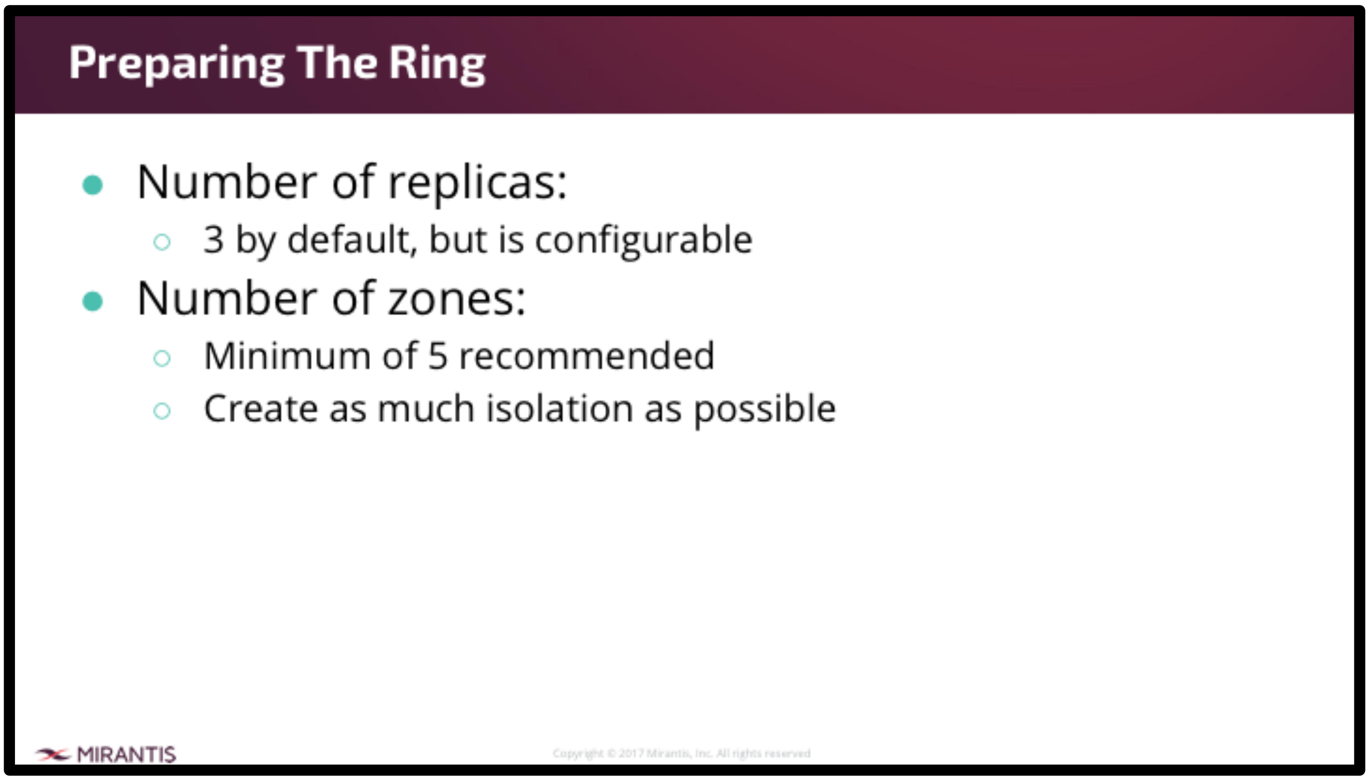
The first step is to determine the number of partitions that will be in the ring. We recommend that there be a minimum of 100 partitions per drive to insure even distribution across the drives. A good starting point might be to figure out the maximum number of drives the cluster will contain, and then multiply by 100, and then round up to the nearest power of two.

For example, imagine we are building a cluster that will have no more than 5,000 drives. That would mean that we would have a total number of 500,000 partitions, which is pretty close to 2^19, rounded up.

It is also a good idea to keep the number of partitions small (relatively). The more partitions there are, the more work that has to be done by the replicators and other backend jobs and the more memory the rings consume in process. The goal is to find a good balance between small rings and maximum cluster size.

The next step is to determine the number of replicas to store of the data. Currently it is recommended to use 3 (as this is the only value that has been tested). The higher the number, the more storage that is used but the less likely you are to lose data.

It is also important to determine how many zones the cluster should have. It is recommended to start with a minimum of 5 zones. You can start with fewer, but our testing has shown that having at least five zones is optimal when failures occur. We also recommend trying to configure the zones at as high a level as possible to create as much isolation as possible. Some example things to take into consideration can include physical location, power availability, and network connectivity. For example, in a small cluster you might decide to split the zones up by cabinet, with each cabinet having its own power and network connectivity. The zone concept is very abstract, so feel free to use it in whatever way best isolates your data from failure. Zones are referenced by number, beginning with 1.



Preparing The Ring

- Number of replicas:
  - 3 by default, but is configurable
- Number of zones:
  - Minimum of 5 recommended
  - Create as much isolation as possible

MIRANTIS

http://docs.openstack.org/trunk/openstack-object-storage/admin/content/preparing-the-ring.html

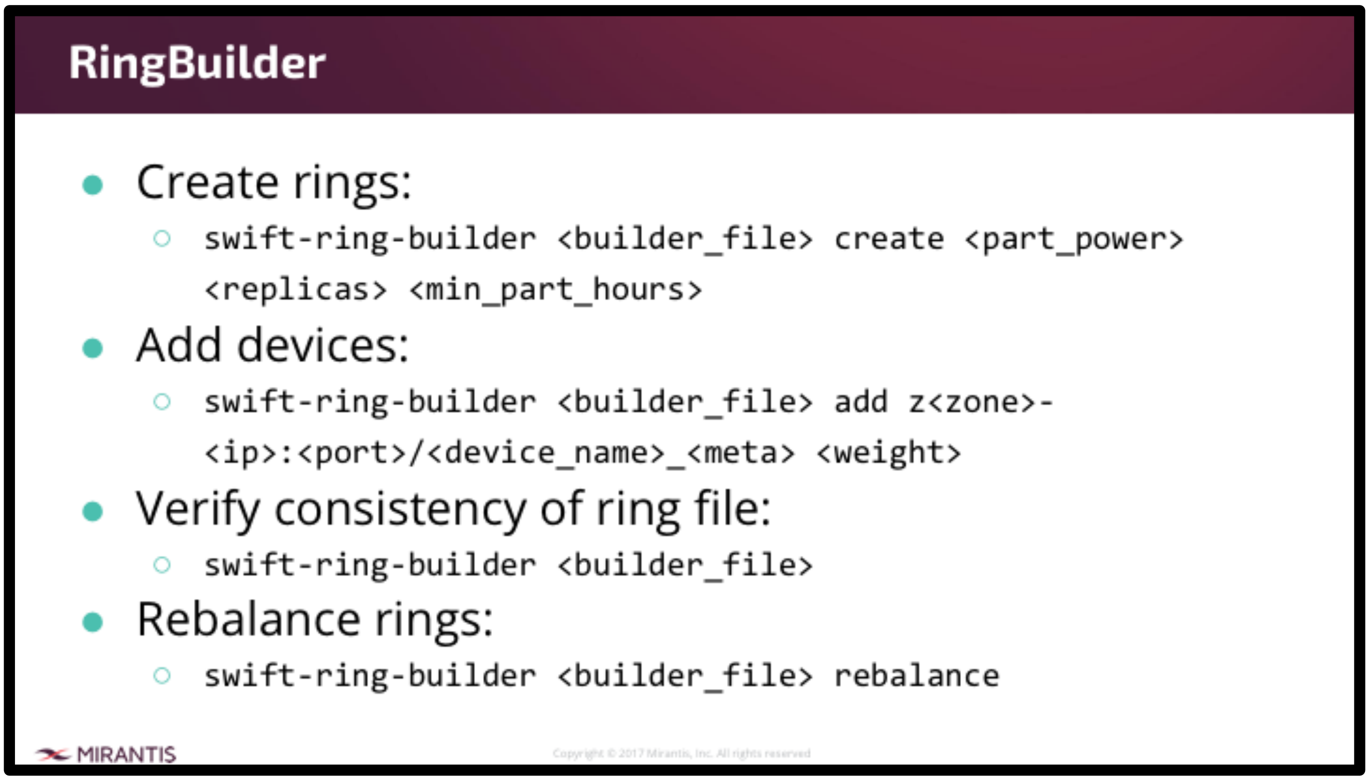
The first step is to determine the number of partitions that will be in the ring. We recommend that there be a minimum of 100 partitions per drive to insure even distribution across the drives. A good starting point might be to figure out the maximum number of drives the cluster will contain, and then multiply by 100, and then round up to the nearest power of two.

For example, imagine we are building a cluster that will have no more than 5,000 drives. That would mean that we would have a total number of 500,000 partitions, which is pretty close to 2^19, rounded up.

It is also a good idea to keep the number of partitions small (relatively). The more partitions there are, the more work that has to be done by the replicators and other backend jobs and the more memory the rings consume in process. The goal is to find a good balance between small rings and maximum cluster size.

The next step is to determine the number of replicas to store of the data. Currently it is recommended to use 3 (as this is the only value that has been tested). The higher the number, the more storage that is used but the less likely you are to lose data.

It is also important to determine how many zones the cluster should have. It is recommended to start with a minimum of 5 zones. You can start with fewer, but our testing has shown that having at least five zones is optimal when failures occur. We also recommend trying to configure the zones at as high a level as possible to create as much isolation as possible. Some example things to take into consideration can include physical location, power availability, and network connectivity. For example, in a small cluster you might decide to split the zones up by cabinet, with each cabinet having its own power and network connectivity. The zone concept is very abstract, so feel free to use it in whatever way best isolates your data from failure. Zones are referenced by number, beginning with 1.


RingBuilder

- Create rings:

  -
    ```
    swift-ring-builder <builder_file> create <part_power>
    <replicas> <min_part_hours>
    ```
- Add devices:

  -
    ```
    swift-ring-builder <builder_file> add z<zone>-
    <ip>:<port>/<device_name>_<meta> <weight>
    ```
- Verify consistency of ring file:

  -
    ```
    swift-ring-builder <builder_file>
    ```
- Rebalance rings:

  -
    ```
    swift-ring-builder <builder_file> rebalance
    ```

MIRANTIS

http://docs.openstack.org/trunk/openstack-object-storage/admin/content/preparing-the-ring.html

Rebalance:

This will distribute the partitions across the drives in the ring. It is important whenever making changes to the ring to make all the changes required before running rebalance. This will ensure that the ring stays as balanced as possible, and as few partitions are moved as possible.

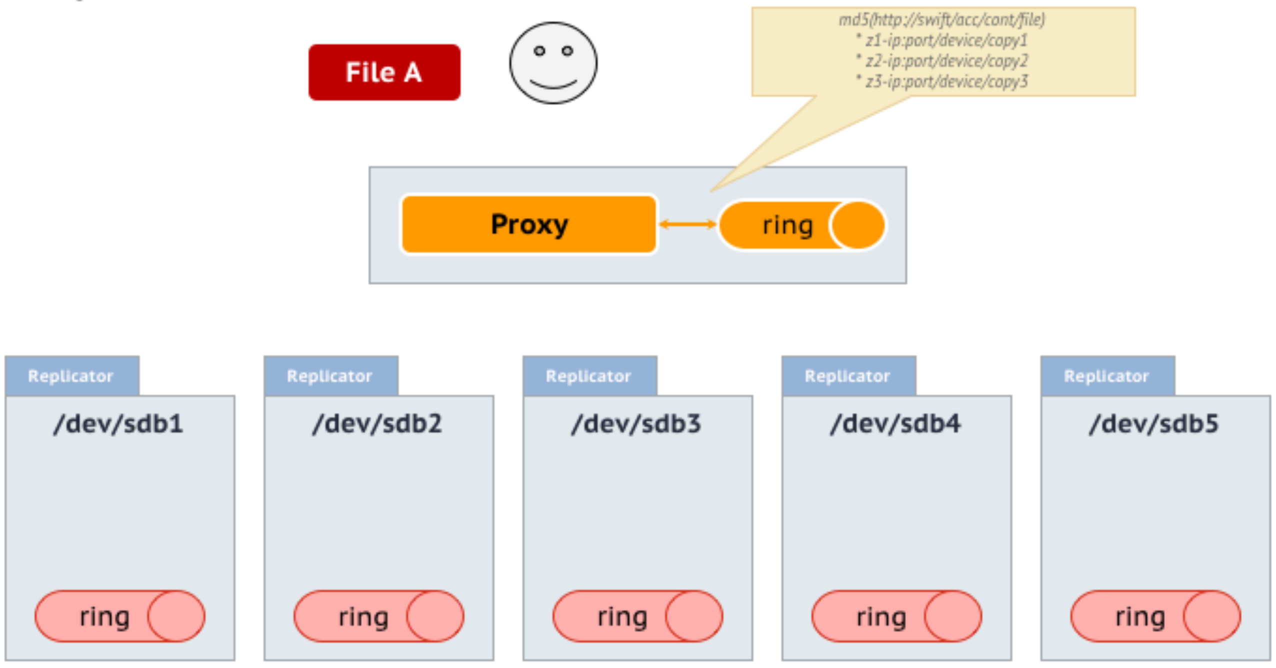
<min_part_hours> is the time in hours before a specific partition can be moved in succession (24 is a good value for this).

