D3.2.1

Version	1.0
Author	TUB
Dissemination	PU
Date	27/01/2015
Status	Final

D3.2.1: Cloud Repository v1

Project acronym:	NUBOMEDIA
Project title:	NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia
Project duration:	2014-02-01 to 2016-09-30
Project type:	STREP
Project reference:	610576
Project web page:	http://www.nubomedia.eu
Work package	WP3: NUBOMEDIA Cloud Platform
WP leader	TUB
Deliverable nature:	Prototype
Lead editor:	Lorenzo Tomasini
Planned delivery date	01/2015
Actual delivery date	27/01/2015
Keywords	Cloud Repository MongoDB OpenStack Swift Cluster

The research leading to these results has been funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement nº 610576

FP7 ICT-2013.1.6. Connected and Social Media

Contributors:

Giuseppe Antonio Carella (TUB)
Lorenzo Tomasini (TUB)

Internal Reviewer(s):

Micael Gallego (URJC)
Luis López (URJC)
Cristian Spoiala (USV)
Alin Calinciuc (USV)

Version History

Version	Date	Authors	Comments
0.1	07/2014	Giuseppe Carella	Created first version of the document
0.2	08/2014	Giuseppe Carella, Lorenzo Tomasini	Created structure and added initial content
0.3	09/2014	Lorenzo Tomasini	Added introduction
0.4	12/2014	Lorenzo Tomasini	Add requirements section, fixed style paragraphs
0.5	12/2014	Lorenzo Tomasini	Fixed Table of Contents, Starting writing Requirements section
0.6	12/2014	Lorenzo Tomasini	Add MongoDB Cluster section
0.7	12/2014	Lorenzo Tomasini	Add Introduction
0.8	12/2014	Lorenzo Tomasini	Add installation
0.9	01/2015	Lorenzo Tomasini	Fixed minor comment
1.0	01/2015	Lorenzo Tomasini	Final Version ready to be reviewed

Table of contents

List of Figures

Acronyms

AAA	Authentication, Authorization and Accounting
CRUD	Create, Read, Update and Delete
KMS	Kurento Media Server
QoS	Quality of Service
VM	Virtual Machine

1. Executive summary

This document concerns the description of the Cloud Repository component. The following sections will provide an overview regarding the Cloud Repository requirements. Based on that, an analysis of the state of the art is provided and finally the tool employed is described in detail.

2. Introduction

The component Cloud Repository has to provide storage capabilities of media content to applications and services. It also has to supply reliability and High Availability. The recovering process of media content has to be on demand and efficient in order to let the applications and services retrieve the media content from the repository and establish QoS.
A cloud repository system is usually composed by different entities. A usual cloud repository configuration is composed by:

- A monitoring / configuration server: this entity has the task of monitoring the state of the whole system and to maintain a structured overview of it. In big environments, this entity is spread over different servers.
- A storage server: this entity is the real storage location. It contains all the media content. In normal environments, the media content is spread over multiple storage servers, so a server can be a master storage server or just a replica of a master server.
- A routing server: this entity has the task of load balancer. It redirects the requests to the correct storage server, basing its choice on load, bandwidth, location of the requested data and so on. In big environments, this entity is spread over different servers.

In order to achieve a good level of efficiency, more storage servers are deployed thus the media elements are stored over different storage servers. And that is the reason of the presence of routing servers. These routing servers contribute to the efficiency as well.

In order to achieve High Availability and reliability, in the system has to be present replicas of media content. This is achieved by replica set of storage server. A replica can contain a copy of a whole storing server or just partial copies of media contents. This replication is also providing a fault tolerance feature. In case of fail of a main storage server, the replica can fulfill the request at its place.

Summarizing, the Cloud Repository offers a reliable, High Available repository service, strongly based on open source tools. It is able to fulfill requests demanding media content and able to store / retrieve this media content efficiently in a distributed, cloud and in a constantly evolving environment.

3. Requirement Analysis

In this section all the requirements that the Cloud Repository solution needs to satisfy will be evaluated.

3.1. Considerations

In NUBOMEDIA requirement analysis, as described in deliverable D2.3.1, no explicit cloud repository requirement were found. However, several general, high-level requirements are specified as the shown in the analysis table of the following subsection.

