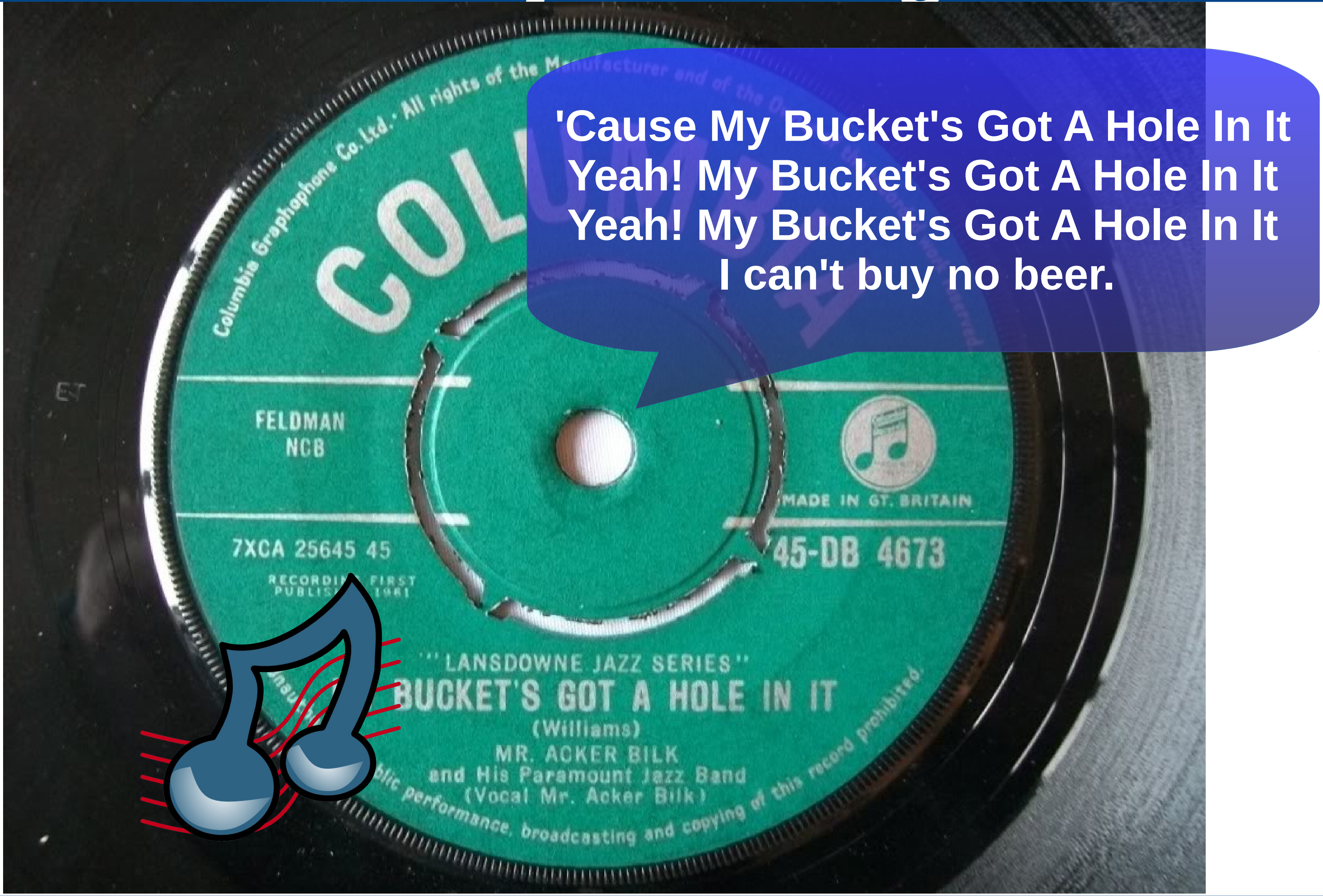
Distributed deployment of Swift Object Storage

Columbia Graphophone Co. Ltd. · All rights of the Manufacturer and of the Owner of the recorded work reserved.
COLUMBIA
FELDMAN
NCB
MADE IN GT. BRITAIN
7XCA 25645 45
RECORDING FIRST PUBLISHED 1961
45-DB 4673
"LANSDOWNE JAZZ SERIES"
MY BUCKET'S GOT A HOLE IN IT
(Williams)
MR. ACKER BILK
and His Paramount Jazz Band
(Vocal Mr. Acker Bilk)
Unauthorised public performance, broadcasting and copying of this record prohibited.

Distributed deployment of Swift Object Storage

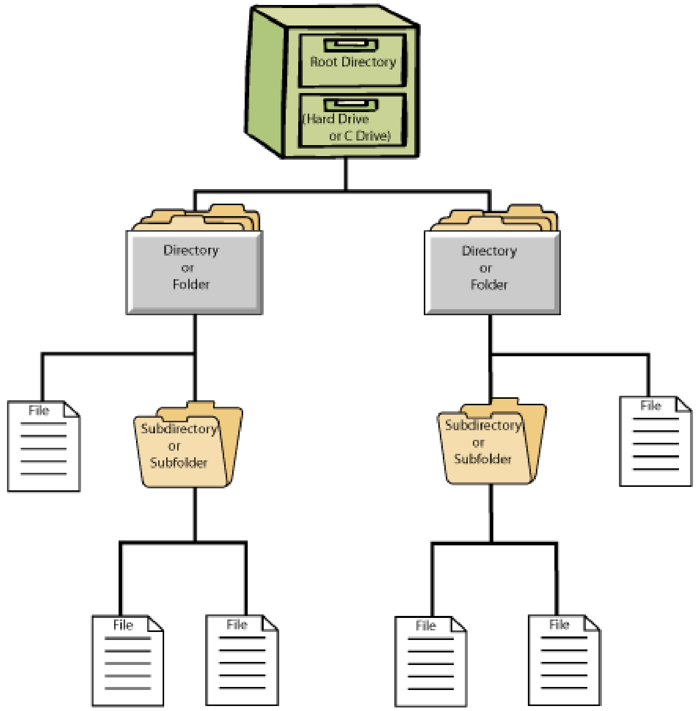

Columbia Graphophone Co. Ltd. · All rights of the Manufacturer and of the Owner of the recorded work reserved
COLUMBIA
FELDMAN
NCB
MADE IN GT. BRITAIN
7XCA 25645 45
45-DB 4673
RECORDING FIRST PUBLISHED 1961
"LANSDOWNE JAZZ SERIES"
BUCKET'S GOT A HOLE IN IT
(Williams)
MR. ACKER BILK
and His Paramount Jazz Band
(Vocal Mr. Acker Bilk)
Unauthorised public performance, broadcasting and copying of this record prohibited.

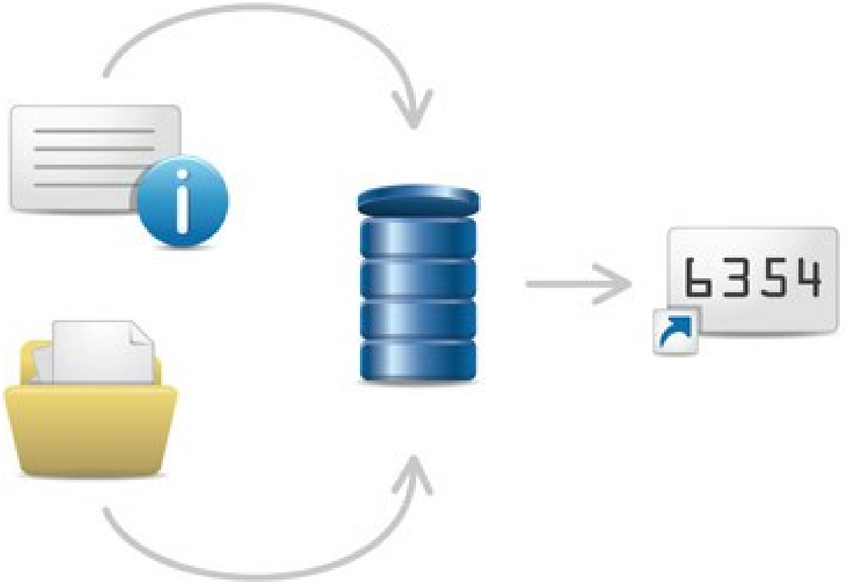
'Cause My Bucket's Got A Hole In It
Yeah! My Bucket's Got A Hole In It
Yeah! My Bucket's Got A Hole In It
I can't buy no beer.

Contents

Object Storage

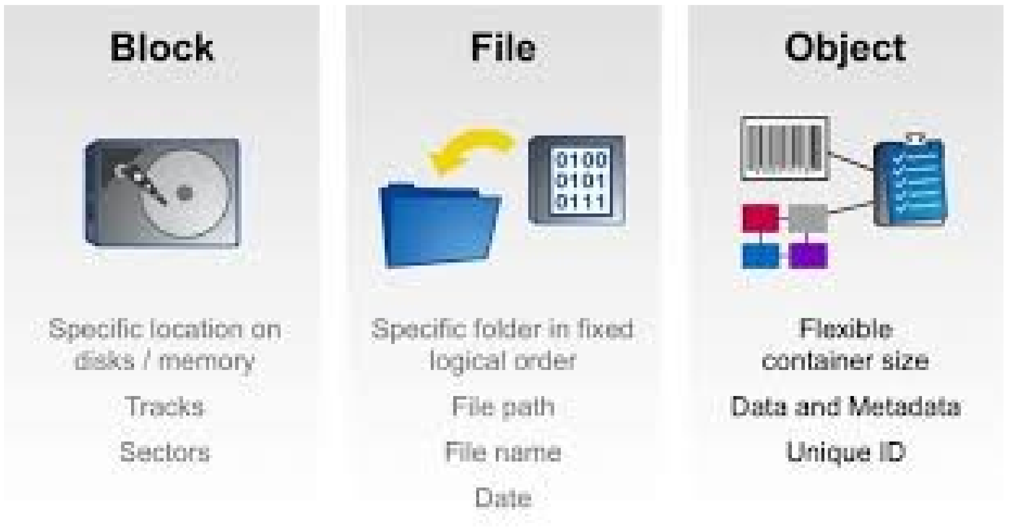
- Object Storage is a storage architecture that manages data as objects, as opposed to other storage architectures like file systems which manage data as a file hierarchy and block storage which manages data as blocks within sectors and tracks.
- Each object typically includes the **data** itself, a variable amount of **metadata**, and a globally **unique identifier**.