<min_part_hours> is number of hours to restrict moving a partition more than once

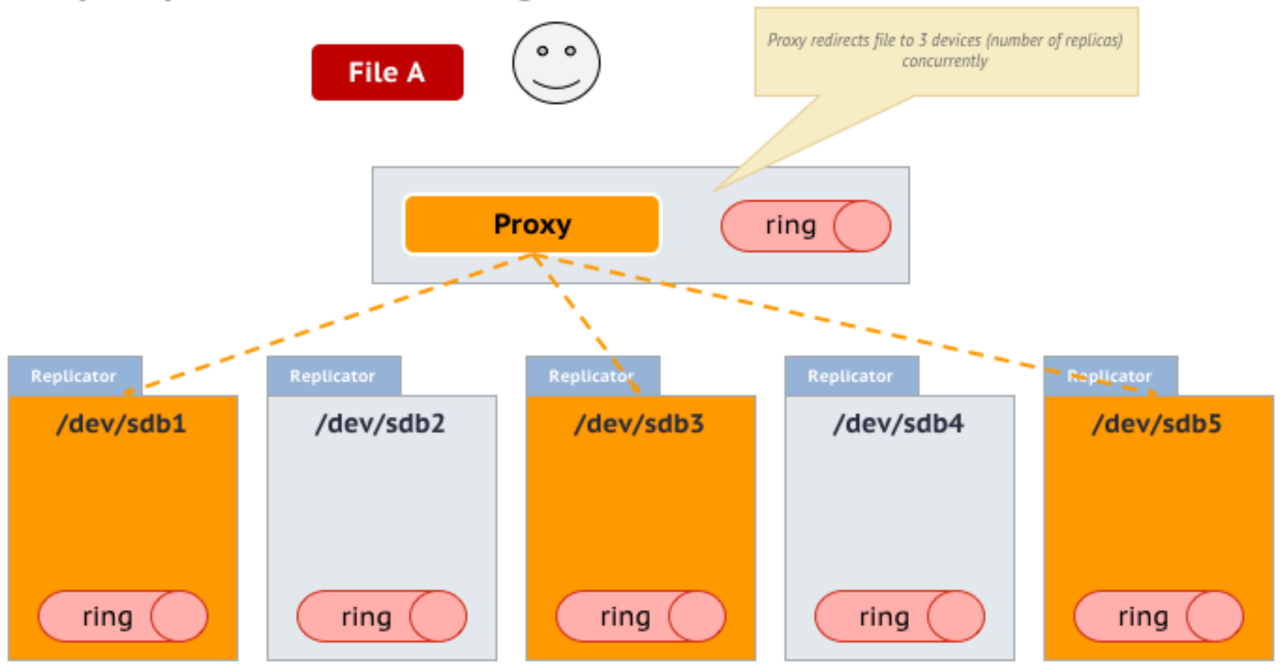
http://manpages.ubuntu.com/manpages/precise/man8/swift-ring-builder.8.html

Devices can have metadata associated with them

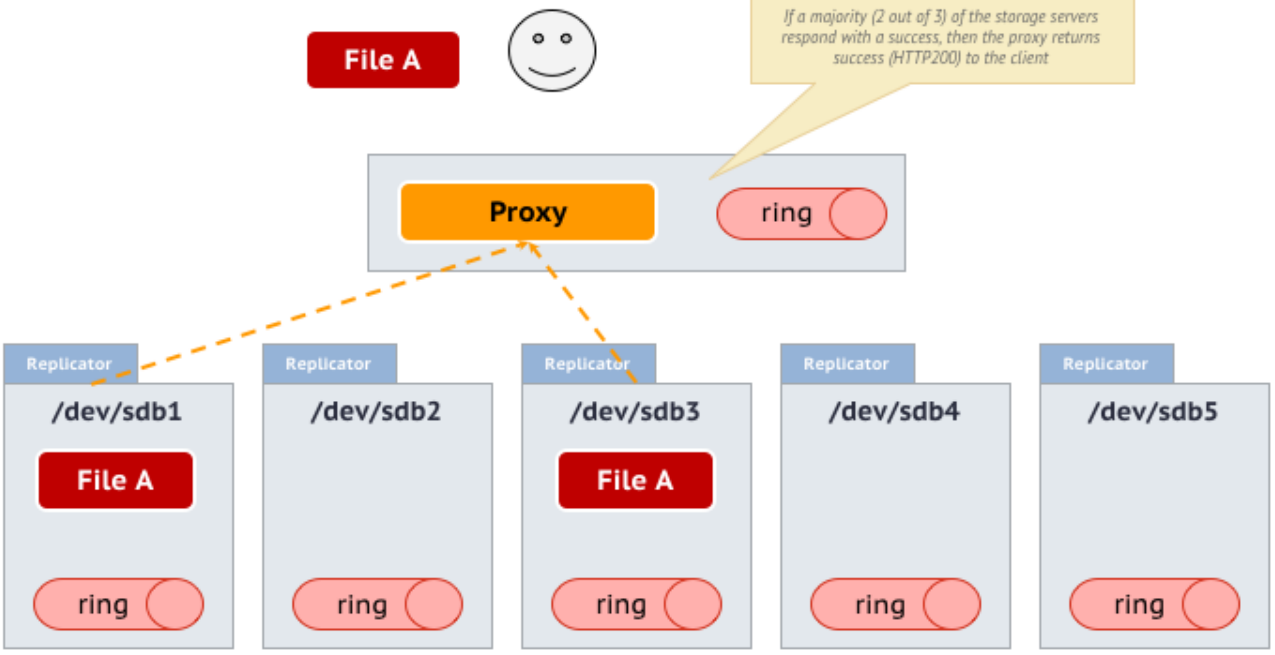
Step 2: Calculate Where the File Should Go



Step 3: Upload the File to Storage Servers

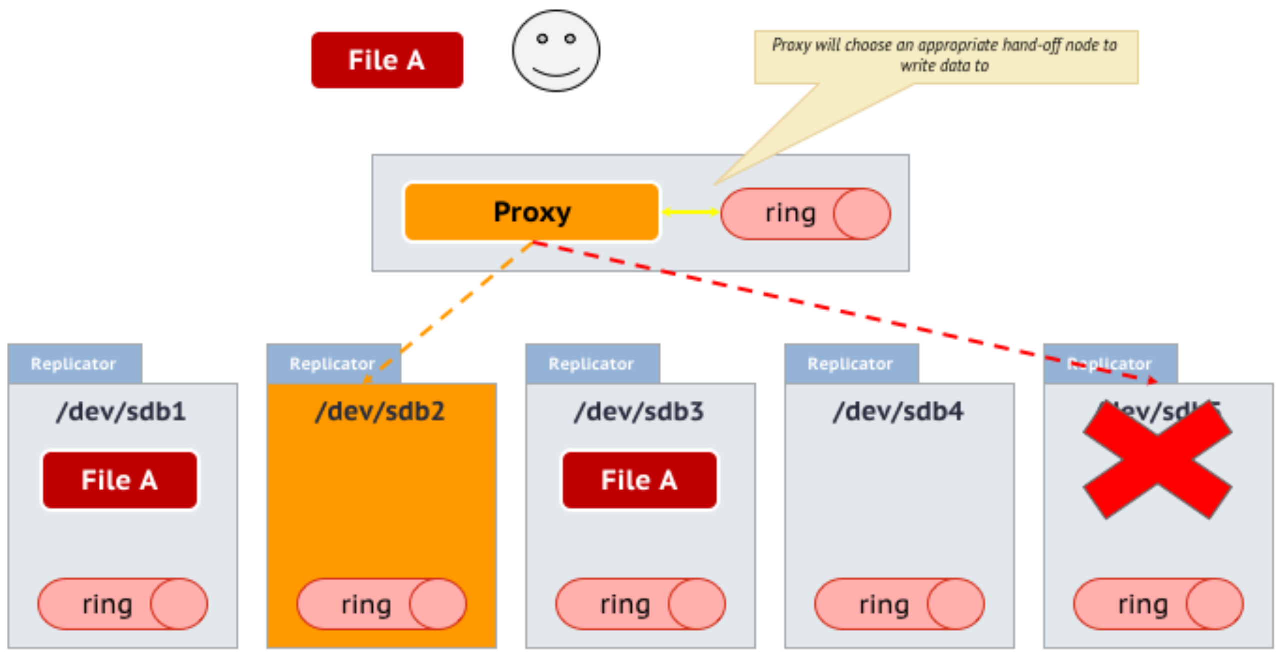


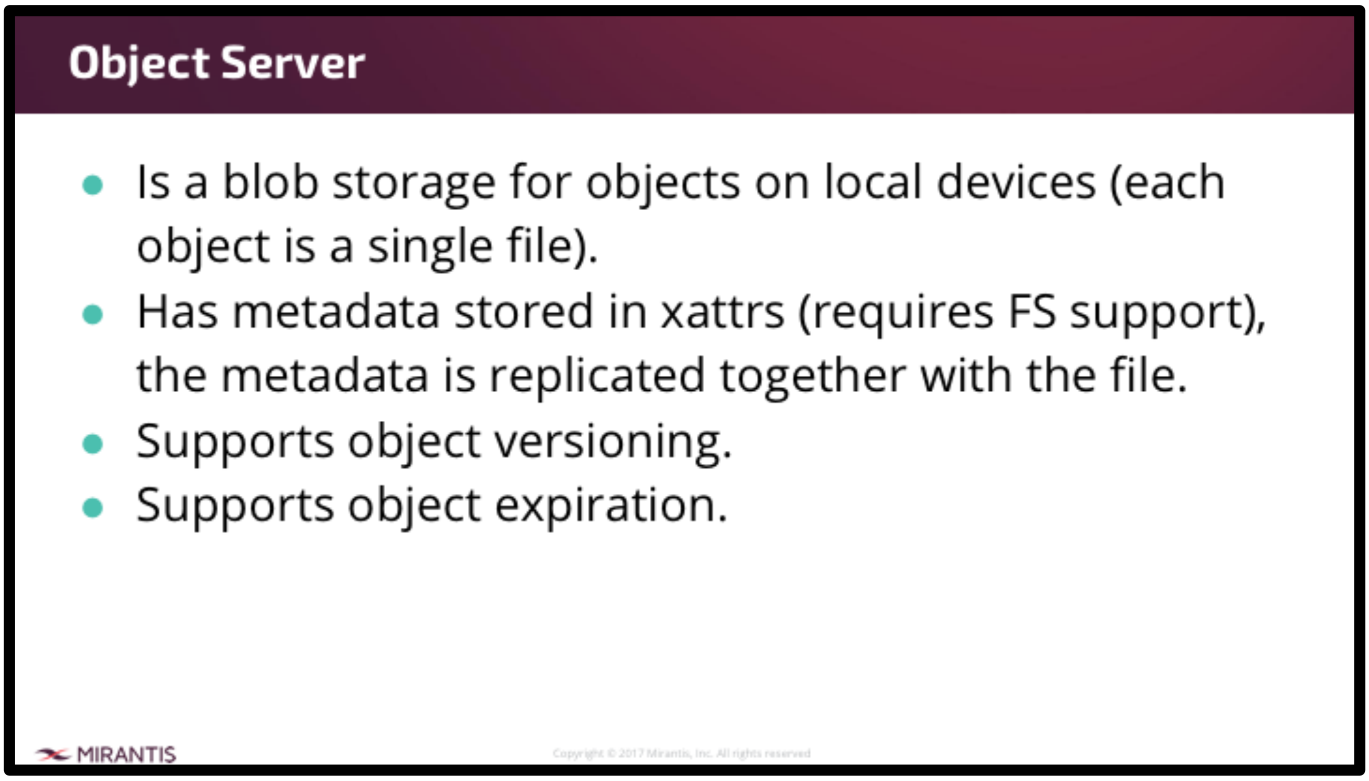
Step 4: Upload Successful



If a majority (2 out of 3) of the storage servers respond with a success, then the proxy returns success (HTTP200) to the client