3.1. General requirements

The following figure shows the general architecture and interfaces needed for the Cloud Repository:

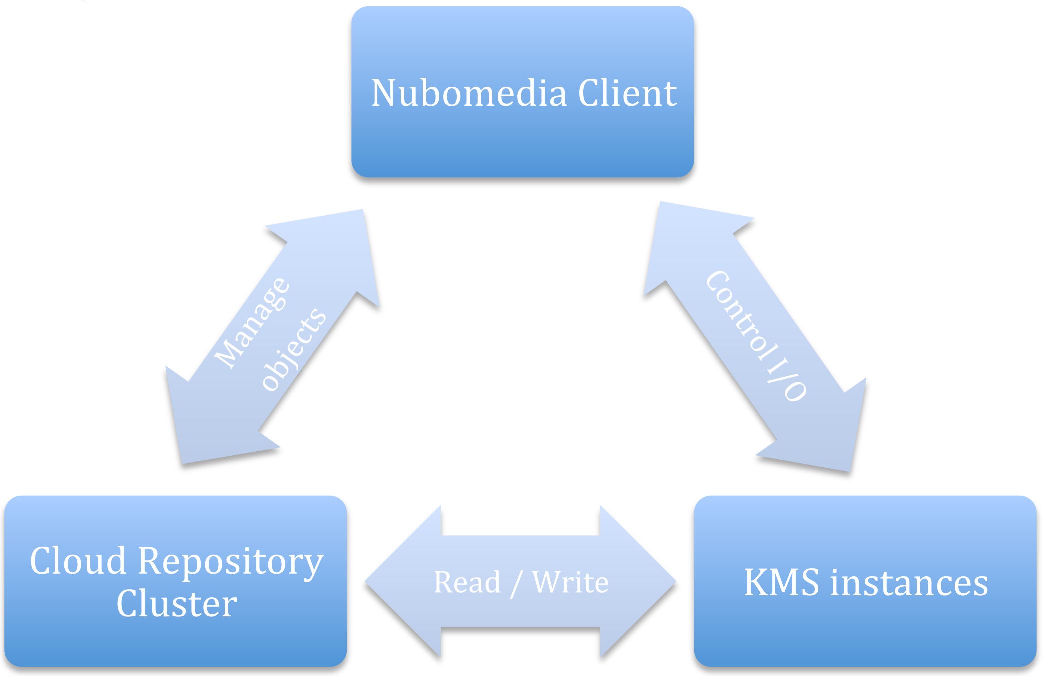


Figure 1: General Architecture Cloud Repository

Where:

- KMS Instances are instances of Kurento Media Server service
- NUBOMEDIA Clients are typically applications that use the Kurento API (kurento-client) in order to interact with the KMS instances and the Media Repository.
- The Cloud Repository Cluster is a set of hosts providing the Cloud Repository functionalities.
- Repository has to provide two different APIs for letting the application to manage repository objects (including METADATA) and for allowing the KMS to read and write Multimedia Data.

3.1.1. Interfaces

Between NUBOMEDIA client and Cloud Repository:
The Cloud Repository has to allow the Media applications to use the media repository. For doing that it must provide the classic APIs concerning the management of repository objects, including creation, deletion and metadata association.
Between NUBOMEDIA client and KMS instances:
This interface allows NUBOMEDIA applications to instantiate player and recording endpoints.
Between Cloud Repository and KMS instances:
The KMS instances have to be able to retrieve Media objects from the Cloud Repository and also to modify them. In order to make this possible a KMS instance will need to implement this functionality encapsulating a driver for the chosen Cloud Repository File System.

3.2. Table of requirements

ID	Item	Priority	Request by
R1	Solution based on open source tools	High	WP3
R2	Storing media content	High	WP3
R3	Storing media content reliably	High	WP3
R7	Media content delivery	High	WP3
R9	Reliability	High	WP3
R10	Distributed	High	WP3
R14	Able to store media pipelines as media applications	High	WP5 Task 5.3 (WWW Developer Portal)
R15	Storing flows (containing audio + video + multisensory)	High	Validation of Objective 3 (Generalized multimedia as video + audio + multisensory data)
R4	Efficient recovery of media content	Normal	WP3
R5	Elastic storage capabilities	Normal	WP3
R6	QoS guaranteed	Normal	WP3
R8	Scalability	Normal	WP3

R11	Automated replication on different servers	Normal	WP3
R12	Incorporate security mechanisms	High	WP4 Subtask 4.1.2 (Distributed media security)
AR1	NUBOMEDIA applications have to be able to directly manage repository objects. The following actions must be available: ● CREATE: create an object ● DELETE: delete an object ● ADD_METADATA: add metadata to an object • DEL_METADATA: remove metadata to an object	High	Application Requirement
AR2	Multimedia DATA (media objects) has to be available from KMS cluster, allowing the KMS instance to execute both READ and WRITE operations.	High	Application Requirement
AR3	Players and recorders must be able to retrieve resources from identifiers that could be URLS (file, http, rtsp, etc) and repository IDs.	High	Application Requirement
AR4	A way to let an application communicate with the Cloud Repository File System has to be available.	High	Application Requirement
AR5	An Authentication, Authorization and Accounting (AAA) protocol has to be available. Multi tenant platforms require AAA functions in order to restrict the access to the resources based on ownership, permissions, quotas, etc. • **Identity**: Each request has to be associated to a user identity. the Identity must be unique and persistent. Multiple identities	Normal	Application Requirement

	can be used simultaneously, covering concepts like: user or group. • **Ownership**: Each object has an ownership relationship with one or more identities. • **Permission**: Define a set of actions that can be performed on objects. Permissions are normally described in terms of Access Control List (ACL). Each one defines the list of actions (verbs) that creates one identity can apply to a one object.		