Object Storage /2

- **Object storage can be implemented at multiple levels**, including the device level (object storage device), the system level, and the interface level.
- In each case, object storage seeks to enable capabilities not addressed by other storage architectures, like interfaces that can be directly programmable by the application, a namespace that can span multiple instances of physical hardware, and data management functions like data replication and data distribution at object-level granularity.



Object Storage /3

Object storage systems allow relatively inexpensive, scalable and self-healing retention of massive amounts of unstructured data. Object storage is used for diverse purposes such as storing photos (Facebook), songs (Spotify), or files in online collaboration services (Dropbox).

Object Storage at device level

Seagate Kinetic Open Storage Platform

The Seagate® Kinetic Open Storage platform is the first device-based storage platform enabling independent software vendors (ISV) and cloud service provider (CSP), and enterprise customers to optimize scale-out file and object-based storage, delivering lower TCO.

Seagate Kinetic Storage comprises storage devices + key/value API + Ethernet connectivity.

Object Storage at device level

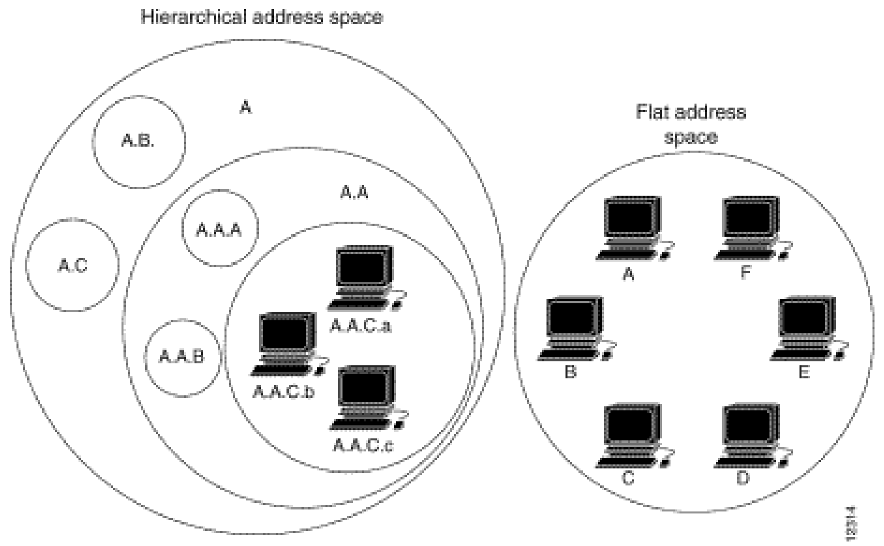
Seagate Kinetic Open Storage Platform

The Seagate® Kinetic Open Storage platform is the first device-based storage platform enabling independent software vendors (ISV) and cloud service provider (CSP), and enterprise customers to optimize scale-out file and object-based storage, delivering lower TCO.

Seagate Kinetic Storage comprises storage devices + key/value API + Ethernet connectivity.

No directory tree

- **Object Storage uses a flat structure**, storing objects in containers, rather than a nested tree structure.
- Many implementations of Object Storage can emulate a directory structure, and give the illusion of hierarchy, but in reality the underlying storage is flat.
- This is another feature of Object Storage that allows for massive scalability: by eliminating the overhead of keeping track of large quantities of directory metadata, one major performance bottleneck that is typically seen once tens of millions of files are present on a file-system is eliminated.



Strong vs Eventual Consistency

Storage systems use one of two different architectural approaches to provide scalability, performance and resiliency:

- **eventual consistency** or
- **strong consistency**

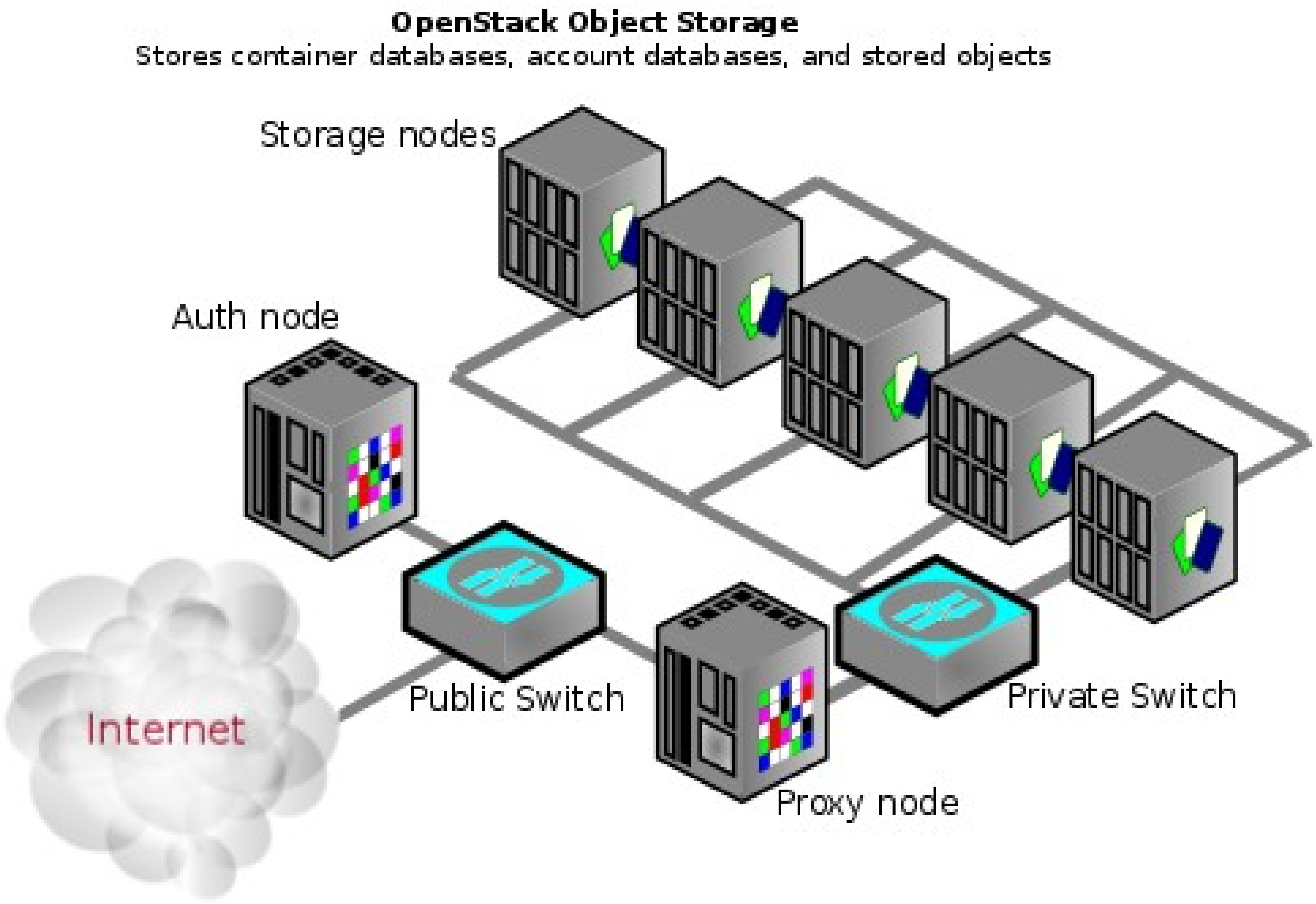
Object storage systems such as Amazon S3 and Swift are eventually consistent, which provide massive scalability and ensures high availability to data even during hardware failures.

Block storage systems and file-systems 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.

Strong vs Eventual Consistency

- **Strong consistency** is a consistency model that guarantees that if a write is successful, **any reads that happen after the write would get the latest value**.
- **Eventual consistency** is a consistency model used in distributed computing to achieve high availability 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**. Eventual consistency is widely deployed in distributed systems, often under the moniker of optimistic replication, and has origins in early mobile computing projects. A system that has achieved eventual consistency is often said to have converged, or achieved replica convergence.

Swift – Basic Architecture



Swift – Proxy Server

- The **Proxy Server** is responsible for tying together the rest of the Swift architecture. For each request, it will look up the location of the account, container, or object in the ring and route the request accordingly.
- The **public API is exposed through the Proxy Server.**
- A large number of **failures are also handled in the Proxy Server**. For example, if a server is unavailable for an object PUT, it will ask the ring for a handoff server and route there instead.
- When objects are streamed to or from an object server, they are streamed directly through the proxy server to or from the user, **the proxy server does not spool them**.

Swift – The Ring

- A **Ring** represents a mapping between the names of entities stored on disk and their physical location. There are separate rings for accounts, containers, and one object ring per storage policy. When other components need to perform any operation on an object, container, or account, they need to interact with the appropriate ring to determine its location in the cluster.
- The Ring maintains this mapping using **regions**, **zones**, **devices**, **partitions**, and **replicas**. Each partition in the ring is replicated, by default, 3 times across the cluster, and the locations for a partition are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used for handoff in failure scenarios.