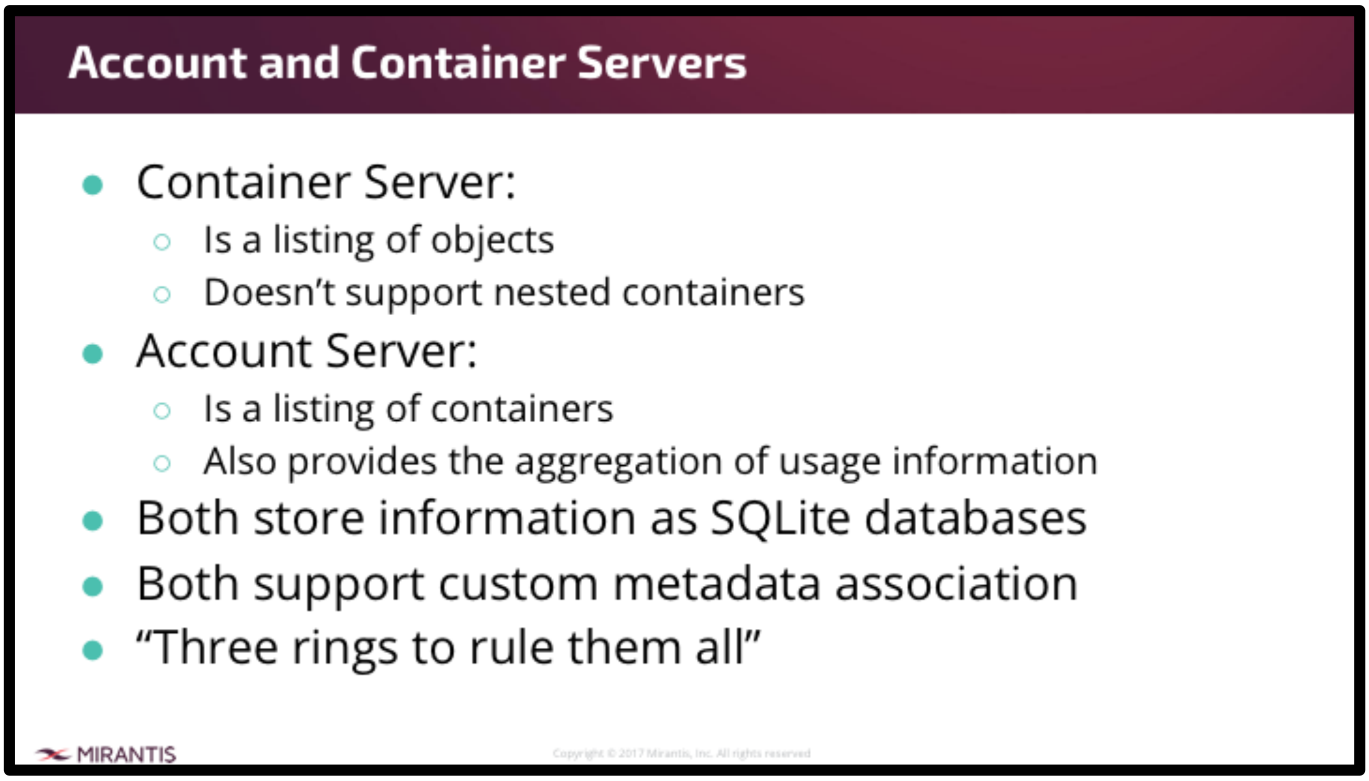
Step 4 (Alternative): One of the Primary Nodes is Down





Object Server

- Is a blob storage for objects on local devices (each object is a single file).
- Has metadata stored in xattrs (requires FS support), the metadata is replicated together with the file.
- Supports object versioning.
- Supports object expiration.

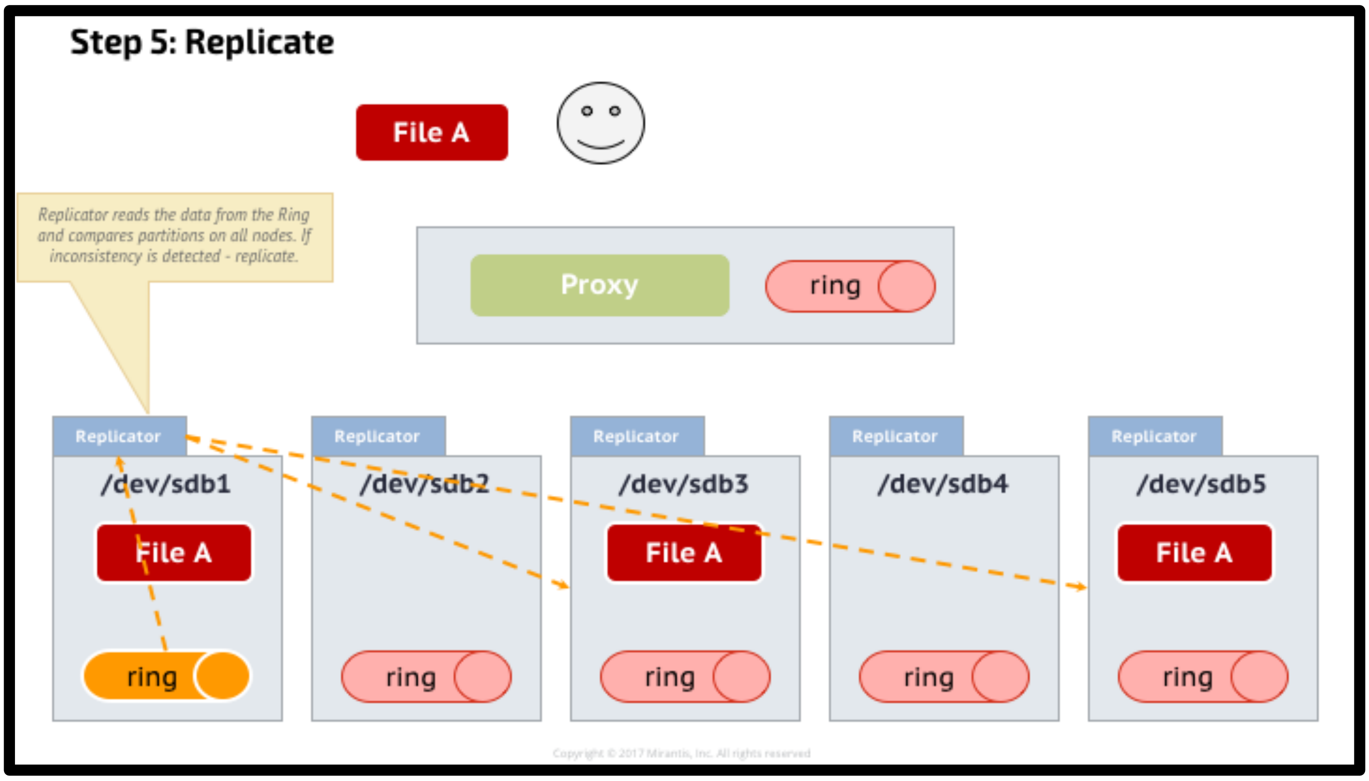


Account and Container Servers

- Container Server:
  - Is a listing of objects
  - Doesn’t support nested containers
- Account Server:
  - Is a listing of containers
  - Also provides the aggregation of usage information
- Both store information as SQLite databases
- Both support custom metadata association
- “Three rings to rule them all”

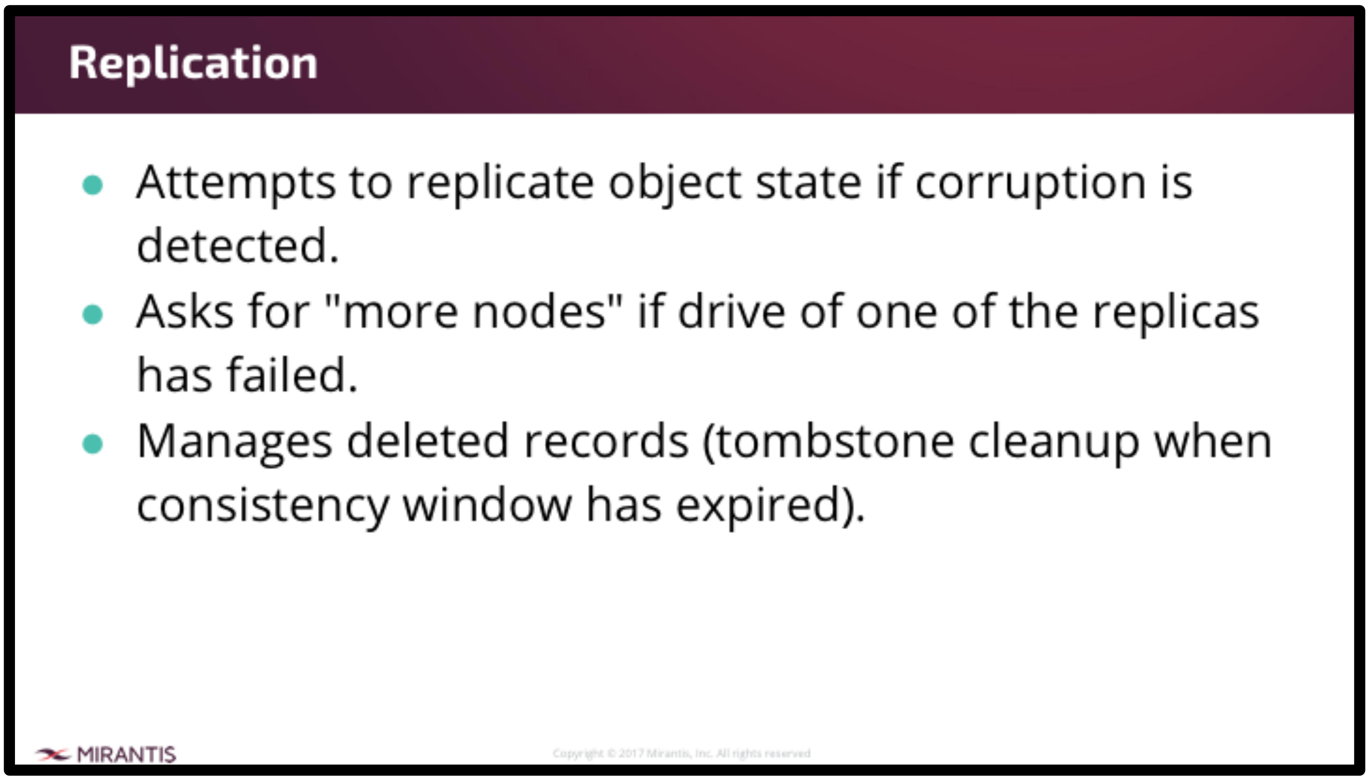
MIRANTIS

The request flow for creation of Container or an Account is exactly the same as for the file


Step 5: Replicate
File A
Replicator reads the data from the Ring and compares partitions on all nodes. If inconsistency is detected - replicate.
Proxy
ring
Replicator
/dev/sdb1
File A
ring
Replicator
/dev/sdb2
ring
Replicator
/dev/sdb3
File A
ring
Replicator
/dev/sdb4
ring
Replicator
/dev/sdb5
File A
ring
Copyright © 2017 Mirantis, Inc. All rights reserved

That's the role of the *swift-{container,account,object}-replicator* processes. These processes run on each node part of a zone and replicates their contents to nodes of the other zones.

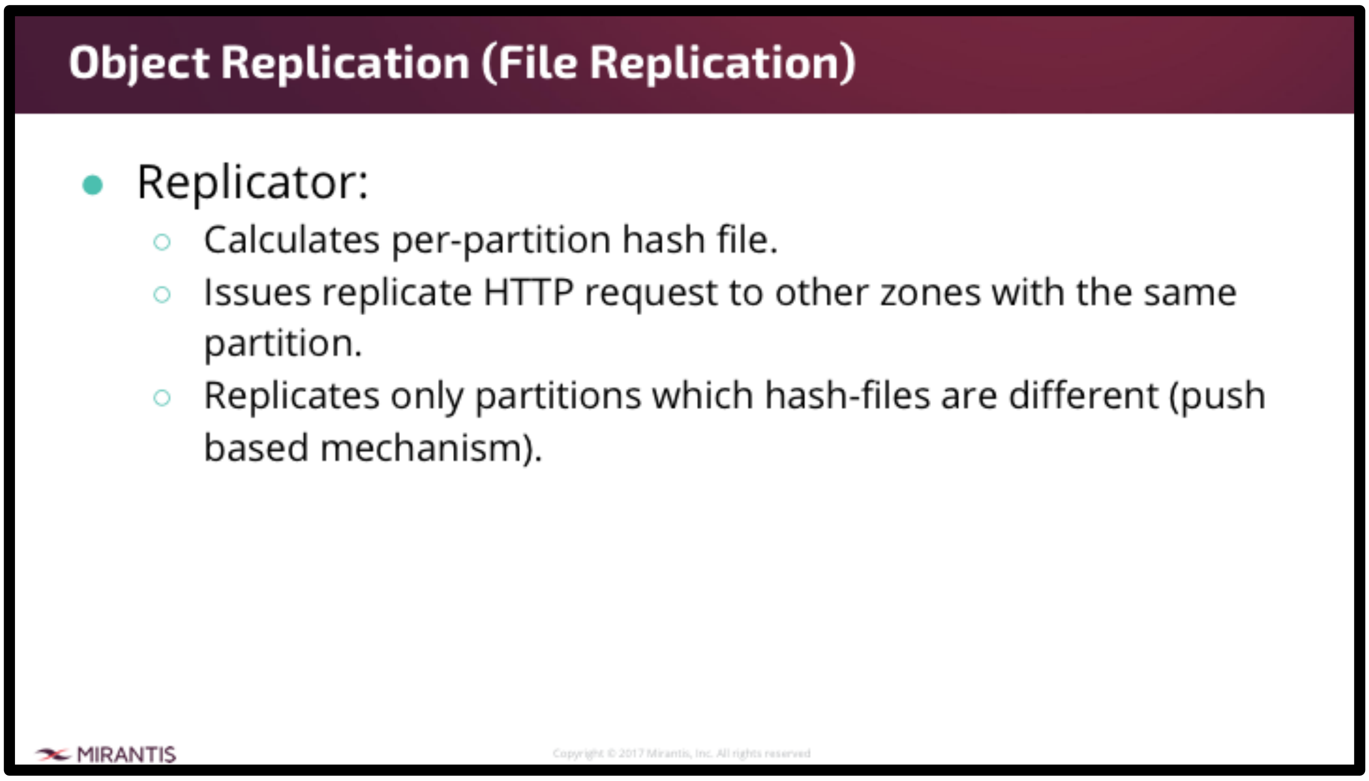
When they run, they walk through all the contents from all the partitions on the whole file system and for each partition, issue a special *REPLICATE* HTTP request to all the other zones responsible for that same partition. The other zone responds with information about the local state of the partition. That allows the replicator process to decide if the remote zone has an up-to-date version of the partition.