4. State of the Art

This section aims to give an overview of the principal available open source Cloud Repositories besides from OpenStack Swift and MongoDB GridFS that will be deeply investigated in sections OpenStack Swift Analysis and MongoDB GridFS Analysis.

4.1. Ceph

Ceph [1] is a open source software storage platform designed to present object, block, and file storage from a single distributed computer cluster. Ceph's main goals are to be completely distributed without a single point of failure, scalable to the exabyte level, and freely-available. The data is replicated, making it fault tolerant.

4.1.1. Components

4.1.1.1. Object Storage

Ceph implements distributed object storage. Ceph’s software libraries provide client applications with direct access to the reliable autonomic distributed object store (RADOS) object-based storage system, and also provide a foundation for some of Ceph’s features, including RADOS Block Device (RBD), RADOS Gateway, and the Ceph File System.
The librados software libraries provide access in C, C++, Java, Python and PHP. The RADOS Gateway also exposes the object store as a RESTful interface which can present as both native Amazon S3 and OpenStack Swift APIs.

4.1.1.2. Block Storage

Ceph's object storage system allows users to mount Ceph as a thinly provisioned block device. When an application writes data to Ceph using a block device, Ceph automatically stripes and replicates the data across the cluster. Ceph's RADOS Block Device (RBD) also integrates with kernel virtual machines (KVMs).
Ceph RBD interfaces with the same Ceph object storage system that provides the librados interface and the CephFS file system, and it stores block device images as objects. Since RBD is built on top of librados, RBD inherits librados's capabilities, including read-only snapshots and revert to snapshot. By striping images across the cluster, Ceph improves read access performance for large block device images.

Integrations

The block device is supported in virtualization platforms, such as:

- Apache CloudStack
- OpenStack
- OpenNebula
- Ganeti
- Proxmox Virtual Environment

These integrations allow administrators to use Ceph's block device as the storage for their virtual machines in these environments.

4.1.1.3. FileSystem

Ceph's file system (CephFS) runs on top of the same object storage system that provides object storage and block device interfaces. The Ceph metadata server cluster provides a service that maps the directories and file names of the file system to objects stored within RADOS clusters. The metadata server cluster can expand or contract, and it can rebalance the file system dynamically to distribute data evenly among cluster hosts. This ensures high performance and prevents heavy loads on specific hosts within the cluster. Clients mount the POSIX-compatible file system using a Linux kernel client. On March 19, 2010, Linus Torvalds merged the Ceph client into Linux kernel version 2.6.34 which was released on May 16, 2010. An older FUSE-based client is also available. The servers run as regular Unix daemons.

4.1.2. How it works

Whether you want to provide Ceph Object Storage and/or Ceph Block Device services to Cloud Platforms, deploy a Ceph Filesystem or use Ceph for another purpose, all Ceph Storage Cluster deployments begin with setting up each Ceph Node, your network and the Ceph Storage Cluster. A Ceph Storage Cluster requires at least one Ceph Monitor and at least two Ceph OSD Daemons. The Ceph Metadata Server is essential when running Ceph Filesystem clients.

Figure 2 Ceph main components

- **Ceph OSDs**: A Ceph OSD Daemon (Ceph OSD) stores data, handles data replication, recovery, backfilling, rebalancing, and provides some monitoring information to Ceph Monitors by checking other Ceph OSD Daemons for a heartbeat. A Ceph Storage Cluster requires at least two Ceph OSD Daemons to achieve an active and clean state when the cluster makes two copies of your data (Ceph makes 2 copies by default, but you can adjust it).
- **Monitors**: A Ceph Monitor maintains maps of the cluster state, including the monitor map, the OSD map, the Placement Group (PG) map, and the CRUSH map. Ceph maintains a history (called an "epoch") of each state change in the Ceph Monitors, Ceph OSD Daemons, and PGs.
- **MDSs**: A Ceph Metadata Server (MDS) stores metadata on behalf of the Ceph Filesystem (i.e., Ceph Block Devices and Ceph Object Storage do not use MDS). Ceph Metadata Servers make it feasible for POSIX file system users to execute basic commands like ls, find, etc. without placing an enormous burden on the Ceph Storage Cluster.

Ceph stores a client's data as objects within storage pools. Using the CRUSH algorithm, Ceph calculates which placement group should contain the object, and further calculates which Ceph OSD Daemon should store the placement group. The CRUSH algorithm enables the Ceph Storage Cluster to scale, rebalance, and recover dynamically.