Swift – The Ring /2

- Data can be isolated with the concepts of regions and zones in the ring. **Each replica of a partition is guaranteed to reside in a different zone or region**, if possible. A zone could represent a drive, a server, a cabinet, a switch, or even a datacenter.
- The partitions of the ring are equally divided among all the devices in the Swift installation. When partitions need to be moved around (for example if a device is added to the cluster), the ring ensures that a minimum number of partitions are moved at a time, and only one replica of a partition is moved at a time.
- **Weights can be used** to balance the distribution of partitions on drives across the cluster. This can be useful, for example, when different sized drives are used in a cluster.
- The ring is used by the Proxy server and several background processes (like replication).

Swift – Object Server

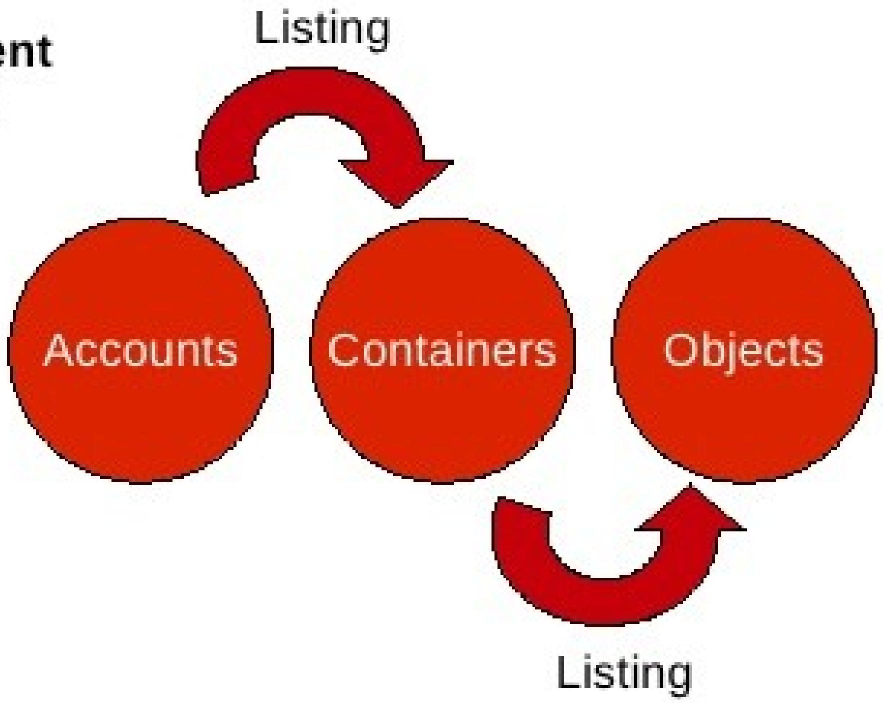
- The Object Server is a **very simple blob storage server** that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with **metadata stored in the file's extended attributes (xattrs)**. This requires that the underlying filesystem choice 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.

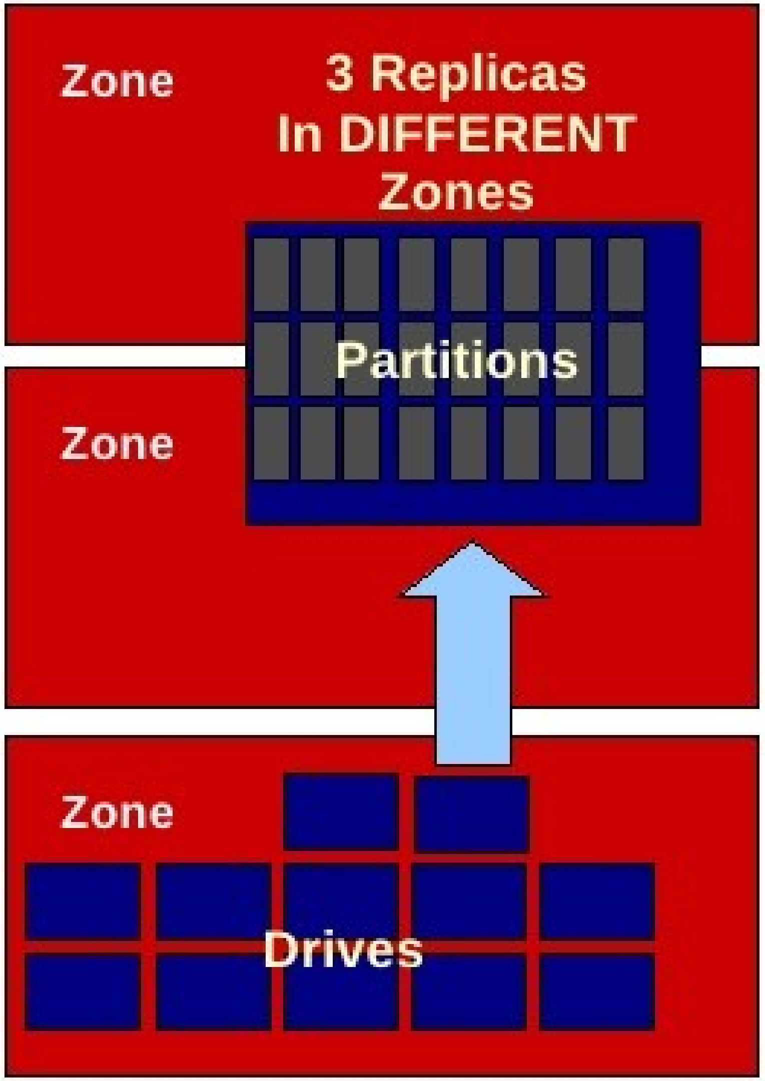
Swift – Container and Account Server

- The Container Server’s primary job is to **handle listings of objects**. It doesn’t know where those object’s 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 Account Server is very similar to the Container Server, excepting that it is **responsible for listings of containers** rather than objects.

Swift Architecture

3 Different entities:





The Ring: location of an entity in the cluster

- Three rings (were intended for the Elves)
- Weights can be used to balance the distribution
- Three different logical levels

Swift – Regions and Zones

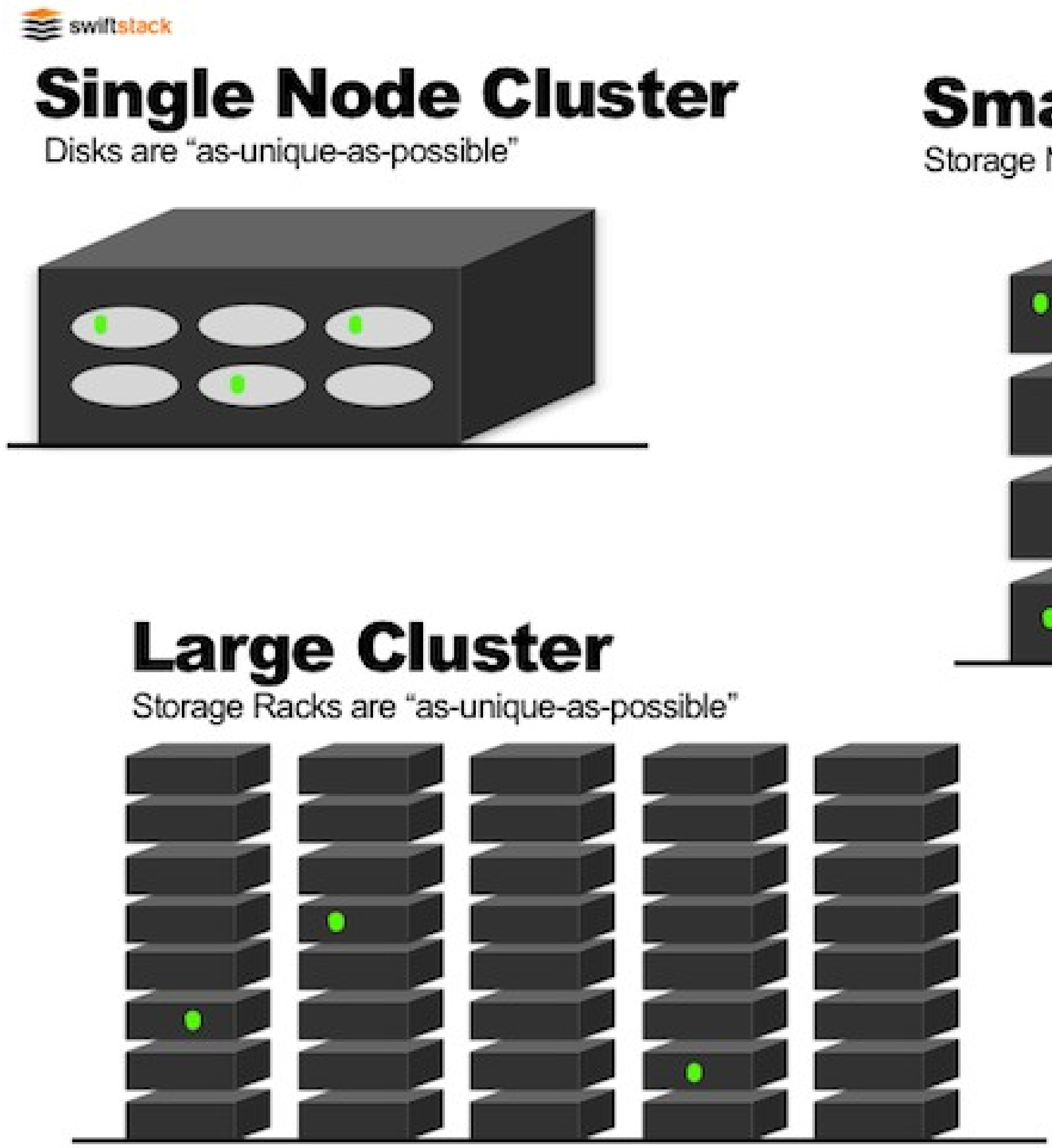
Zones are your defined single points of failure within your cluster.