Replication

- Attempts to replicate object state if corruption is detected.
- Asks for "more nodes" if drive of one of the replicas has failed.
- Manages deleted records (tombstone cleanup when consistency window has expired).

MIRANTIS

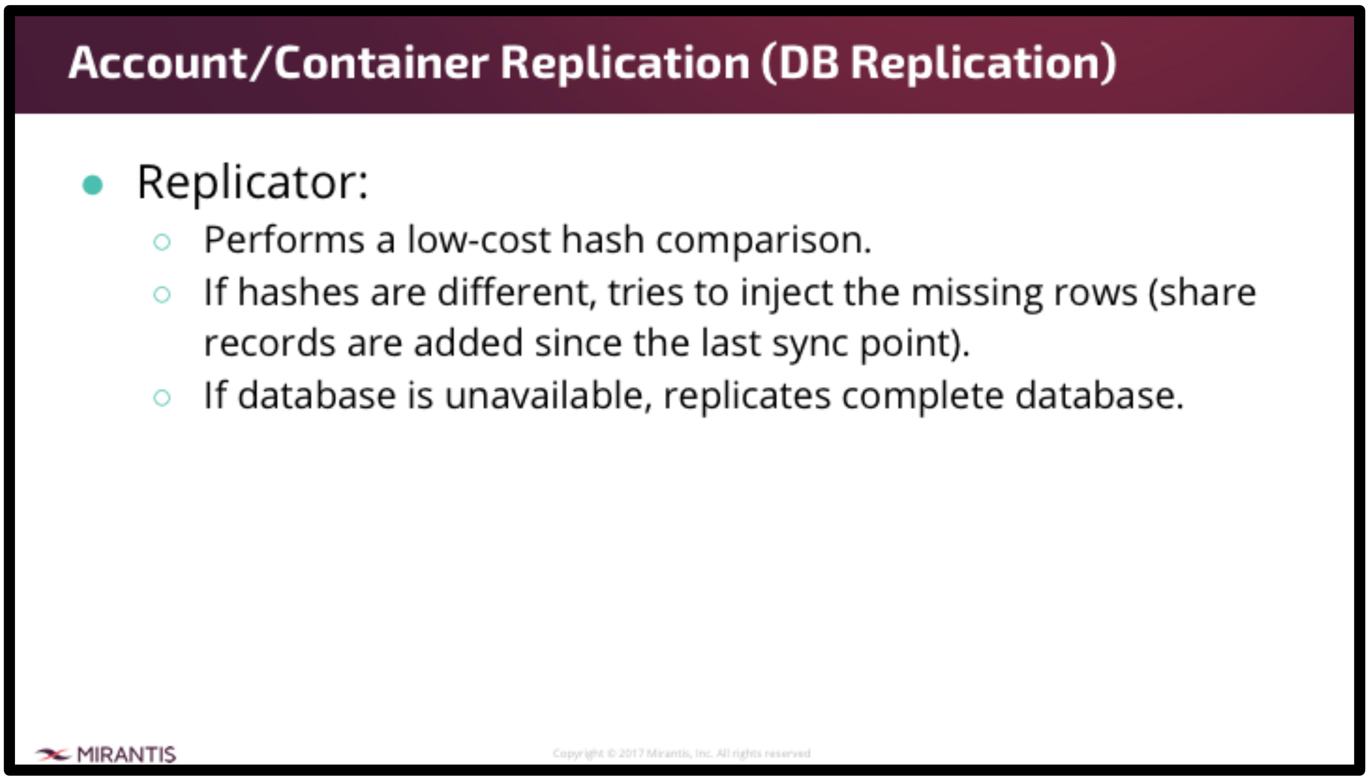


Object Replication (File Replication)

- Replicator:
  - Calculates per-partition hash file.
  - Issues replicate HTTP request to other zones with the same partition.
  - Replicates only partitions which hash-files are different (push based mechanism).

MIRANTIS

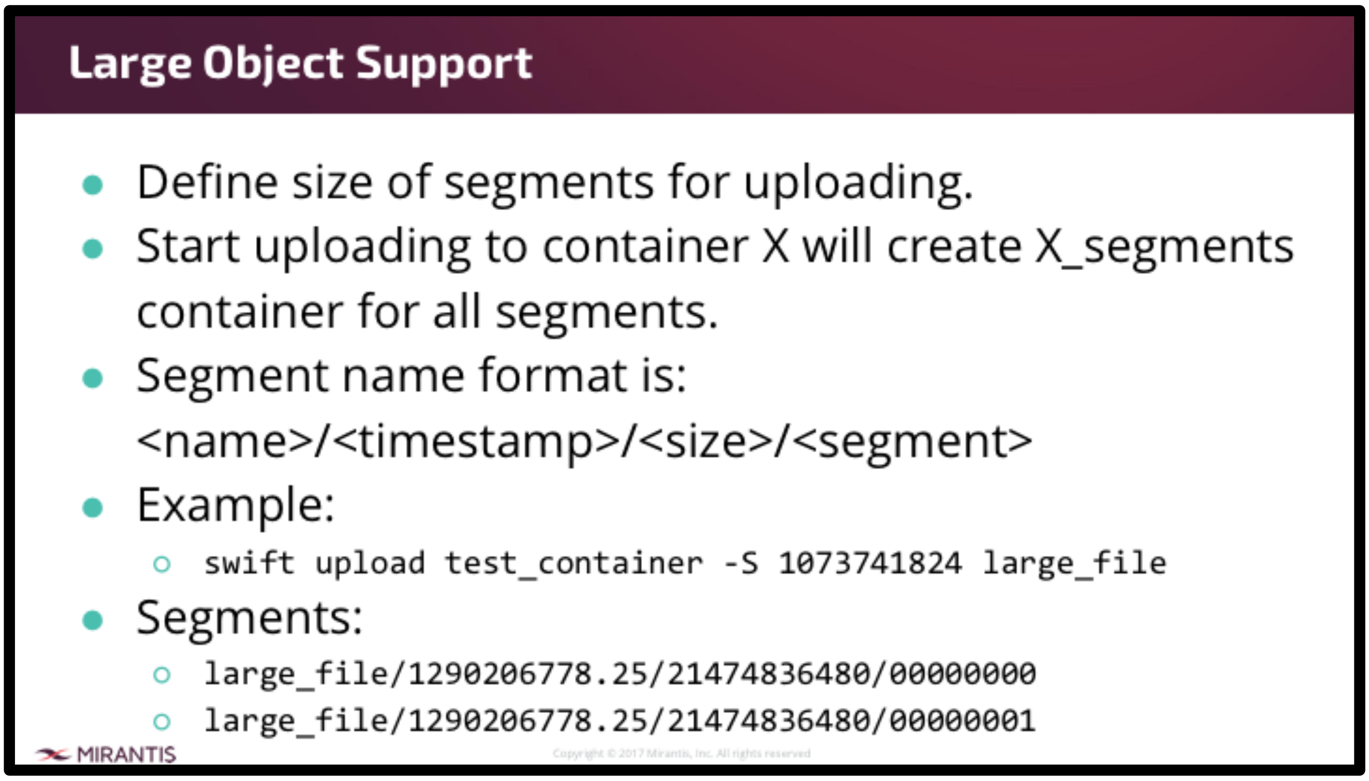
But if one of the zone is failing, you can't upload all your copies in all zones at the upload time. So you need a mechanism to be sure the failing zone will catch up to a correct state at some point.
That's the role of the *swift-{container,account,object}-replicator*processes. This processes are **running on each node part of a zone** and replicates their contents to nodes of the other zones.
When they run, they walk through all the contents from all the partitions on the whole file system and for each partition, issue a special *REPLICATE*HTTP request to all the other zones responsible for that same partition. The other zone responds with information about the local state of the partition. That allows the replicator process to decide if the remote zone has an up-to-date version of the partition.



Account/Container Replication (DB Replication)

- Replicator:
  - Performs a low-cost hash comparison.
  - If hashes are different, tries to inject the missing rows (share records are added since the last sync point).
  - If database is unavailable, replicates complete database.