4.1.3. Architecture

Ceph uniquely delivers object, block, and file storage in one unified system. Ceph is highly reliable, and easy to manage. Ceph delivers scalability to thousands of clients accessing petabytes to exabytes of data. A Ceph Node leverages commodity hardware and intelligent daemons, and a Ceph Storage Cluster accommodates large numbers of nodes, which communicate with each other to replicate and re-distribute data dynamically.

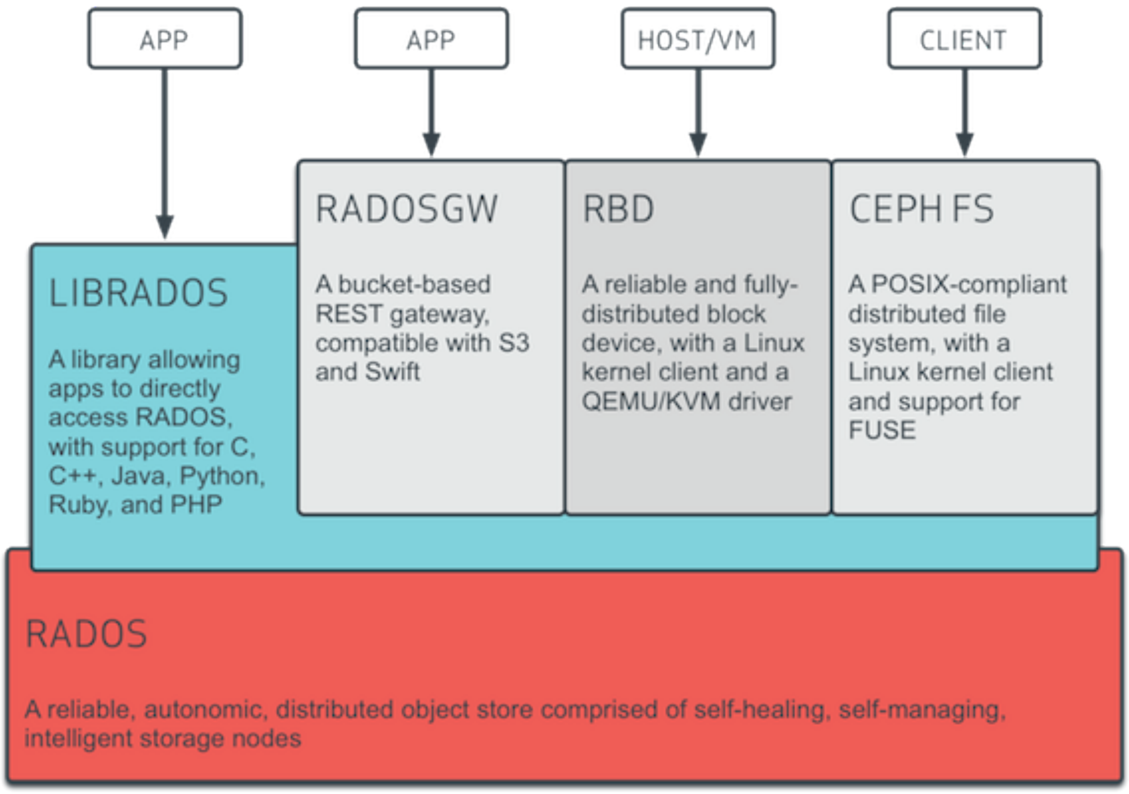

APP
APP
HOST/VM
CLIENT
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOS
A reliable, autonomic, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

Figure 3 Ceph Architecture

4.2. GlusterFS

GlusterFS [2] is an open source, distributed scalable file system capable of handling thousands of clients. GlusterFS is a powerful network/cluster filesystem; it handles the filesystem in the user space taking advantage of FUSE [7] in order to connect itself with VFS layer. GlusterFS exploits a layered approach to combine itself with the file system, where features are added/removed depending on the requirements. Though GlusterFS is a File System, it can use already tried and tested disk file systems as for example ext3, ext4, xfs, etc. to store the data. It can easily scale up to petabytes of storage, which are available to user through a single mount point. GlusterFS puts together storage building blocks over Infiniband RDMA or TCP/IP interconnect, combining together disk and memory resources and managing data in a single global namespace. GlusterFS is designed following a stackable user space and can deliver exceptional performance for diverse workloads.
The GlusterFS principal elements are:

- **Brick**: The brick is the storage filesystem that has been assigned to a volume.
- **Client**: the machine that mounts the volume (this may also be a server).
- **Server**: the machine (virtual or bare metal) that hosts the actual filesystem in which data will be stored.
- **Subvolume**: a brick after being processed by at least one translator.
- **Volume**: the final share after it passes through all the translators.
- **Translator**: A translator connects to one or more subvolumes, does something with them, and offers a subvolume connection.

- **Distribute**: Distribute takes a list of subvolumes and distributes files across them, effectively making one single larger storage volume from a series of smaller ones.