- Whereas Zones are designed to distribute replicas among nodes and drives such that there is no single point of hardware/networking failure, **Regions are conceptually designed to distribute those replicas among different geographical areas**.
- Note that from Swift's perspective, there is no requirement that Regions be geographically separated. However, this is the practice that is used in general.
- The Swift object placement algorithm will attempt to place objects across regions, just as it does with zones, nodes, and drives.

Swift – Unique-as-possible placement /1

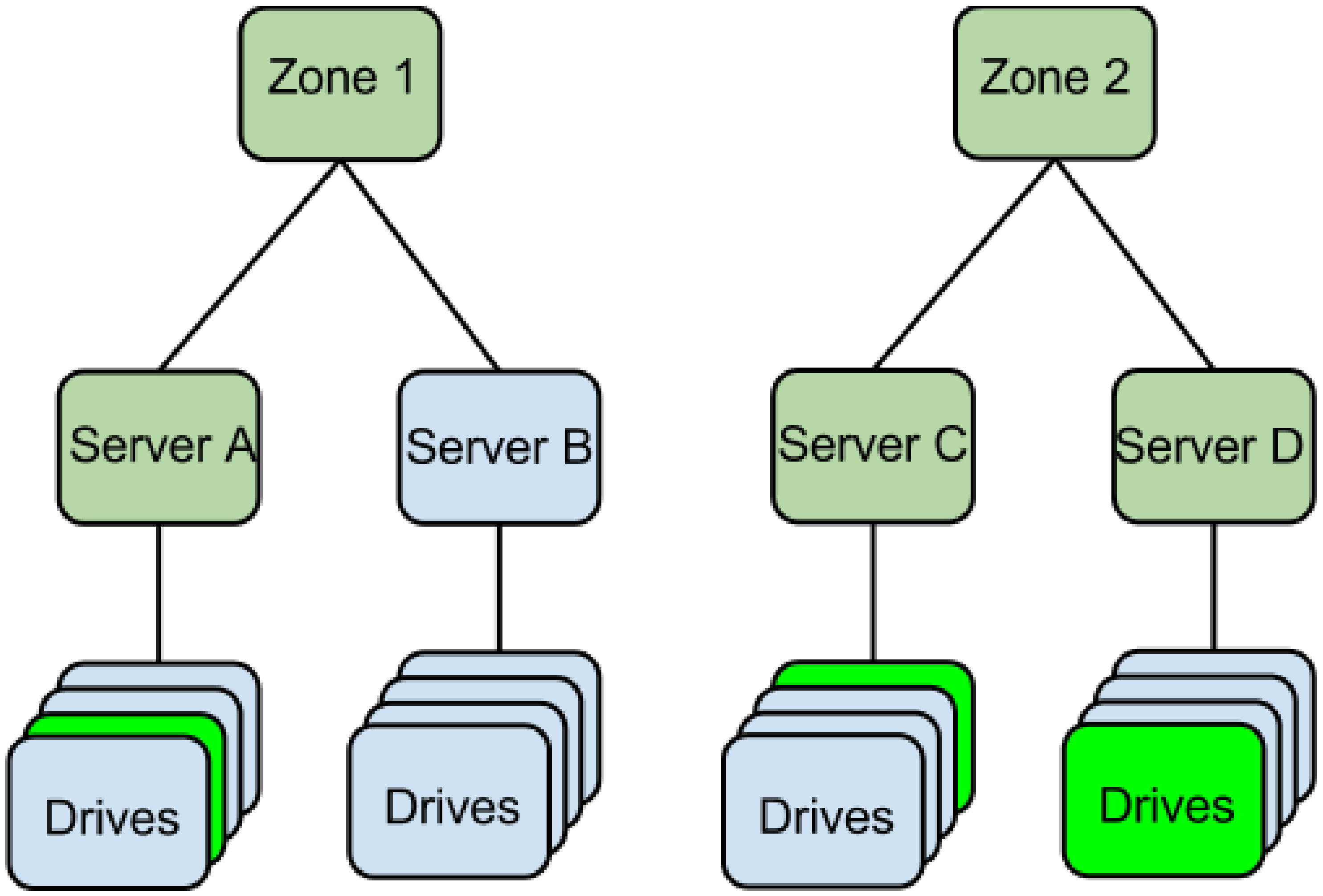
- Swift’s unique-as-possible placement works like this: data is placed into tiers–first the availability zone, next the server, and finally the storage volume itself. **Replicas of the data are placed so that each replica has as much separation as the deployment allows**.
- When Swift chooses how to place each replica, it first will choose an availability zone that hasn’t been used. If all availability zones have been chosen, the data will be placed on a unique server in the least used availability zone. Finally, if all servers in all availability zones have been used, then Swift will place replicas on unique drives on the servers.

Swift – Unique-as-possible placement /2


swiftstack
Single Node Cluster
Disks are "as-unique-as-possible"
Small Cluster
Storage Nodes are "as-unique-as-possible"
Large Cluster
Storage Racks are "as-unique-as-possible"

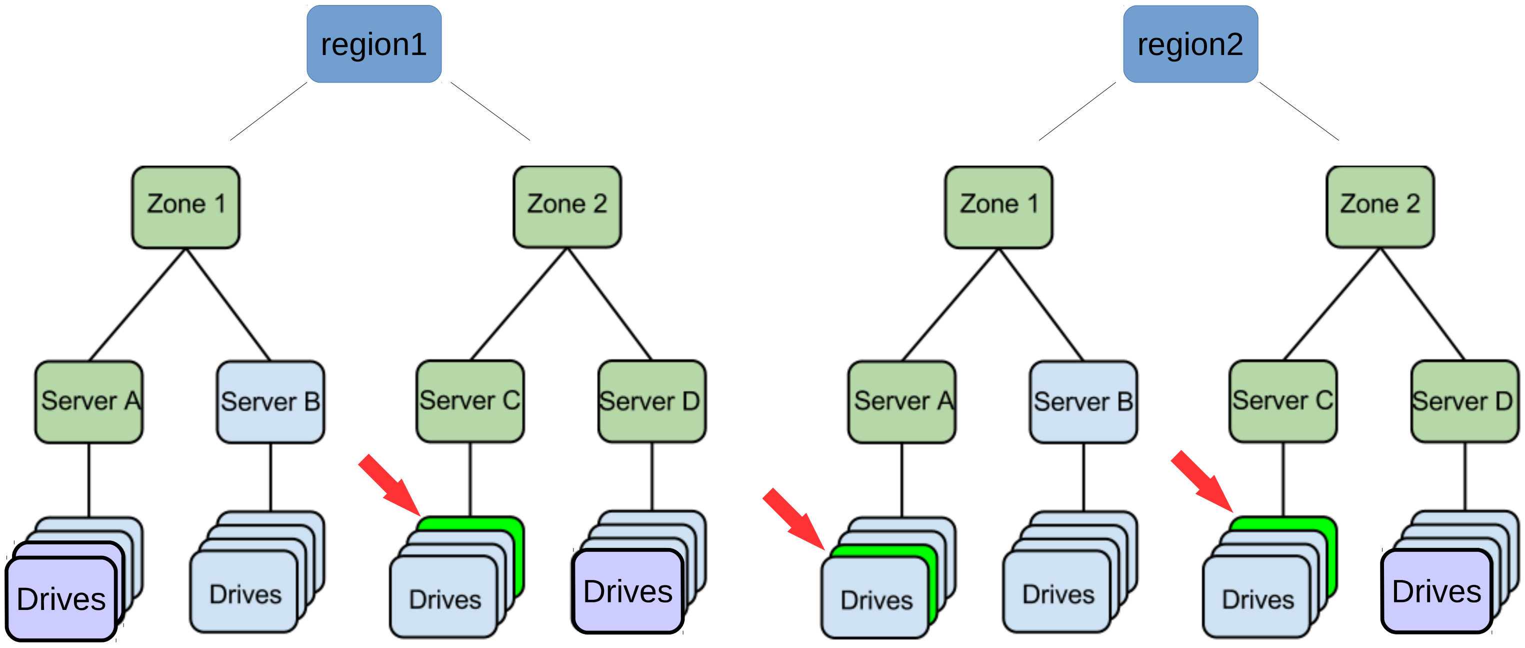
Swift – Unique-as-possible placement /3

As an example, suppose you are storing three replicas, and you have two availability zones, each with two servers.



Swift – Unique-as-possible placement /4

But if you add regions to your infrastructure.....