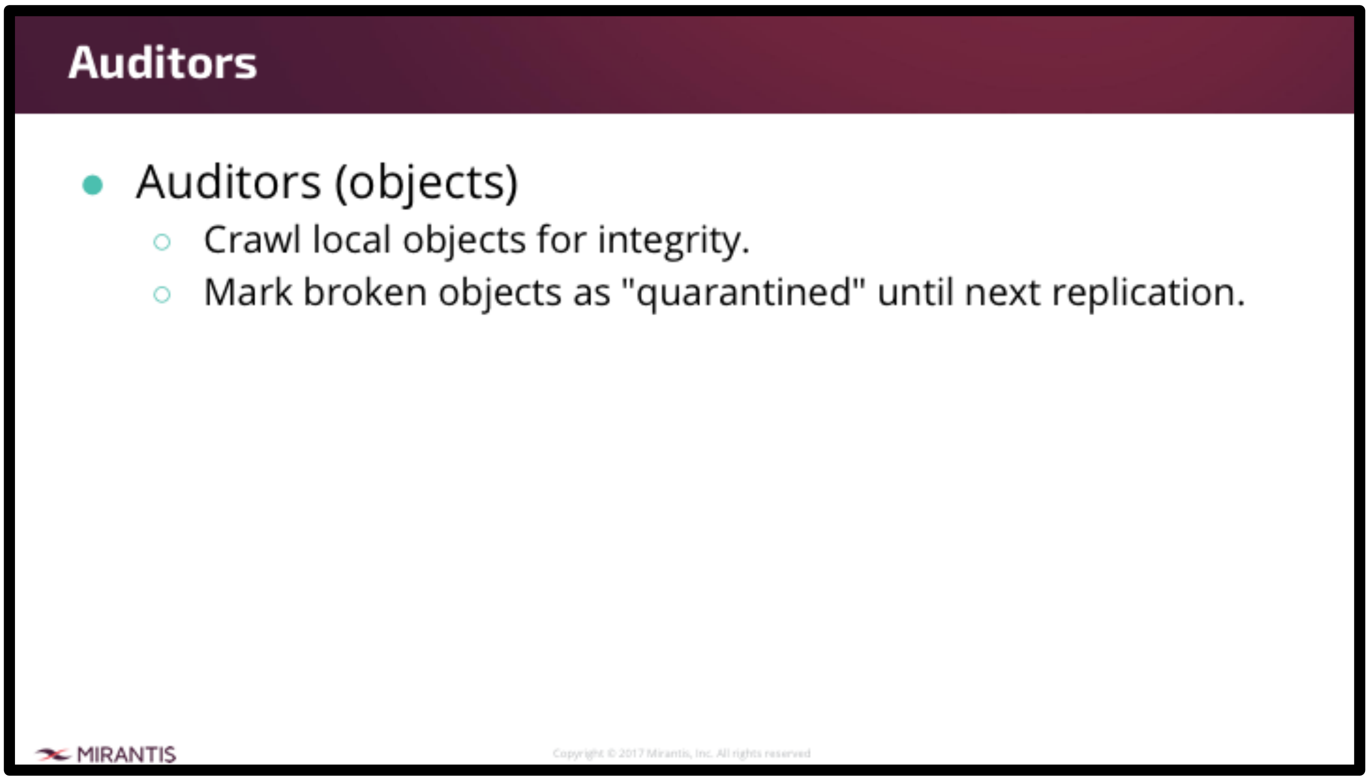
MIRANTIS



Large Object Support

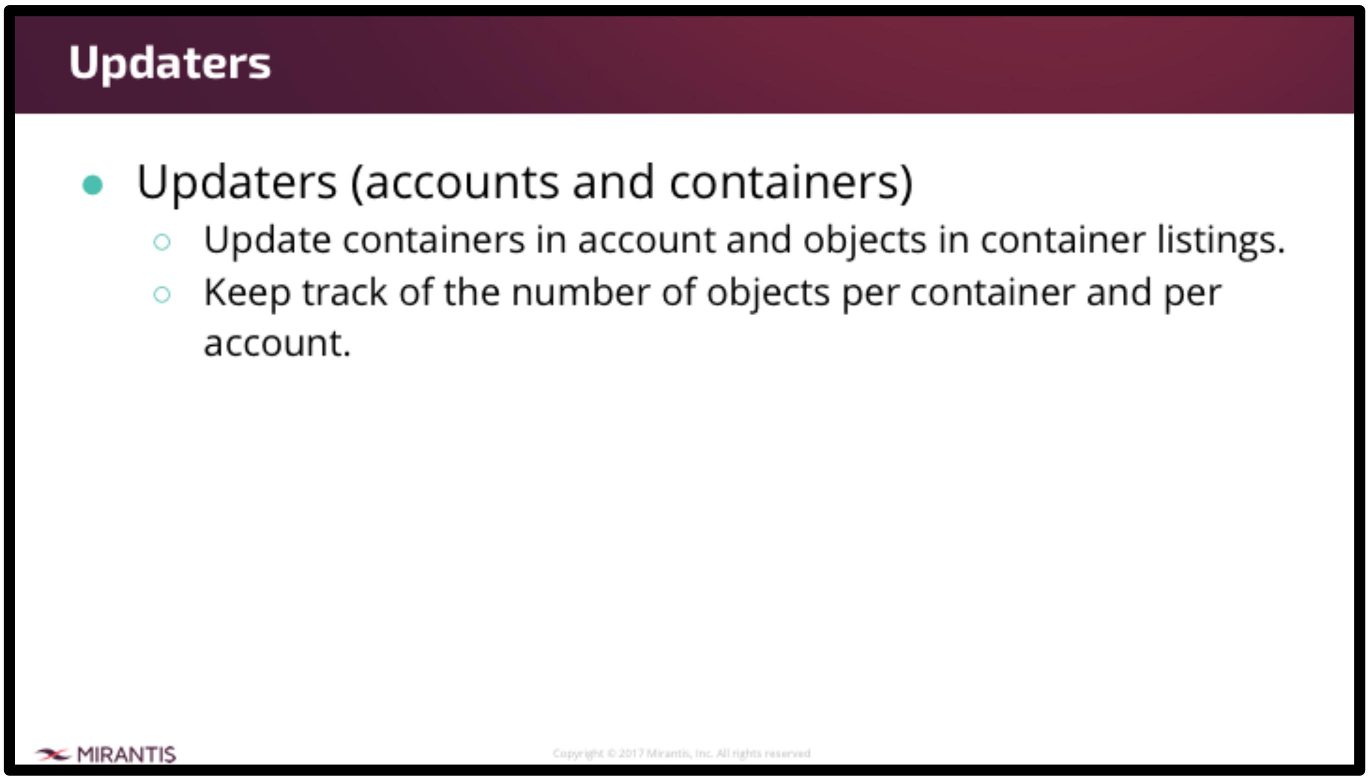
- Define size of segments for uploading.
- Start uploading to container X will create X_segments container for all segments.
- Segment name format is: <name>/<timestamp>/<size>/<segment>
- Example:
  - `swift upload test_container -S 1073741824 large_file`
- Segments:
  - `large_file/1290206778.25/21474836480/00000000`
  - `large_file/1290206778.25/21474836480/00000001`

MIRANTIS


Auditors

- Auditors (objects)
  - Crawl local objects for integrity.
  - Mark broken objects as "quarantined" until next replication.

MIRANTIS



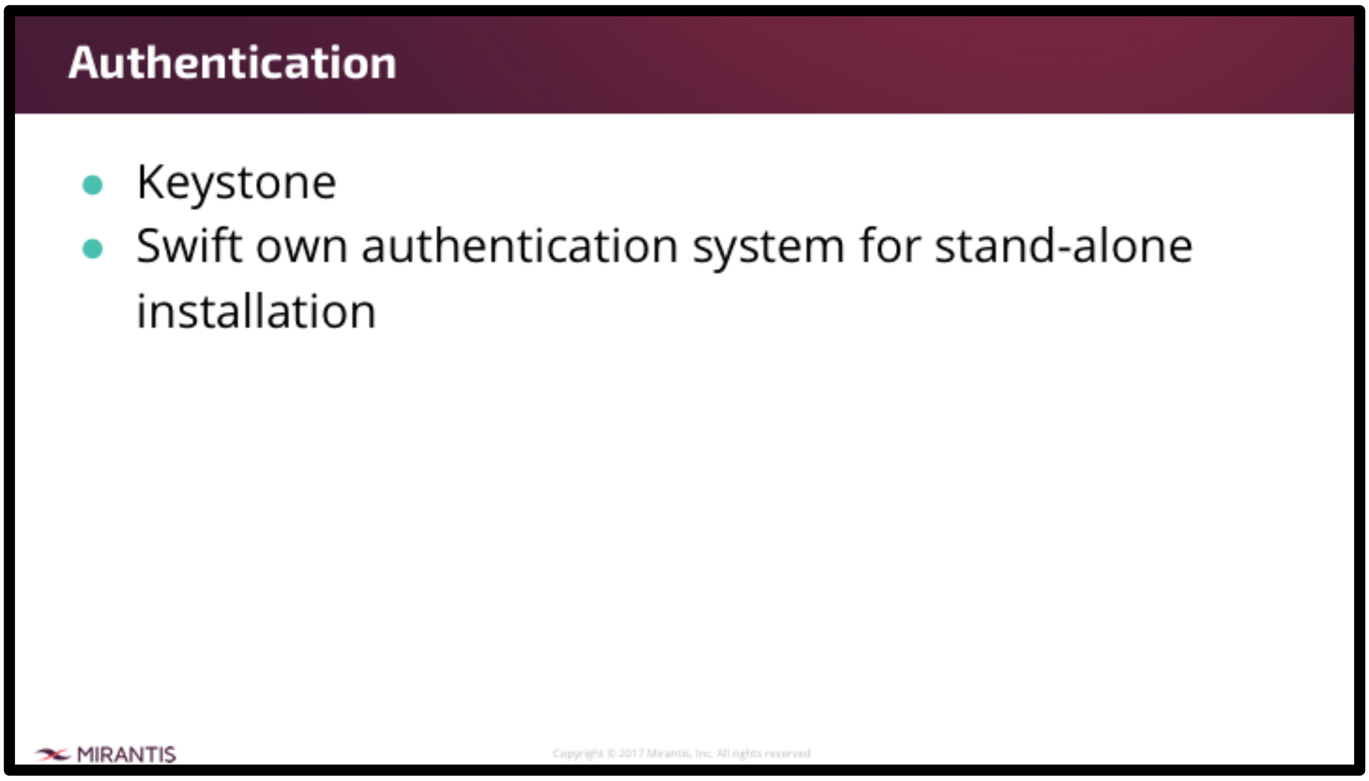
Updaters

- Updaters (accounts and containers)
  - Update containers in account and objects in container listings.
  - Keep track of the number of objects per container and per account.

MIRANTIS

The object updater is responsible for updating object information in container listings. It will check to see if there are any locally queued updates on the filesystem of each devices, what is also known as async pending file(s), walk each one and update the container listing.

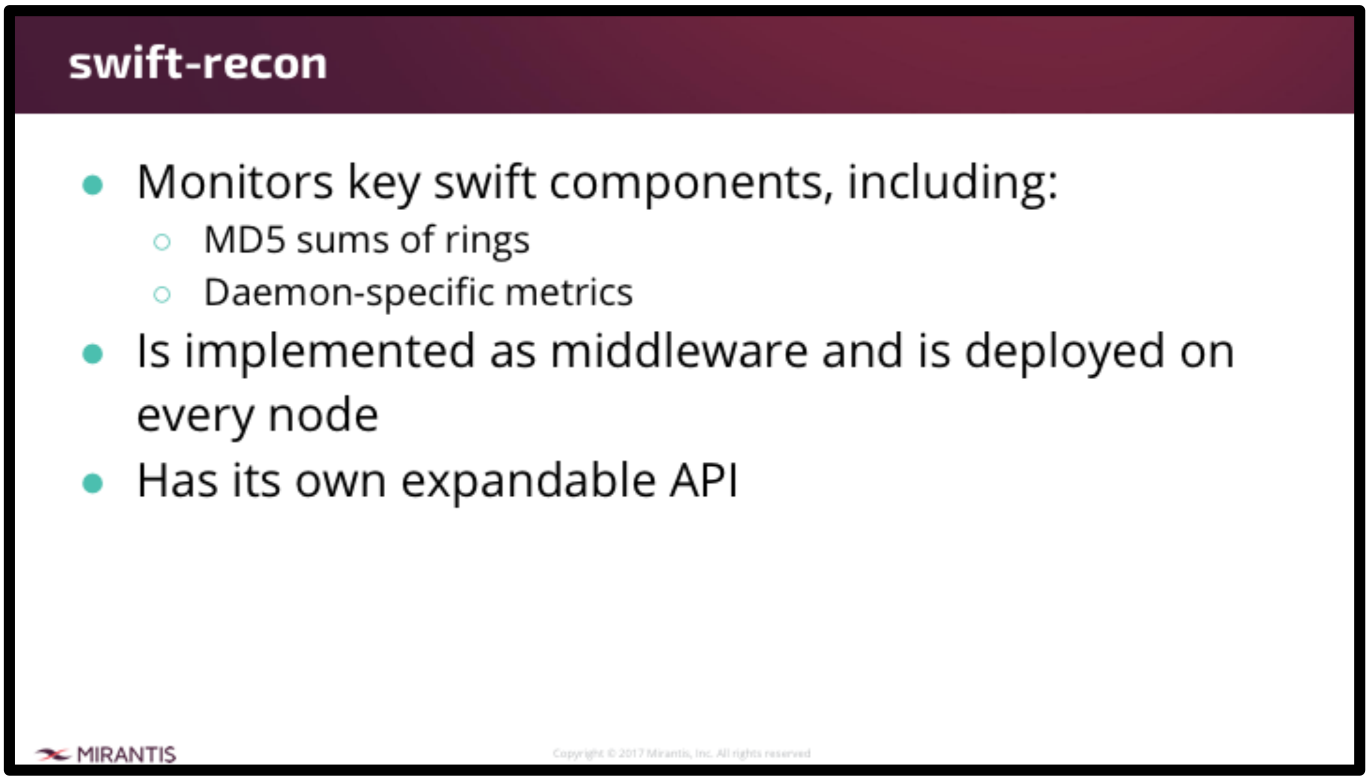
For example, suppose a container server is under load and a new object is put into the system. The object will be immediately available for reads as soon as the proxy server responds to the client with success. However, the object server has not been able to update the object listing in the container server. Therefore, the update would be queued locally for a later update. Container listings, therefore, may not immediately contain the object. This is where an eventual consistency window will most likely come in to play.



Authentication

- Keystone
- Swift own authentication system for stand-alone installation

MIRANTIS



swift-recon

- Monitors key swift components, including:
  - MD5 sums of rings
  - Daemon-specific metrics
- Is implemented as middleware and is deployed on every node
- Has its own expandable API

MIRANTIS

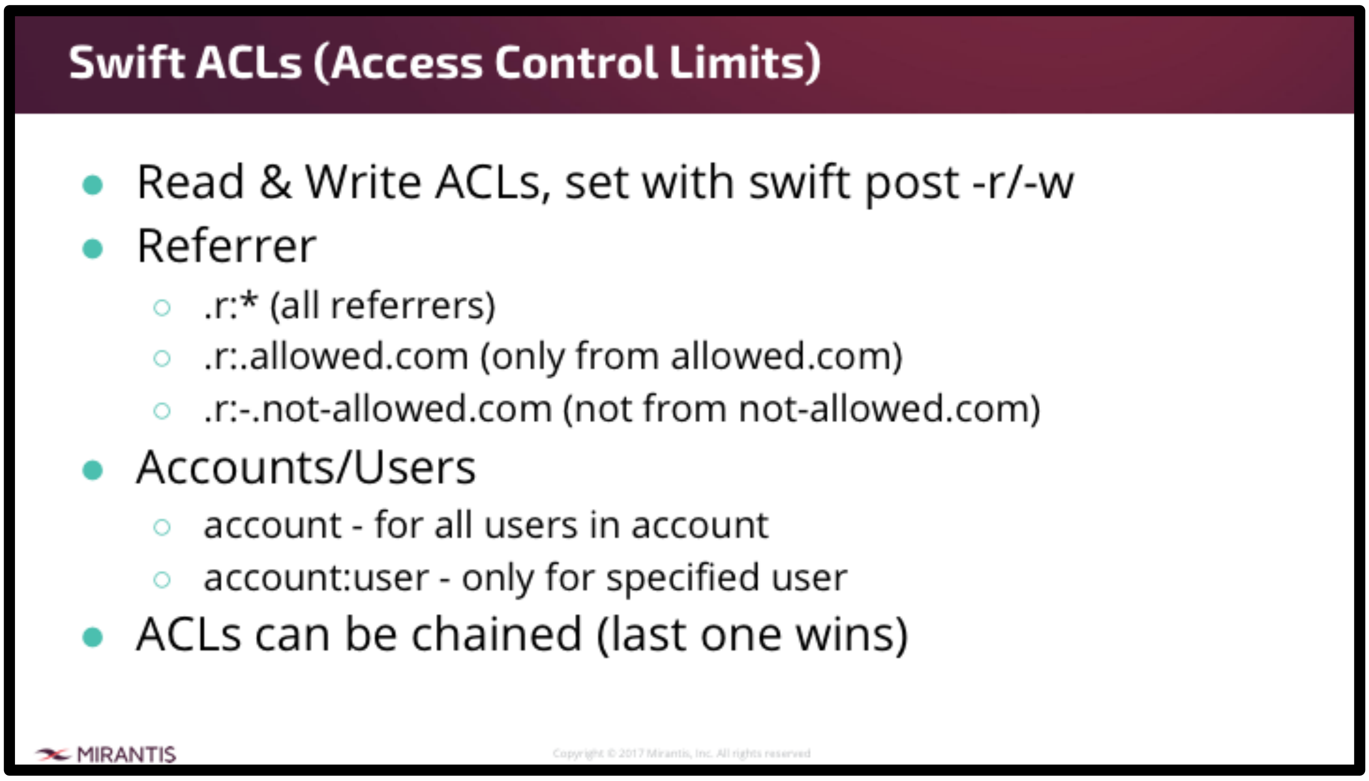
RLA – repair level analysis

The Swift Recon middleware provides general machine statistics, such as load average, socket statistics, `/proc/meminfo` contents, and so on, as well as Swift-specific metrics:

- The MD5 sum of each ring file.
- The most recent object replication time.
- Count of each type of quarantined file: Account, container, or object.
- Count of "async_pendings" (deferred container updates) on disk.