The brick's first translator (or last, depending on what direction data is flowing) is the storage/posix translator that manages the direct filesystem interface for the rest of the translators.

All the translators hooked together to perform a function is called a graph.

The configuration of translators (since GlusterFS 3.1) is managed through the gluster command line interface (CLI), so you don't need to know how the order of the

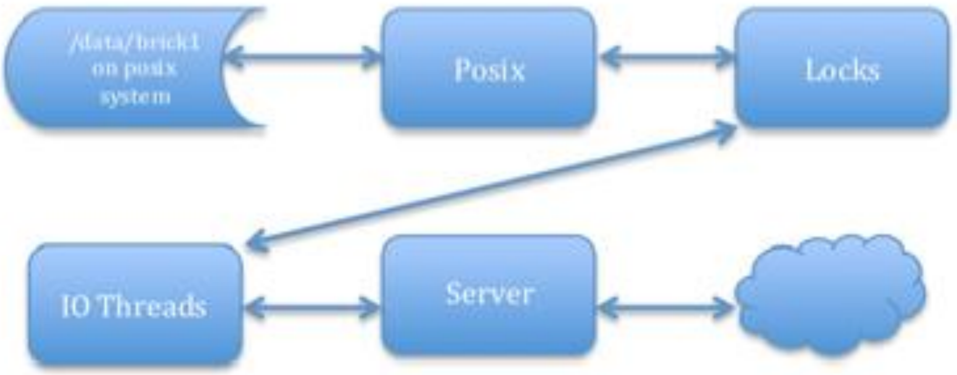


Figure 4 Complete brick graph

GlusterFS supports standard clients running standard applications over any standard IP network. Figure 5 illustrates how users can access application data and files in a global namespace using a variety of standard protocols.

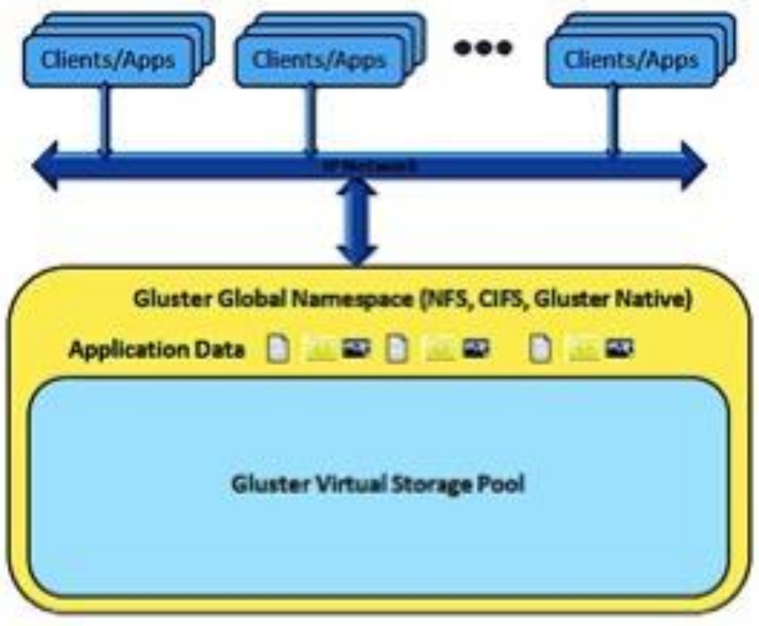

Clients/Apps
Clients/Apps
Clients/Apps
IP Network
Gluster Global Namespace (NFS, CIFS, Gluster Native)
Application Data
Gluster Virtual Storage Pool

Figure 5 GlusterFS – One Common Mount Point

GlusterFS gives users the ability to deploy scale-out, virtualized storage – scaling from terabytes to petabytes in a centrally managed pool of storage. The aim of GlusterFS is also to avoid locked users into pricey, monolithic and legacy storage platform. It provides these features:

- Scalability and Performance
- High Availability
- Global Namespace
- Elastic Hash Algorithm
- Elastic Volume Manager
- Gluster Console Manager
- Standards-based

4.3. MogileFS

MogileFS [3] is an open source distributed filesystem. MogileFS architecture comprises storage nodes that are the real place where the files are stored. The trackers, the MogileFS processes, communicate with the application and they have well-known hostnames. They use a DataBase server in order to store metadata. Domains distinguish files for different applications and classes distinguish file types. A more detailed view on the principal elements is described below:

- **Application**: client that wants to store/load files
- **Tracker** (the MogilefFS process): it handles all the communications between the application and the "query workers", not only the requests of performing and operation but also the ones include load balancing. It is an event-based parent process/message bus and it manages all communication between MogileFSd child processes. The child processes under MogileFSd include:
  - *Replication*: replicating files around
  - *Deletion*: deletes from the namespace are immediate; deletes from *filesystems* are asynchronous
  - *Query*: answering requests from clients
  - *Reaper*: re-queuing files for replication after a disk fails
  - *Monitor*: monitor health and status of hosts and devices
  - ...
- **Database**: the database that stores the MogileFS metadata (the namespace, and which files are where). There should be an HA configuration for the database placement for avoiding single point of failure
- **Storage Nodes**: real place where files are stored. The storage nodes are HTTP servers that provide operations as DELETE, PUT, etc. Any WebDAV server is fine, but mogstored is recommended. Mogilefsd can be configured to use two servers on different ports.