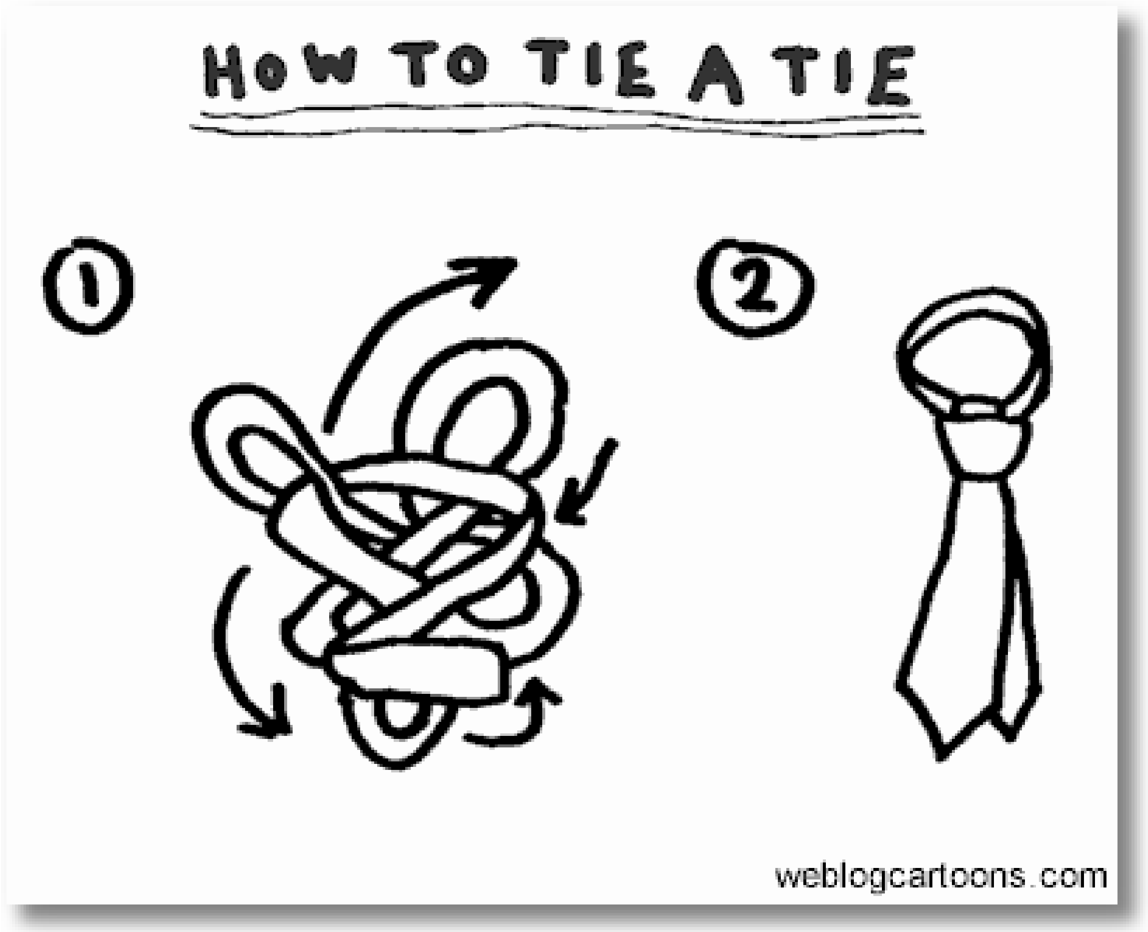
Swift – Storage Policies

- **Storage Policies** provide a way for object storage providers to differentiate service levels, features and behaviors of a Swift deployment. Each Storage Policy configured in Swift is exposed to the client via an abstract name. Each device in the system is assigned to one or more Storage Policies. This is accomplished through the use of multiple object rings, where each Storage Policy has an independent object ring, which may include a subset of hardware implementing a particular differentiation.
- For example, one might have the default policy with 3x replication, and create a second policy which, when applied to new containers only uses 2x replication. Another might add SSDs to a set of storage nodes and create a performance tier storage policy for certain containers to have their objects stored there.

Swift - Installation

A not-so-up-to-date howto:

http://wiki.infn.it/cn/ccr/cloud/cloudstorage/installazione_swift



Swift - Global Cluster Rationale

Globally distributed clusters may be desired for a number of reasons:

- **Offsite Disaster Recovery**
  - In the event of a natural disaster at the primary data center, at least one replica of all objects has also been stored at an off-site data center.
- **Active-Active / Multi-site Sharing**
  - Data stored to a data center on one side of the country is replicated to a data center on the other side of the country in order to provide faster access to clients in both locations.

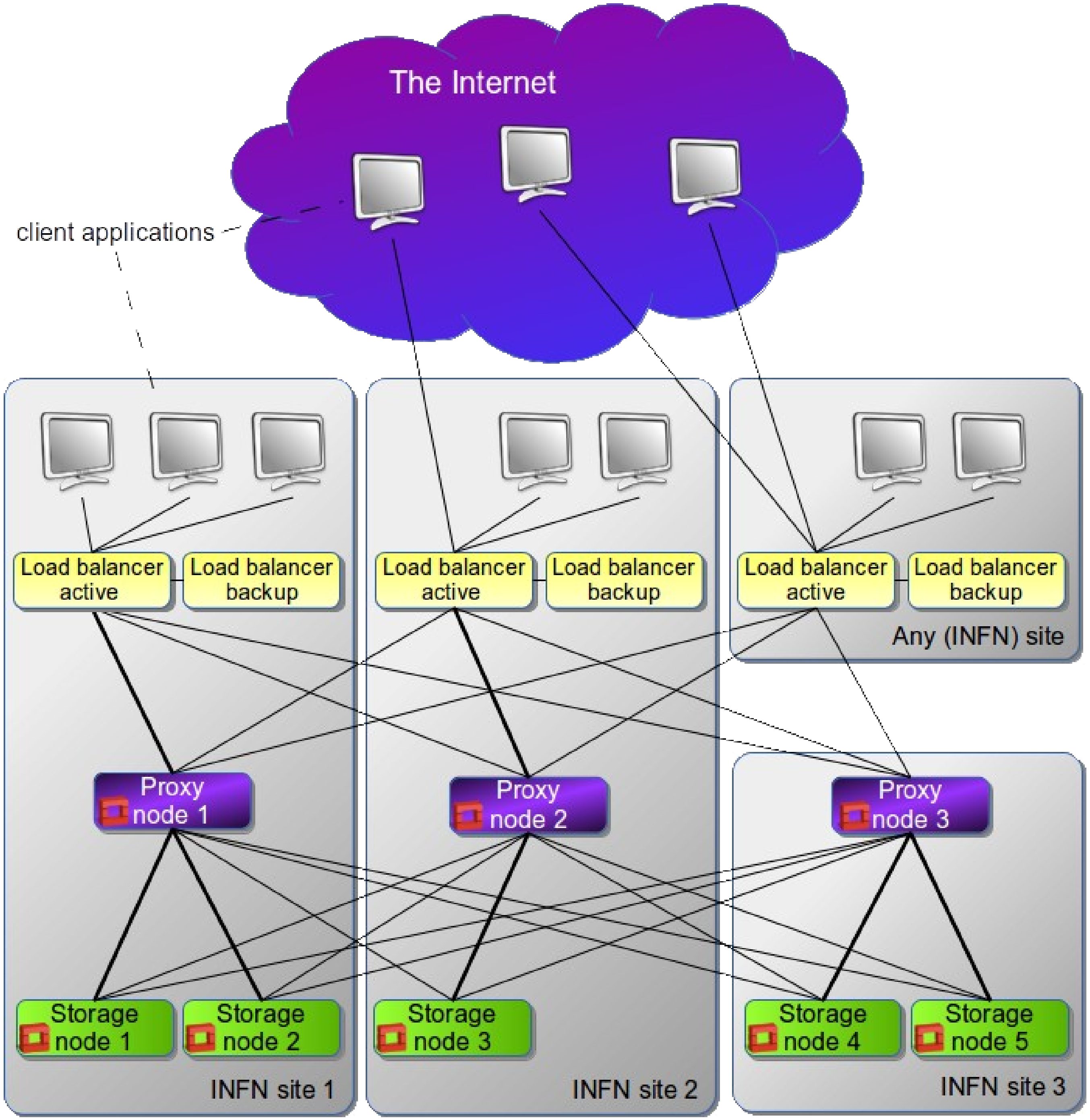
Swift - Regional Connectivity Requirements

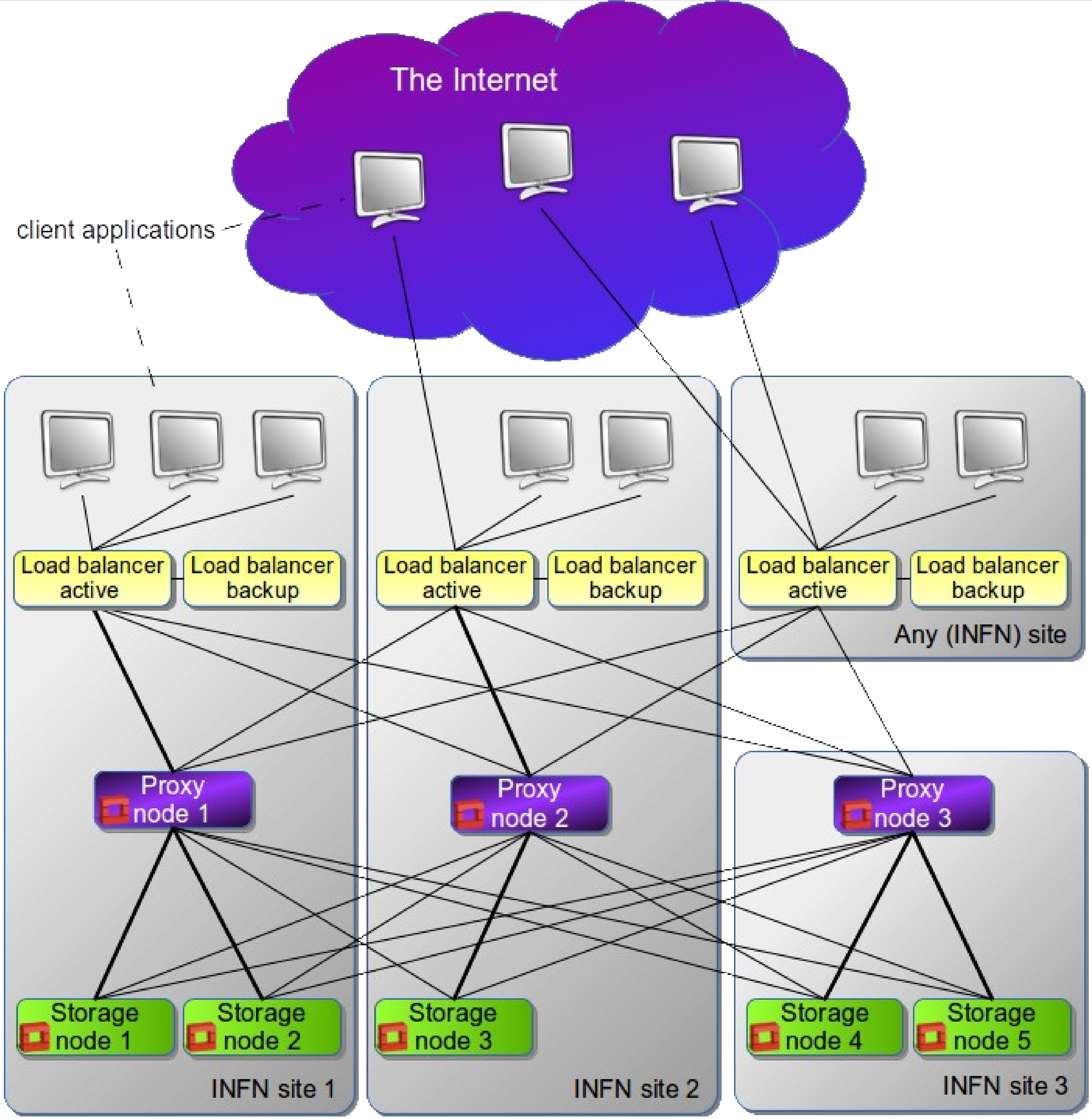
- All of the nodes within the cluster must be able to see all of the other nodes within the cluster, even across regions. Typically one of two methods is used to ensure this is possible:
  - **Private Connectivity** - site-to-site via MPLS or a private Ethernet circuit
  - **VPN Connectivity** - a standalone VPN controller through an Internet connection.
- Both methods require that the routing information, via static routes or a learned routing protocol, be configured on the storage and proxy nodes to support data transfer between regions.