Swift Recon is middleware that is installed in the object servers pipeline and takes one required option: A local cache directory. To track `async_pendings`, you must set up an additional cron job for each object server. You access data by either sending HTTP requests directly to the object server or using the `swift-recon` command-line client.

There are some good Object Storage cluster statistics but the general server metrics overlap with existing server monitoring systems. To get the Swift-specific metrics into a monitoring system, they must be polled. Swift Recon essentially acts as a middleware metrics collector. The process that feeds metrics to your statistics system, such as `collectd` and `gmond`, probably already runs on the storage node. So, you can choose to either talk to Swift Recon or collect the metrics directly.



Swift ACLs (Access Control Limits)

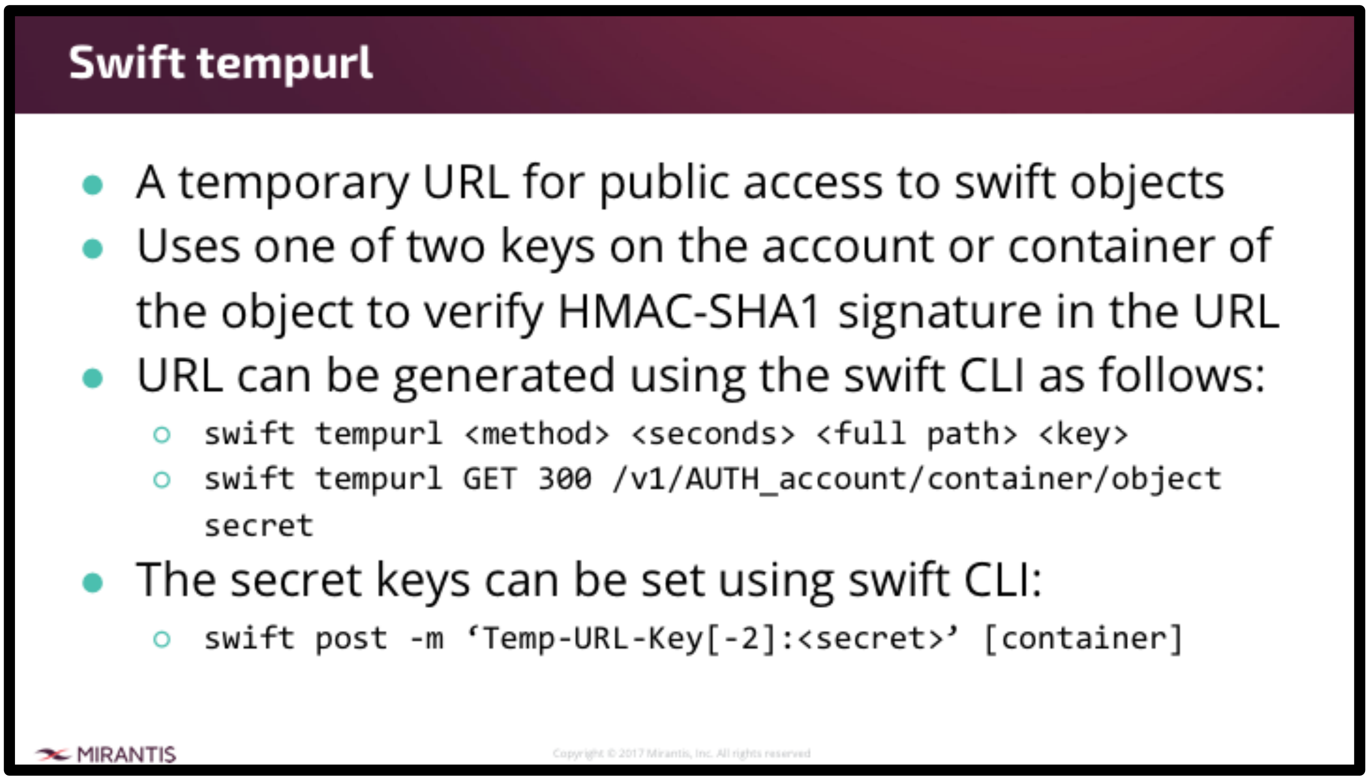
- Read & Write ACLs, set with swift post -r/-w
- Referrer
  - .r:* (all referrers)
  - .r:.allowed.com (only from allowed.com)
  - .r:-.not-allowed.com (not from not-allowed.com)
- Accounts/Users
  - account - for all users in account
  - account:user - only for specified user
- ACLs can be chained (last one wins)

MIRANTIS

Cross-domain Policy File¶
A cross-domain policy file allows web pages hosted elsewhere to use client side technologies such as Flash, Java and Silverlight to interact with the Swift API.
See http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html for a description of the purpose and structure of the cross-domain policy file. The cross-domain policy file is installed in the root of a web server (i.e., the path is /crossdomain.xml).
The crossdomain middleware responds to a path of /crossdomain.xml with an XML document such as:

```
<?xml version="1.0"?> <!DOCTYPE cross-domain-policy SYSTEM
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd" > <cross-domain-policy>
<allow-access-from domain="*" secure="false" /> </cross-domain-policy>
```

You should use a policy appropriate to your site. The examples and the default policy are provided to indicate how to syntactically construct a cross domain policy file – they are not recommendations.

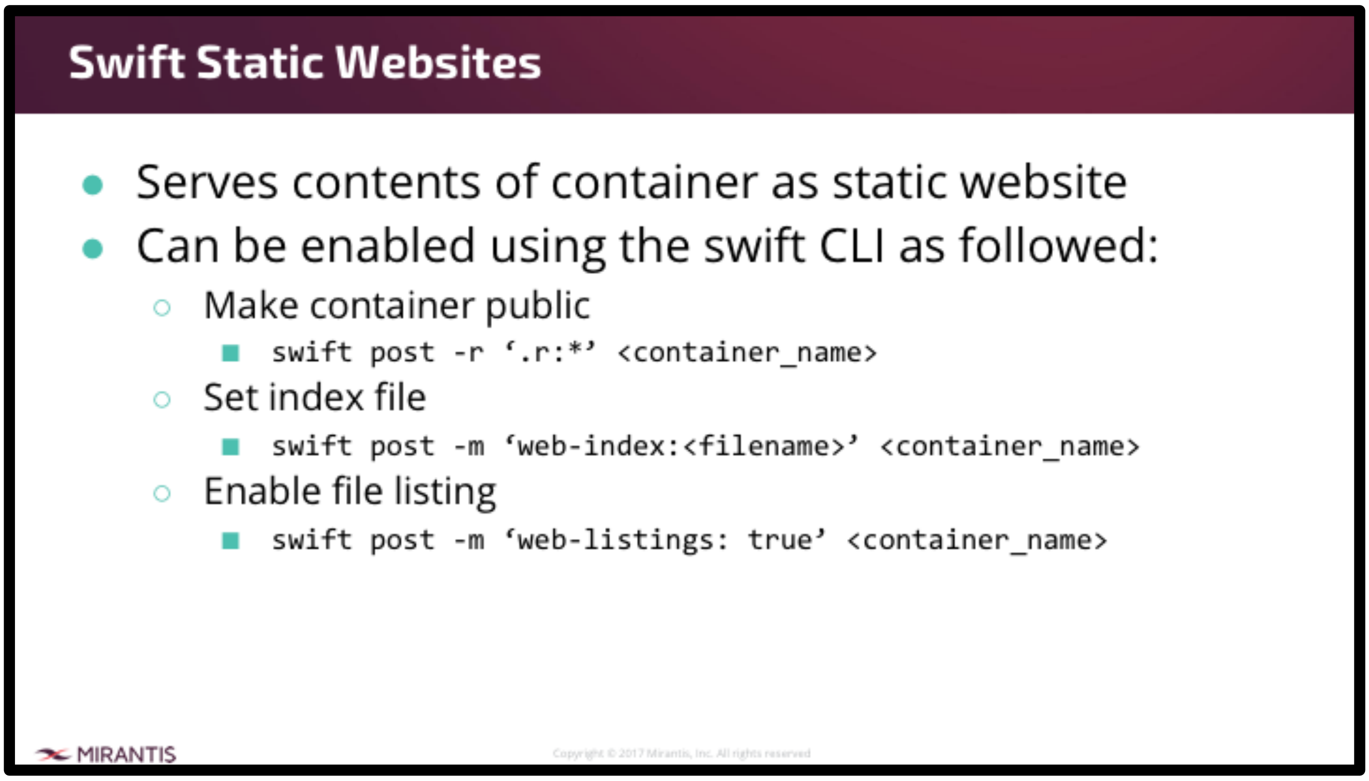


Swift tempurl

- A temporary URL for public access to swift objects
- Uses one of two keys on the account or container of the object to verify HMAC-SHA1 signature in the URL
- URL can be generated using the swift CLI as follows:
  - `swift tempurl <method> <seconds> <full path> <key>`
  - `swift tempurl GET 300 /v1/AUTH_account/container/object secret`
- The secret keys can be set using swift CLI:
  - `swift post -m ‘Temp-URL-Key[-2]:<secret>’ [container]`

MIRANTIS

http://docs.openstack.org/developer/swift/api/temporary_url_middleware.html
http://blog.fsquat.net/?p=16