Its properties and features include:

- **No single point of failure:** all three components of a MogileFS setup (storage nodes, trackers, and the tracker's database(s)) can be run on multiple machines, so there's no single point of failure. (you can run trackers on the same machines as storage nodes, too, so you don't need 4 machines...) A minimum of 2 machines is recommended.

- **Automatic file replication**: files, based on their "class", are automatically replicated between enough different storage nodes as to satisfy the minimum replica count as requested by their class. For instance, for a photo hosting site you can make original JPEGs have a minimum replica count of 3, but thumbnails and scaled versions only have a replica count of 1 or 2. If you lose the only copy of a thumbnail, the application can just rebuild it. In this way, MogileFS (without RAID) can save money on disks that would otherwise be storing multiple copies of data unnecessarily.
- **"Better than RAID"**: in a non-Storage Area Network RAID setup, the disks are redundant, but the host isn't. If you lose the entire machine, the files are inaccessible. MogileFS replicates the files between devices that are on different hosts, so files are always available.
- **Flat Namespace**: Named keys in a flat, global namespace identify files. There is no limit on how many namespace is possible to create so multiple applications with potentially conflicting keys can run on the same MogileFS installation.
- **Shared-Nothing**: MogileFS doesn't depend on a pricey Storage Area Network with shared disks. Every machine maintains its own local disks.
- **No RAID required**: Local disks on MogileFS storage nodes can be in a RAID, or not. MogileFS already provides all the features that RAID does, so it is cheaper not to use it.
- **Local filesystem agnostic**: Local disks on MogileFS storage nodes can be formatted with your filesystem of choice (ext3, XFS, etc.). MogileFS does its own internal directory hashing so it doesn't hit filesystem limits such as "max files per directory" or "max directories per directory".

5. OpenStack Swift Analysis

Swift [4] is the default data repository in the OpenStack project. It is the key component for creating redundant, scalable data storage by using clusters of standardized servers. It uses a distributed architecture with no central point of control. By using this approach, it provides greater scalability, redundancy and permanence because the data are written to multiple hardware devices. If any node failed, Swift replicates its content from other active nodes. The storage clusters are scaled horizontally by adding new nodes in case of any lacks of performance or storage capacity.

5.1. Swift Details

The architecture (see following figure) of Swift consists of different components, all in all, a big cluster. These are on the one hand the physical devices such as proxy servers and storage nodes and on the other hand the logical units, for instance the rings, zones, accounts, containers, objects and partitions. Every unit is assigned to a specific task.