Swift – distributed installations

Geographically distributed Swift deployments allow for disaster proof data storage systems and fully distributed applications with no single point of failure.

There is nothing really different, from the Swift point of view, in a geographically distributed deployment with respect to a local installation.

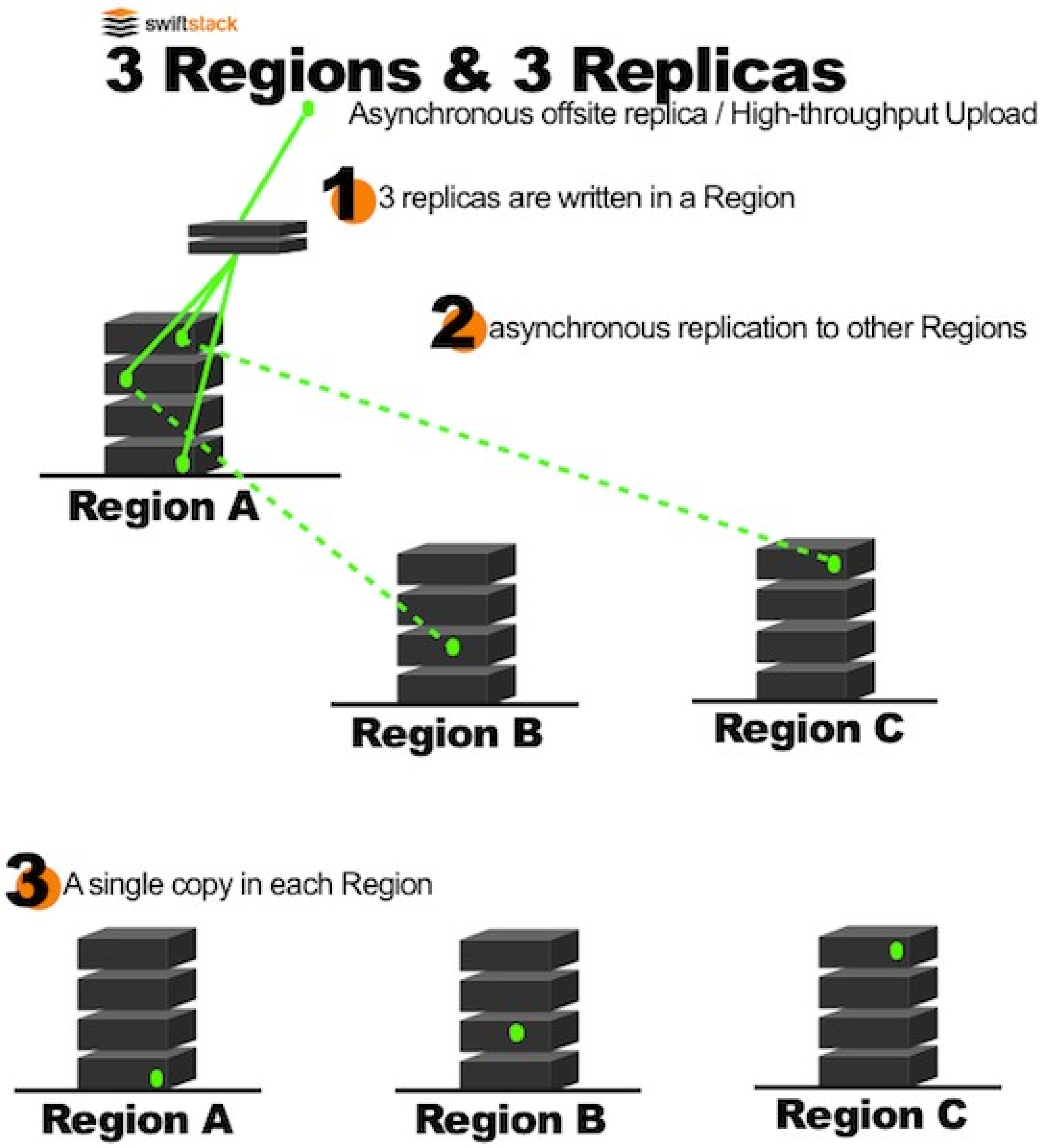



The Internet
client applications
Load balancer active
Load balancer backup
Load balancer active
Load balancer backup
Load balancer active
Load balancer backup
Any (INFN) site
Proxy node 1
Proxy node 2
Proxy node 3
Storage node 1
Storage node 2
Storage node 3
Storage node 4
Storage node 5
INFN site 1
INFN site 2
INFN site 3

Swift – Write Affinity

The write affinity feature can be extremely useful in geographically distributed environments. Data is first uploaded in one region in three copies (possibly on SSD storage) and lately transferred to the other regions.

This allows for increased upload bandwidth and gives the impression of a lower latency.


swiftstack
3 Regions & 3 Replicas
Asynchronous offsite replica / High-throughput Upload
1 3 replicas are written in a Region
2 asynchronous replication to other Regions
Region A
Region B
Region C
3 A single copy in each Region
Region A
Region B
Region C

Swift – Read Affinity

- When GET requests come into a proxy server, it attempts to connect to a random storage node where the data resides. When the Swift cluster is geographically distributed, some of that data will live in another geographic region, with a higher latency link between the proxy server and the storage node.
- Swift allows for you to **set read affinity for your proxy servers**. When enabled, the proxy server will attempt to connect to nodes located within the same region as itself for data reads. If the data is not found locally, the proxy server will continue to the remote region.

Swift – Distributed installation

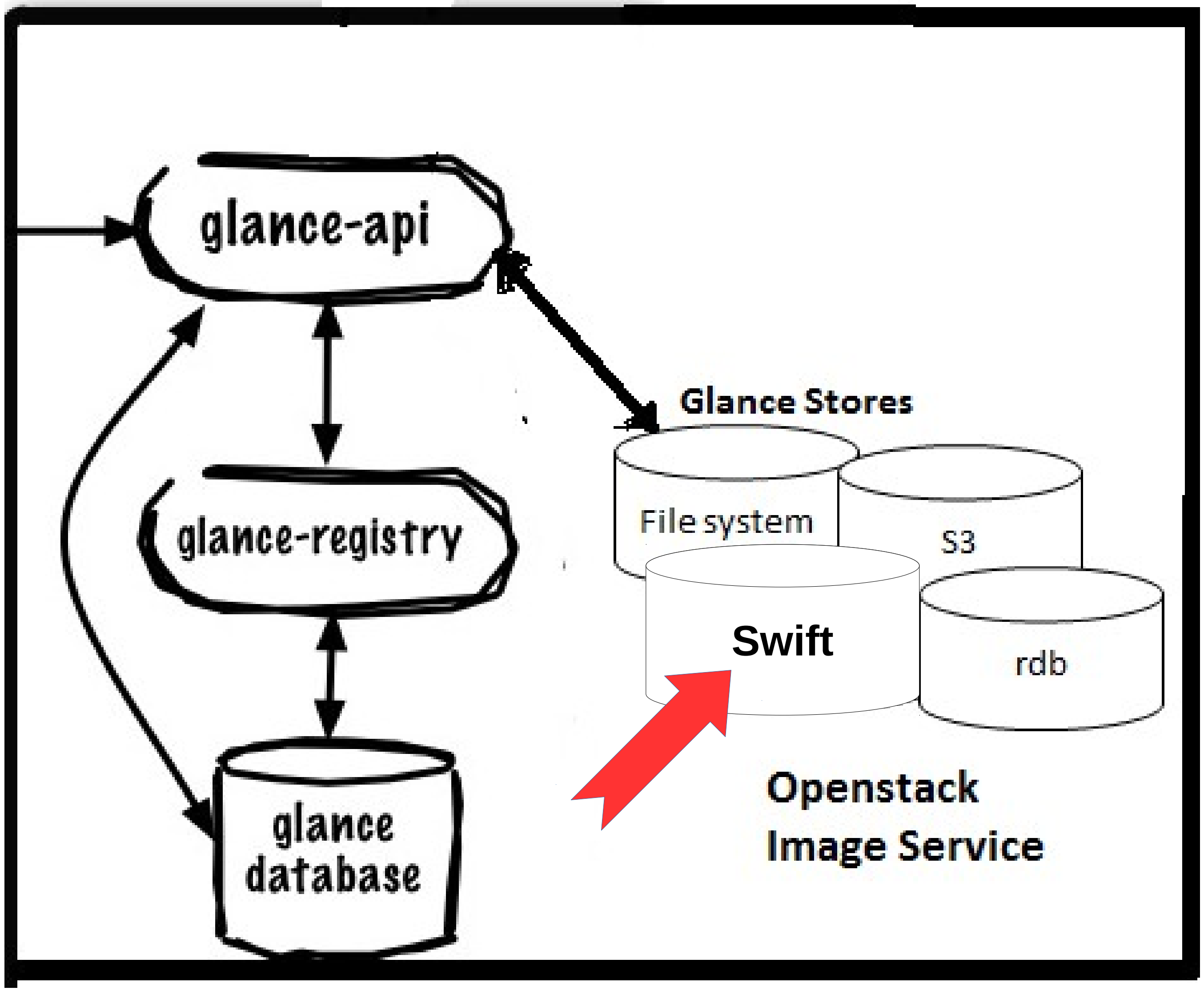
Installation of a geographically distributed Swift infrastructure

is not different from a local installation except:

- You need to take firewalls into account
- You may want to deploy multiple proxy nodes (probably one per site at least)
- You must take data and metadata security into account
- Read/Write affinity

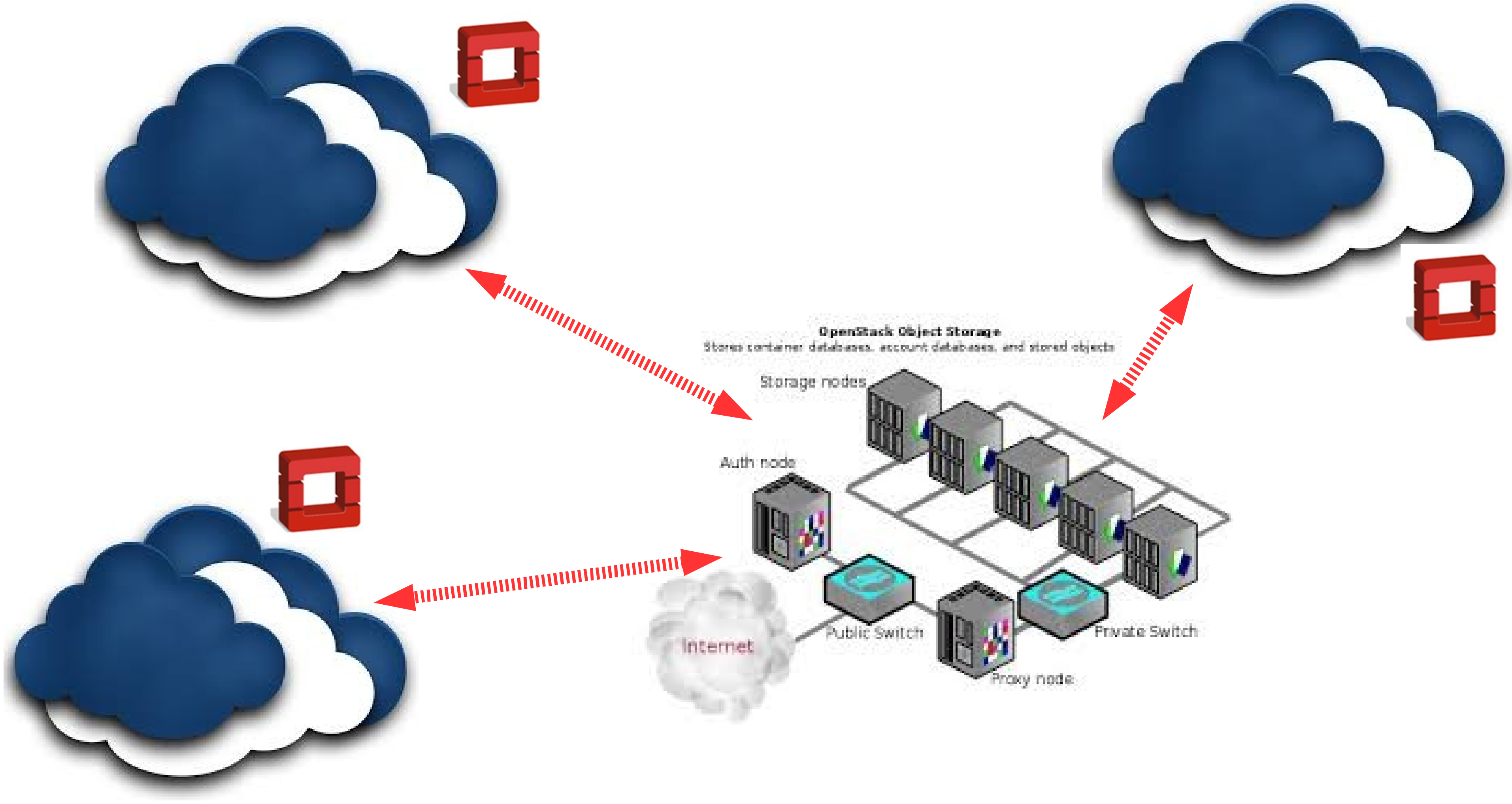
Use cases – backend for Glance

- A simple Swift use case is that of using it as a back-end for the OpenStack image service, Glance.



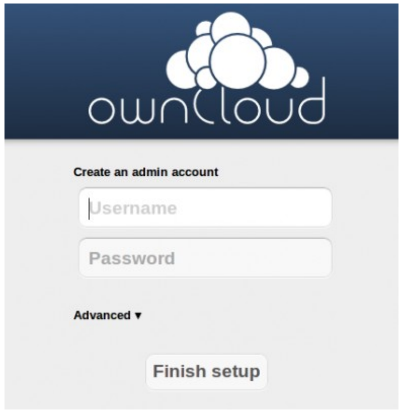
Use cases – backend for multiple Glance instances

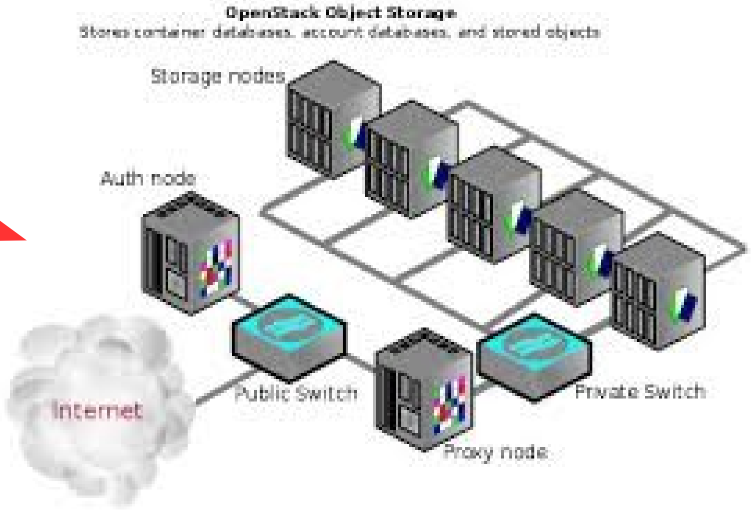
- A single Swift installation can serve multiple, federated clouds or a cloud that is distributed in multiple sites. This allows for easy geographic sharing of cloud images and for cold geographic migration of VMs. **There is no need for Swift to be enrolled to the same identity service as the one(s) used by the other cloud services.**



Use cases - backend for ownCloud

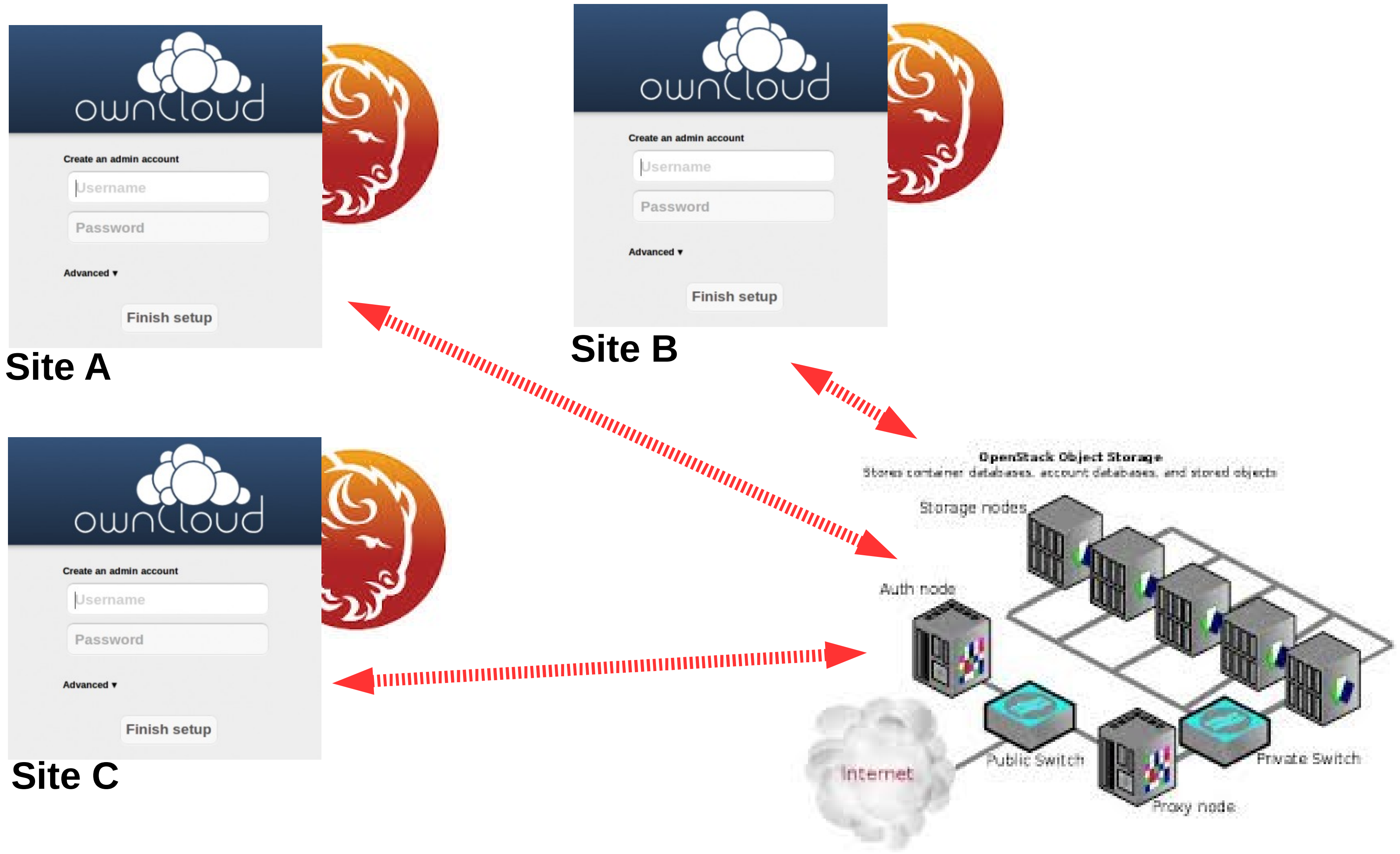
- Starting from version 7 ownCloud supports Swift both as external storage and as main storage backend.
- **This use case can be generalized for any web application (e.g. CMS, doc management, ...)**