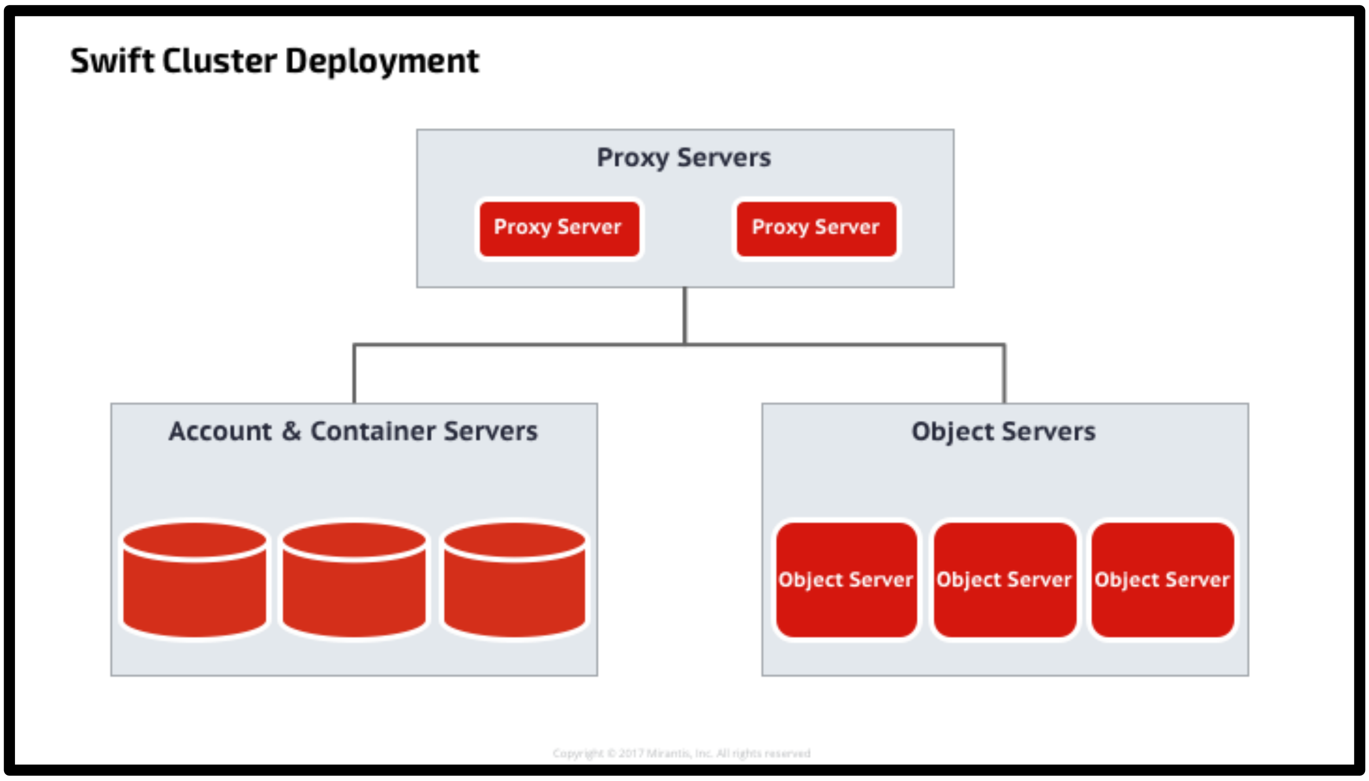
Swift Static Websites

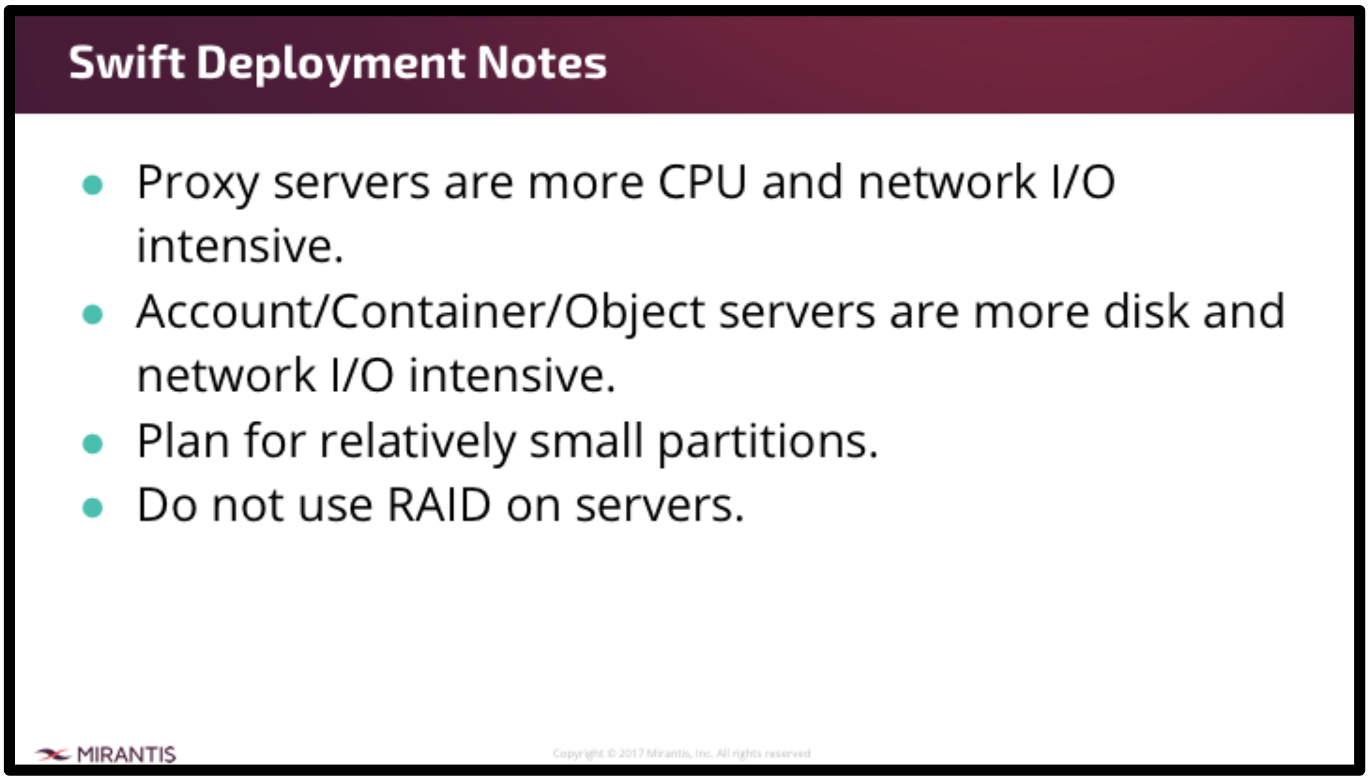
- Serves contents of container as static website
- Can be enabled using the swift CLI as followed:
  - Make container public
    - `swift post -r '.r:*' <container_name>`
  - Set index file
    - `swift post -m 'web-index:<filename>' <container_name>`
  - Enable file listing
    - `swift post -m 'web-listings: true' <container_name>`

MIRANTIS

http://docs.openstack.org/user-guide/cli_swift_static_website.html

Pretty nifty feature that's been in Swift for quite some time, allows users to place their static website content (html, css, images, etc.) into a container in swift, and just tell swift to serve the contents out as if it's a regular website. URL of the site ends up looking like http://<swift>:8080/account/container. Pseudo nested containers are supported as well to provide multiple levels of hierarchy for web content.

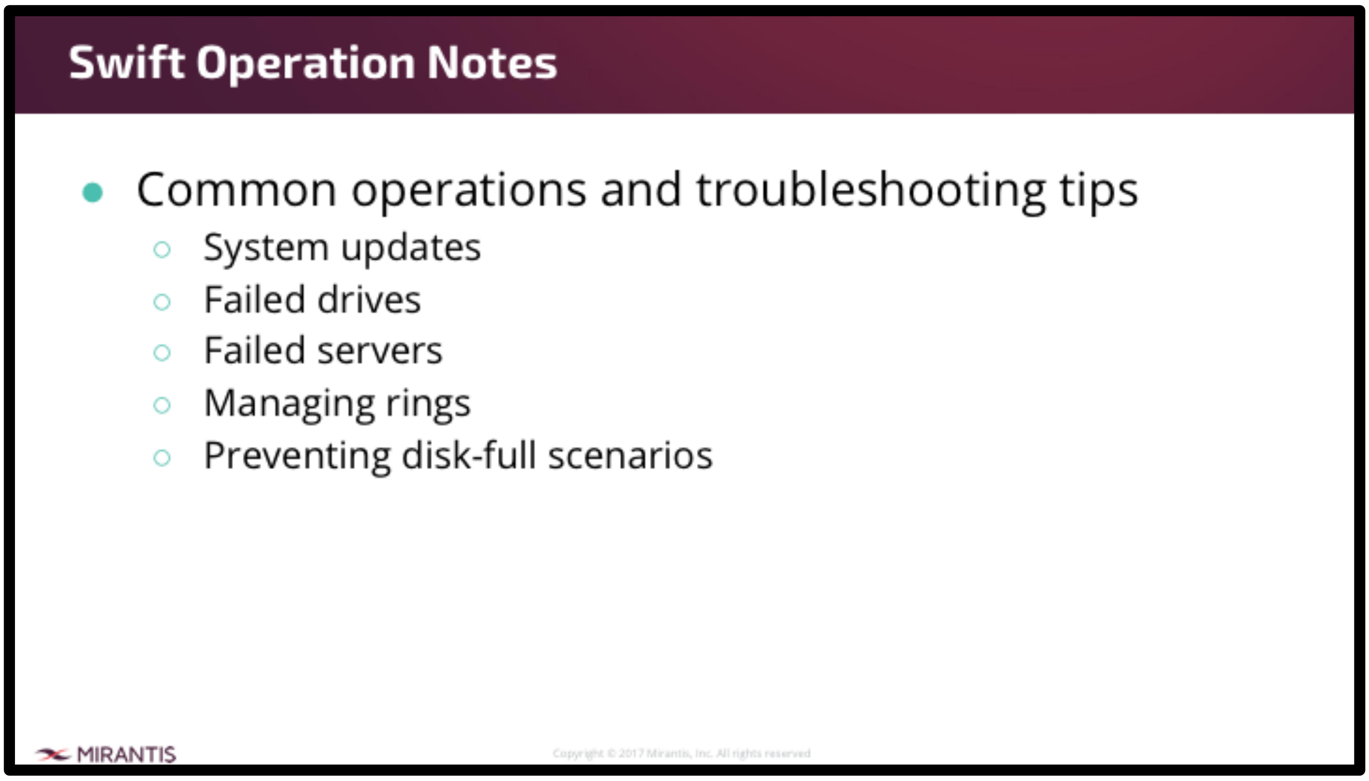

Swift Cluster Deployment
Proxy Servers
Proxy Server
Proxy Server
Account & Container Servers
Object Servers
Object Server
Object Server
Object Server
Copyright © 2017 Mirantis, Inc. All rights reserved


Swift Deployment Notes

- Proxy servers are more CPU and network I/O intensive.
- Account/Container/Object servers are more disk and network I/O intensive.
- Plan for relatively small partitions.
- Do not use RAID on servers.

MIRANTIS

Swift has support for using dedicated network for replication traffic.

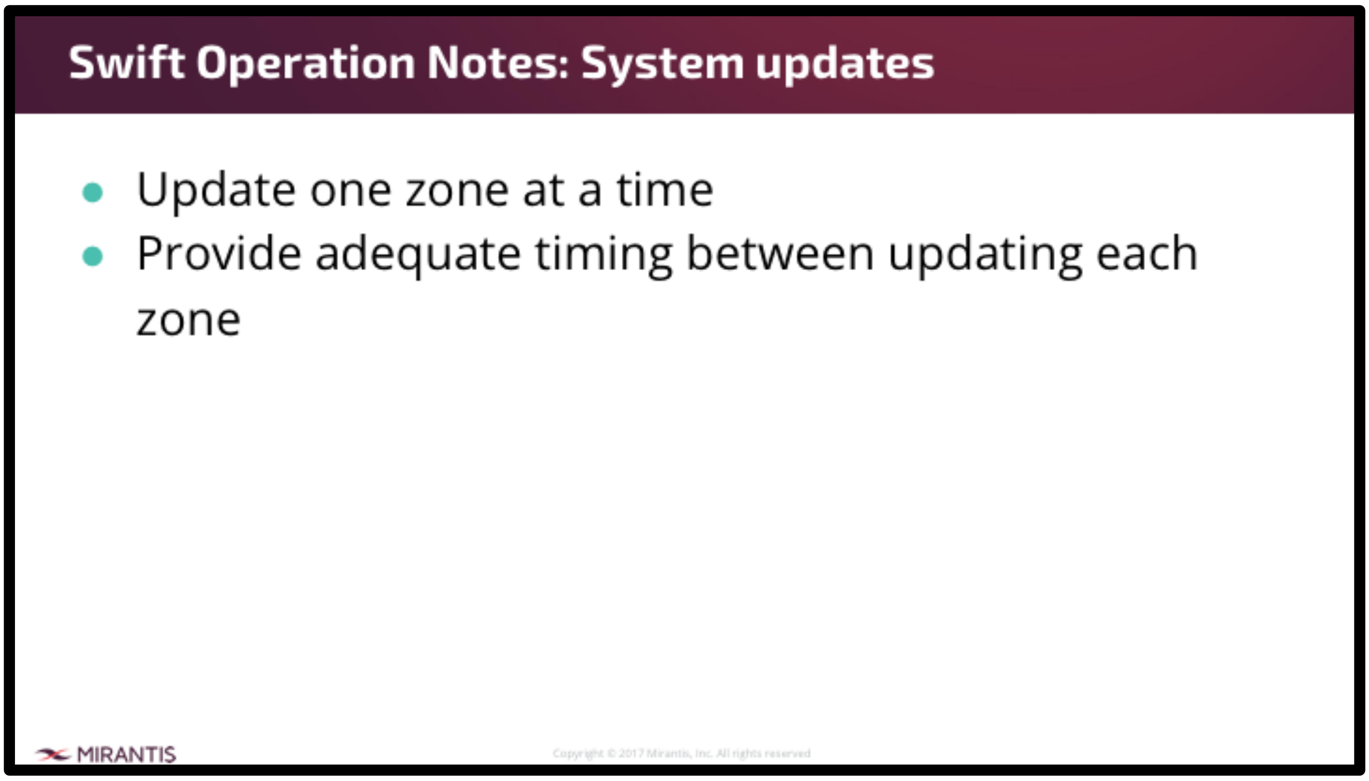


Swift Operation Notes

- Common operations and troubleshooting tips
  - System updates
  - Failed drives
  - Failed servers
  - Managing rings
  - Preventing disk-full scenarios

MIRANTIS

Source:
https://docs.openstack.org/swift/latest/admin_guide.html

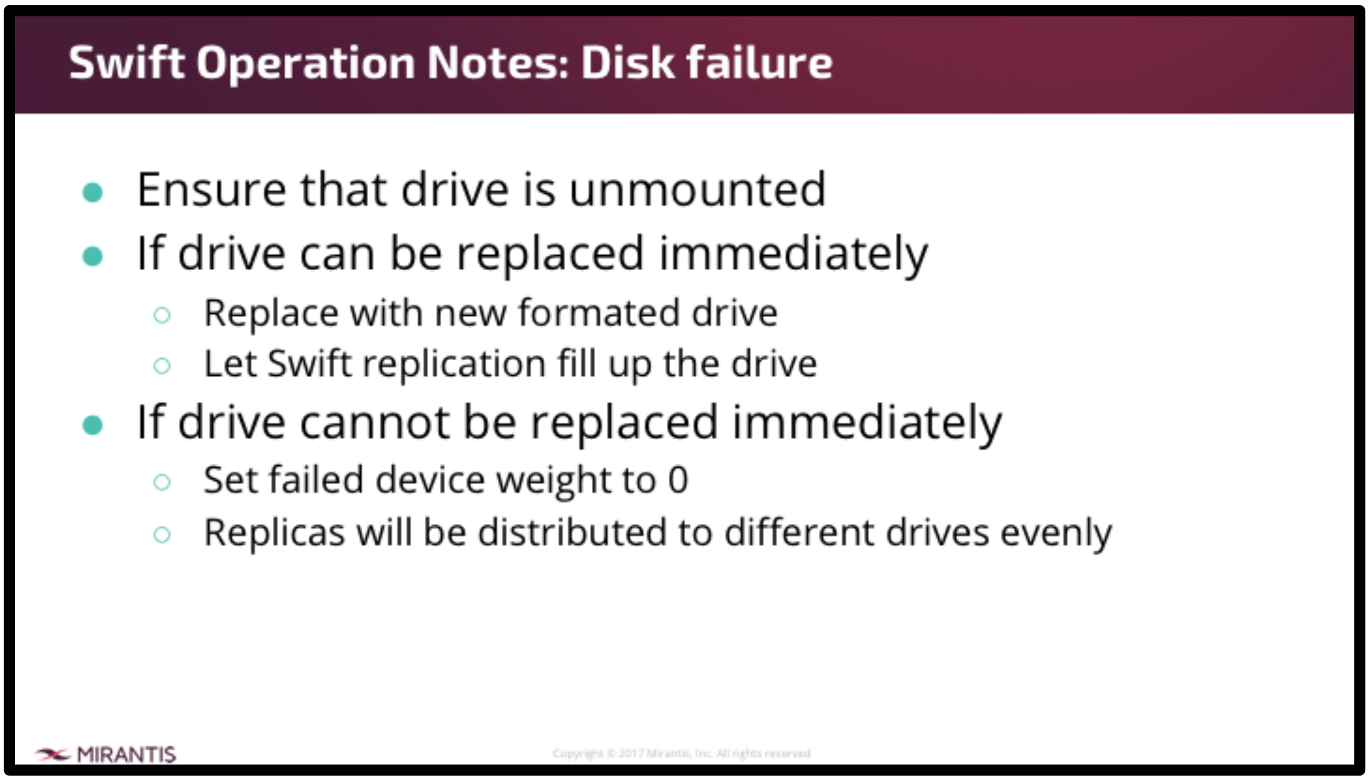


Swift Operation Notes: System updates

- Update one zone at a time
- Provide adequate timing between updating each zone

MIRANTIS

Source:
https://docs.openstack.org/swift/latest/admin_guide.html#handling-system-updates