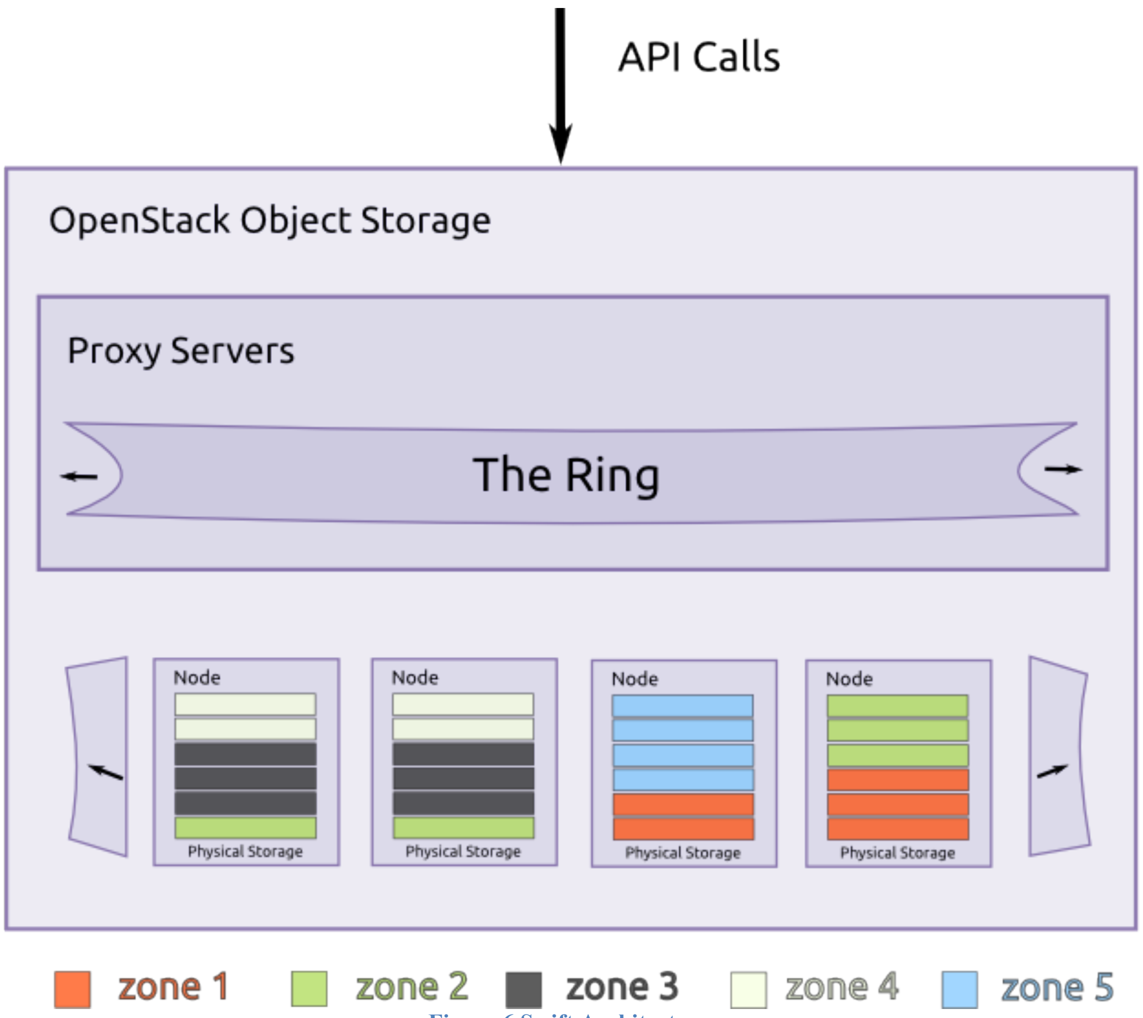


Figure 6 Swift Architecture

The **proxy servers** are responsible to handle all the incoming API requests. They are the public interface of the Object storage service and can be scaled as needed based on the workload. The **rings** map the logical names of data to the locations on particular disks knowing all the corresponding devices, partitions, replicas and zones. The **zones** are an important unit for the reliability. Failures of one zone don't impact other zones in the cluster because the data is replicated across them. By default, everything is stored three times. So there are three replicas. If any disk fails, the data is automatically distributed to another zone. Furthermore, each **account** has an individual database and holds a list of all containers that belongs to it. Each **container** can also contain many **objects** (see following figure).

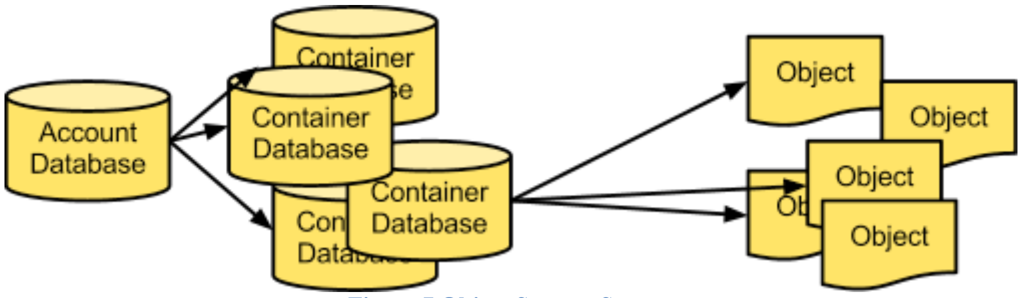


Figure 7 Object Storage System

The **partitions** (see next figure) store these account databases, container databases and objects. It helps to manage locations to get know where the data can be found in the cluster. It is also the core of the replication system. Conceptually, these partitions are just a directory on a disk that are addressed through a hash table.

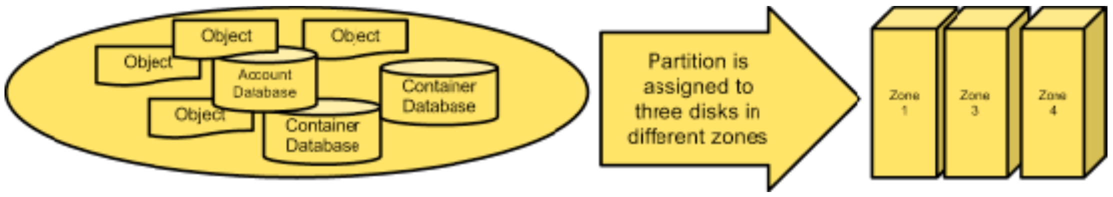


Figure 8 Partitions

The key characteristic of Swift is the object-based storage system that stores content and metadata as objects.