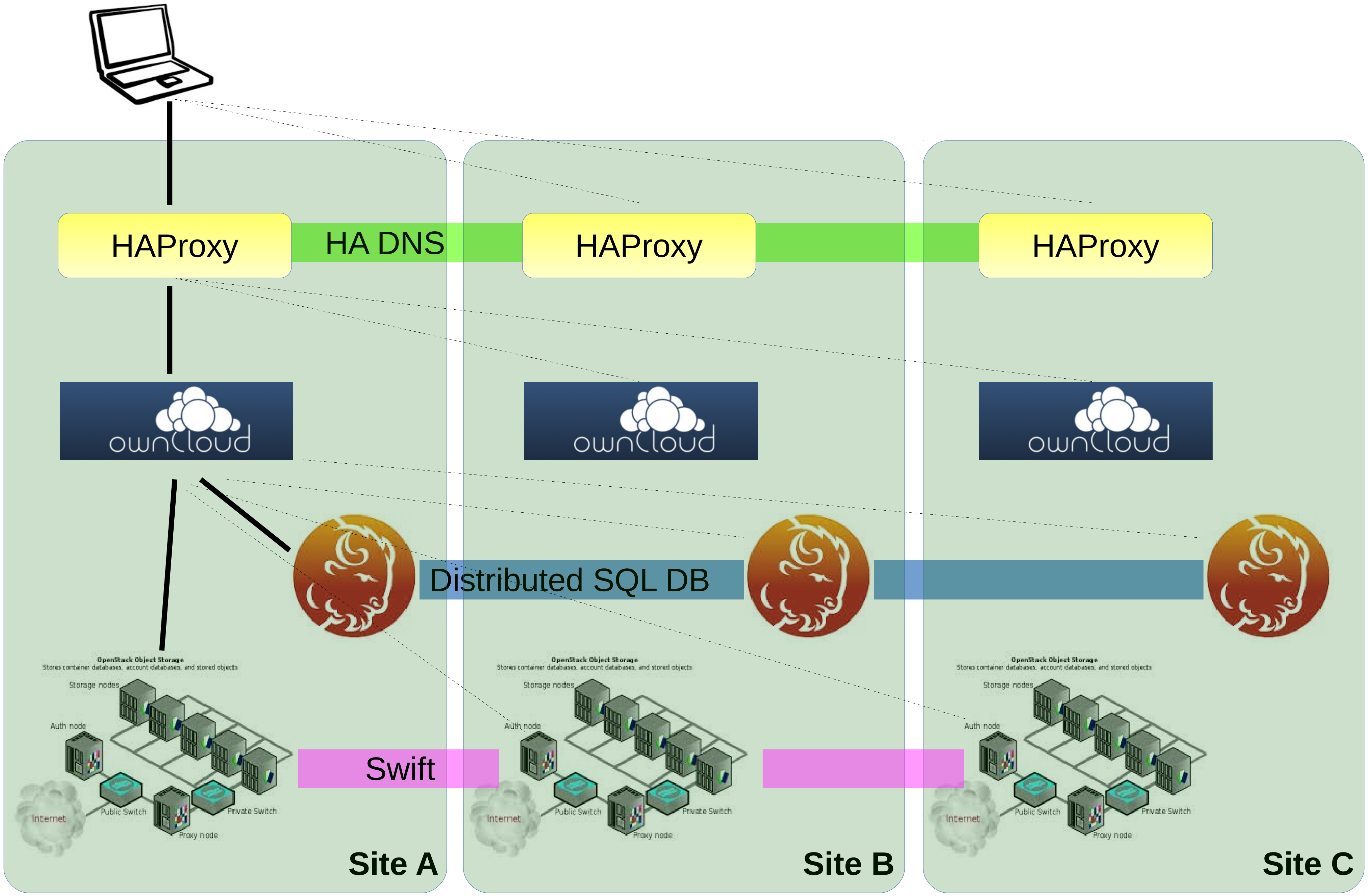


Use cases - backend for a distributed ownCloud

- Swift can also help building up an ownCloud installation with geographically distributed web server + db (e.g. Percona xtradb cluster).



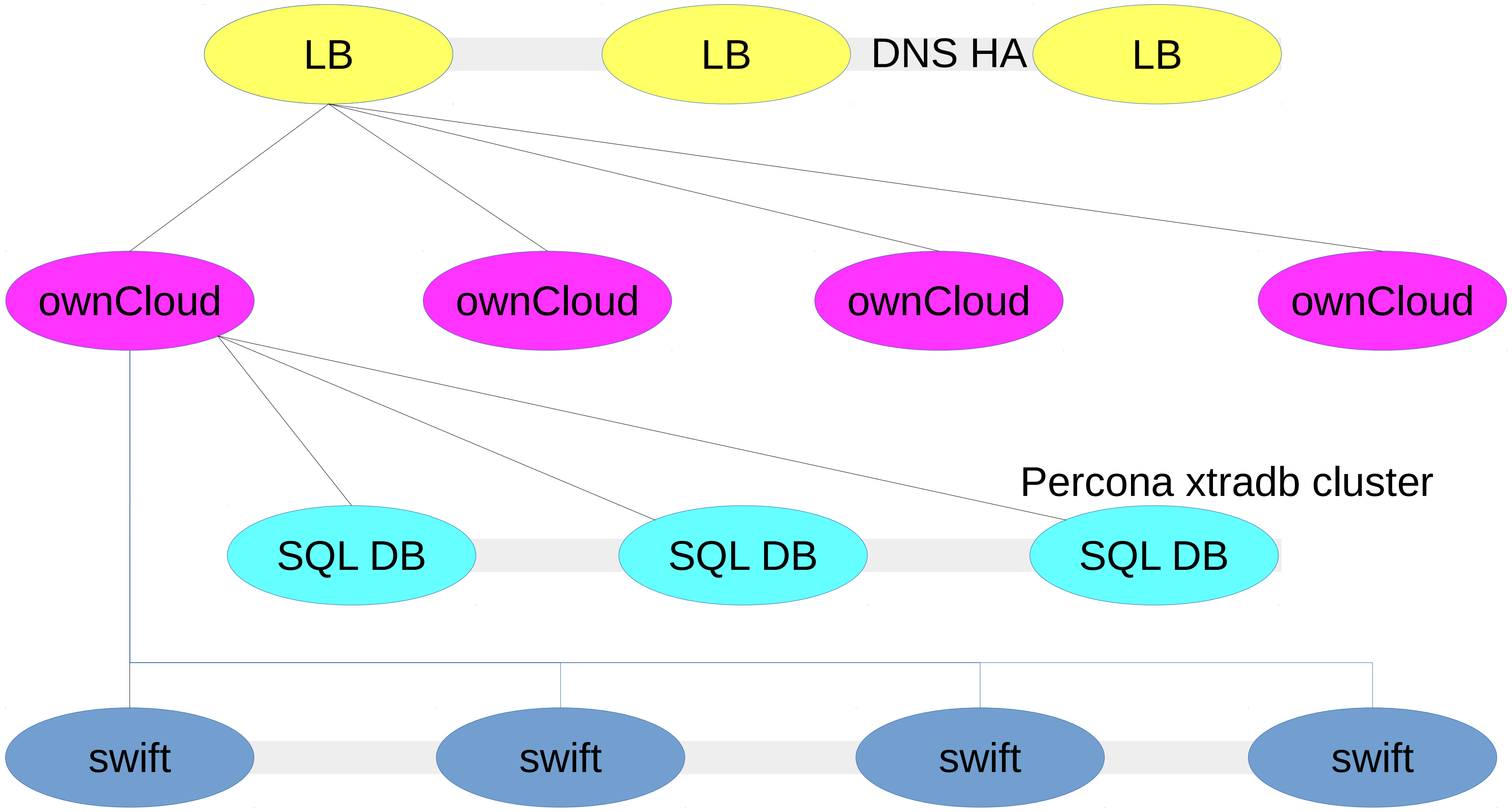
Use cases – fully distributed ownCloud



Use cases – extremely distributed ownCloud

Each **can be a different site**

Different load balancing/ha strategies can hide site failures



<u>Does this make any sense?</u>