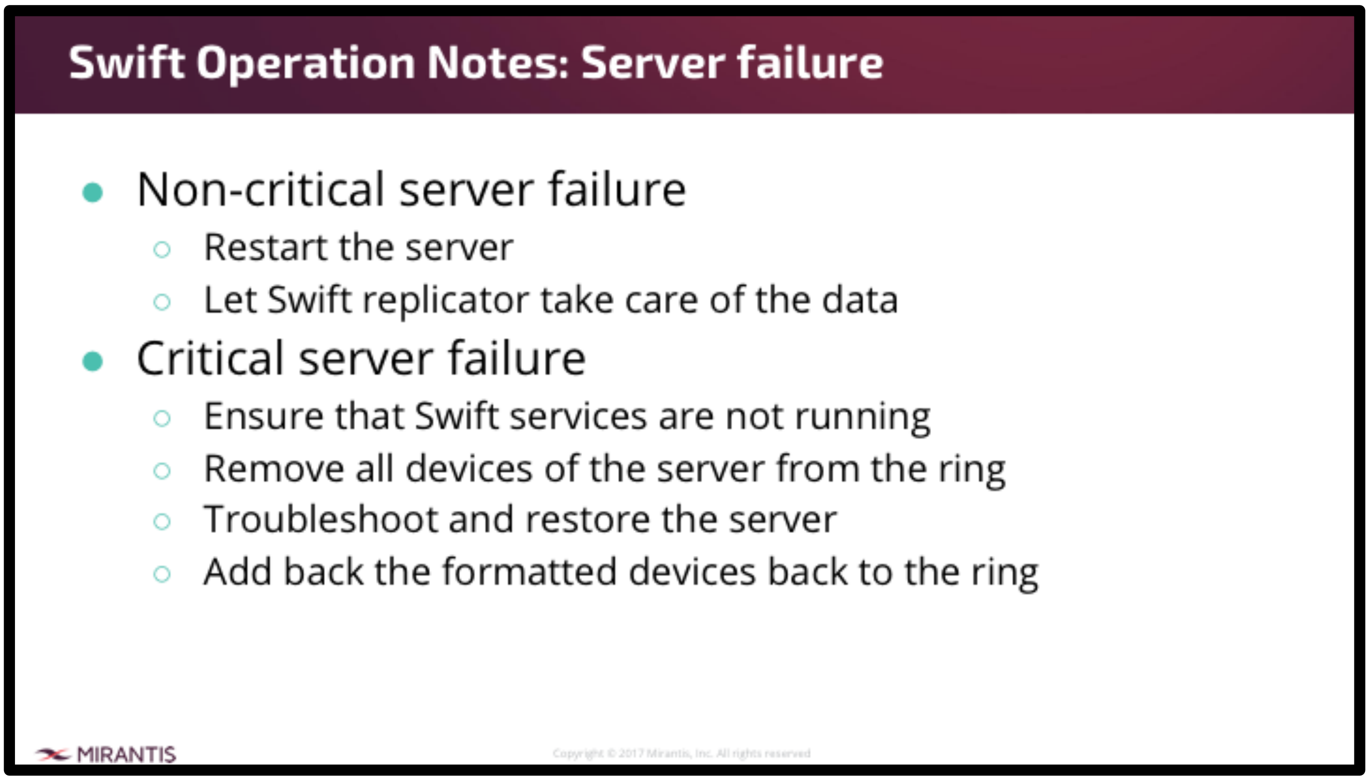


Swift Operation Notes: Disk failure

- Ensure that drive is unmounted
- If drive can be replaced immediately
  - Replace with new formated drive
  - Let Swift replication fill up the drive
- If drive cannot be replaced immediately
  - Set failed device weight to 0
  - Replicas will be distributed to different drives evenly

MIRANTIS

Source:
https://docs.openstack.org/swift/latest/admin_guide.html#handling-drive-failure

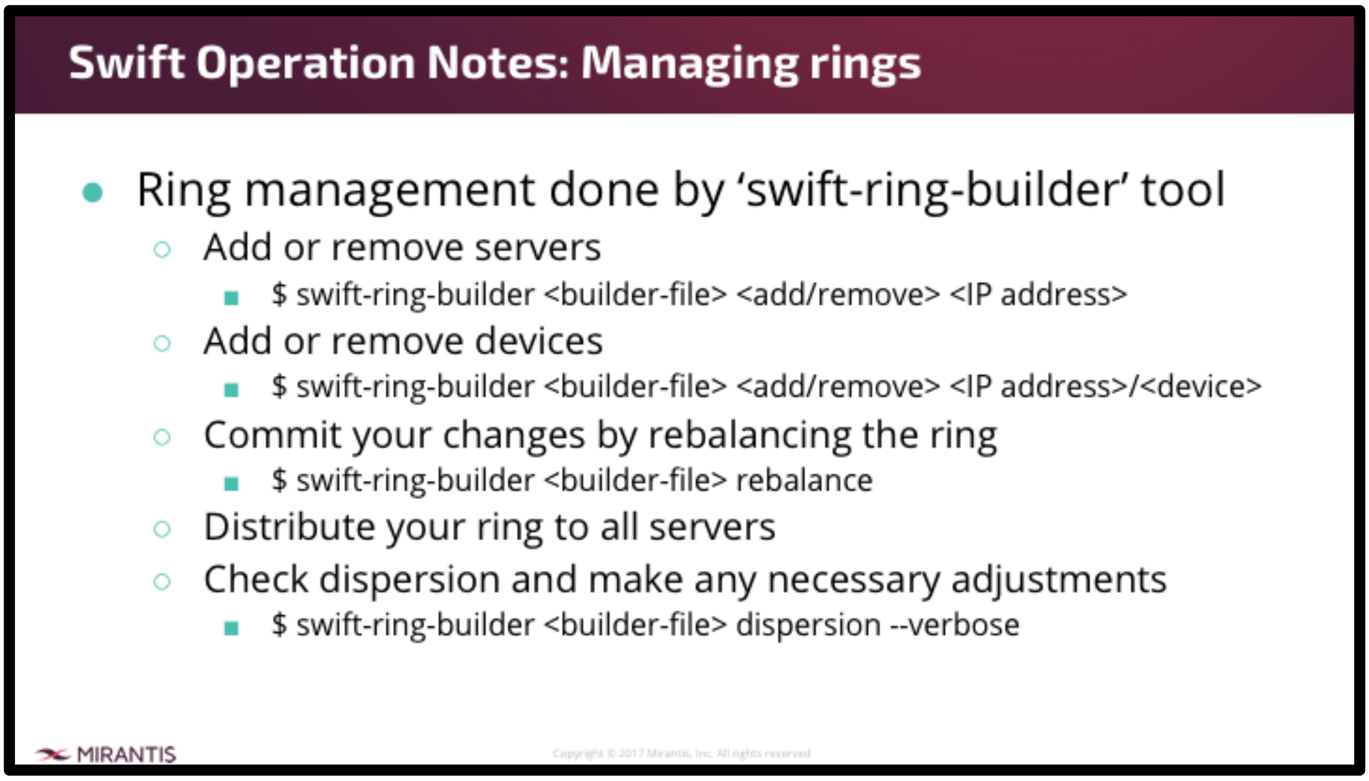


Swift Operation Notes: Server failure

- Non-critical server failure
  - Restart the server
  - Let Swift replicator take care of the data
- Critical server failure
  - Ensure that Swift services are not running
  - Remove all devices of the server from the ring
  - Troubleshoot and restore the server
  - Add back the formatted devices back to the ring

MIRANTIS

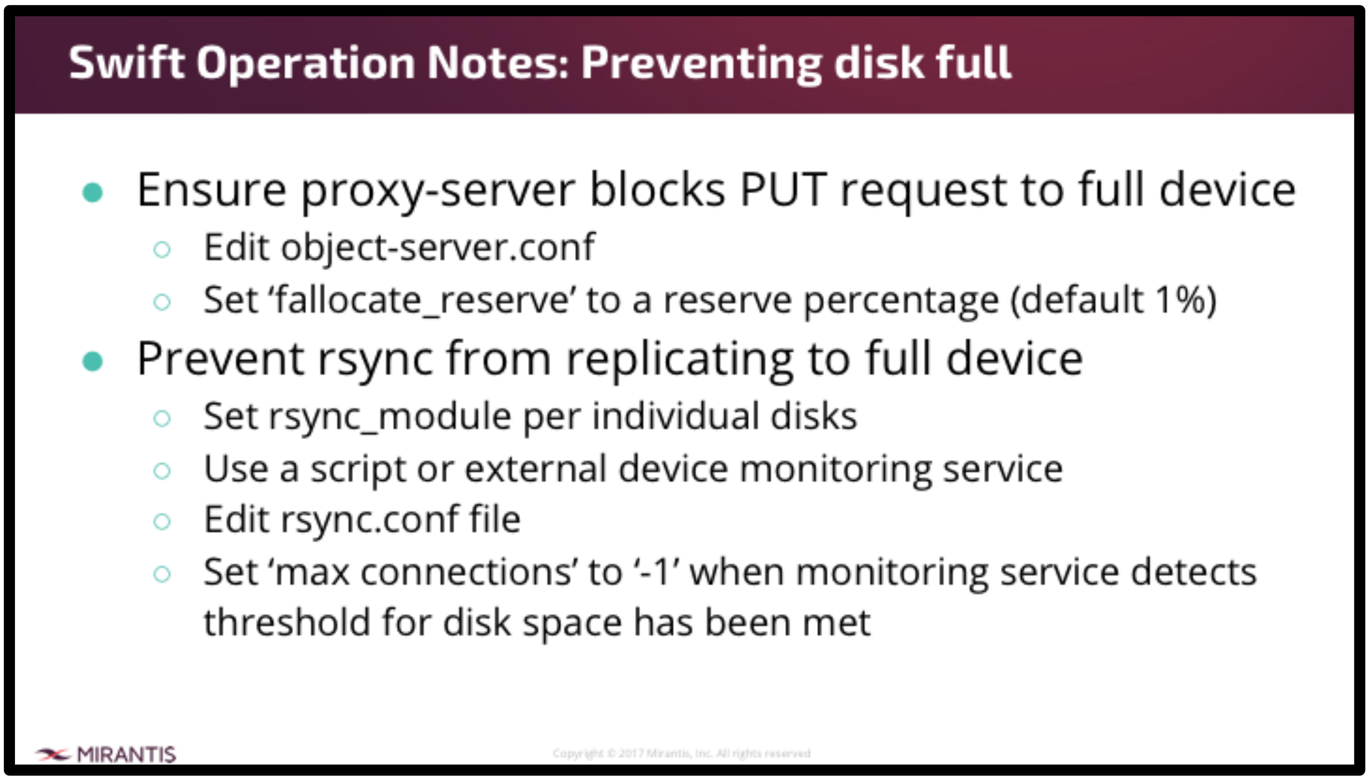
Source:
https://docs.openstack.org/swift/latest/admin_guide.html#handling-server-failure



Swift Operation Notes: Managing rings

- Ring management done by 'swift-ring-builder' tool
  - Add or remove servers
    - $ swift-ring-builder <builder-file> <add/remove> <IP address>
  - Add or remove devices
    - $ swift-ring-builder <builder-file> <add/remove> <IP address>/<device>
  - Commit your changes by rebalancing the ring
    - $ swift-ring-builder <builder-file> rebalance
  - Distribute your ring to all servers
  - Check dispersion and make any necessary adjustments
    - $ swift-ring-builder <builder-file> dispersion --verbose

MIRANTIS



Swift Operation Notes: Preventing disk full

- Ensure proxy-server blocks PUT request to full device
  - Edit object-server.conf
  - Set 'fallocate_reserve' to a reserve percentage (default 1%)
- Prevent rsync from replicating to full device
  - Set rsync_module per individual disks
  - Use a script or external device monitoring service
  - Edit rsync.conf file
  - Set 'max connections' to '-1' when monitoring service detects threshold for disk space has been met

MIRANTIS

Source:
https://docs.openstack.org/swift/latest/admin_guide.html#preventing-disk-full-scenarios