As indicated before, the architecture of that Object Storage system is organized in a hierarchy. The top-level of that hierarchy is the account that is created by the service provider. The owner of that account is the owner of all resources that belongs to him. The next level in the hierarchy are the containers. Each account can contain many containers and each of these containers can also contain many objects. However, Objects are the atomic data structures in that hierarchy. So the namespace of those objects is defined by the accounts and containers. Specifically, the resource paths symbolize that structure again (see below). These paths are used to address accounts, containers and finally the objects.

/v1/{account}/{container}/{object}

To handle these structures the Object Storage API provides a set of functions for creating, modifying and retrieving containers, objects and metadata. This API is implemented through a RESTful HTTP interface. Due to this fact, it is also possible to use language-specific APIs for integrating it into a wide set of different programming languages. This API allows it to upload and store objects of any size, use cross-origin resource sharing to manage object security, compress files, override browser behavior for an object, schedule objects for deletion, bulk-delete up to 10,000 objects in a single request and auto-extract archive files. Another nice benefit is the object versioning that allows to store multiple versions of your content by using a kind of version controlling.

The default response format for every request is a text/plain format, but additionally it is also possible to make use of the JSON and XML data serialization response formats if you need it. The right to get access and change data via the Object Storage API is done by the authentication service. This can be the OpenStack Identity Service (Keystone), Tempauth or any other custom middleware. First you have to present your credentials to any of these authentication services, which returns a token and the storage URL for your account. Afterwards you can identify yourself to Object Storage by using this token. If the request is permitted, you have full access to make changes to it. If the token is not valid, you get no access to the account. Invalid tokens might occur when doing a request for another account or when the token is expired (defined by the authentication service). Another possibility to grant access to accounts is the usage of access control lists (ACLs). Here you can define all kinds of permissions for other users allowing them the access to your data.

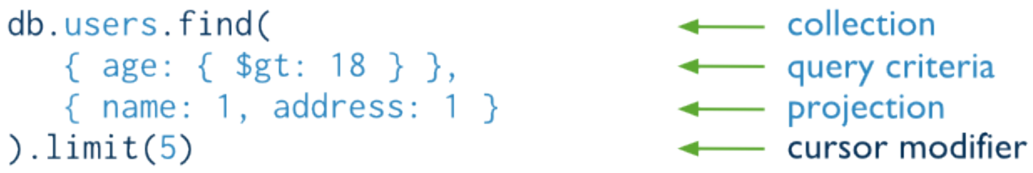
6. MongoDB GridFS Analysis

6.1. MongoDB

MongoDB [5] makes use of document-structured data and the relations between them. It stores data as documents in a binary representation called BSON (Binary JSON). That format extends the widely used JSON representation and includes additional types of data. Each document consists of one or more fields containing values of any data type. Instead of tables - analogous to relational databases - MongoDB makes use of collections. So the rows are the documents and the fields in that document are similar to columns. Furthermore the documents can vary in their structure and also the fields can vary from document to document. In case of adapting a specific document you do not care about other document structures in the system.

The document size is limited to 16MB. If you need to store files that are bigger, you can use GridFS. GridFS is a specification for storing and retrieving files that exceed this maximal BSON document size. The storing process of those files with GridFS is achieved by dividing it into several parts (chunks), where each part is stored in a separate document. The default size of these chunks is 255k. It uses two collections, on the one hand a collection for the file chunks (stores binary chunks) and on the other hand a collection for the file metadata. Both collections are placed in a common bucket by prefixing each with the bucket name. Each document in the chunk collection defines a distinct chunk and is identified by a unique ObjectId. After requesting such a file, the driver or client will reassemble it as needed. It is also possible to request data directly for specific parts of that files, for instance skip into the middle of a video or audio file. So it is also useful for files that do not exceed the maximal file size of 16MB, for example when you do not want to load the entire file into memory.
On the operational level, we can distinguish between two big groups: write and read operations.
A write operation is any operation for creating or modifying data in the database including insertions, updates and removals. Every write operation targets only a single collection and is atomic on the level of a single document. Read operations are responsible to retrieve data from the database. Read operations are done by using queries.

Queries give us much power in the term of data retrieving and manipulating. It is similar to SQL queries. It specifies criteria and conditions that identify requested documents. It can include a projection to limit the amount of returned data. Such a query operation is shown below.



```
db.users.find(
   { age: { $gt: 18 } },
   { name: 1, address: 1 }
).limit(5)
```

Figure 9 Components of MongoDB find operation

The efficient execution of queries is done by indexing. It uses a unique, compound index on the chunks collection that allows efficient retrieval of data. Obviously it is not necessary to scan every document that matches the query statement. Basically, indexes in MongoDB are similar to indexes in other database systems. Indexes are defined at the collection level and supports indexes on any field or sub-field of the documents.
For high redundancy and data availability purposes MongoDB supports replication. Therefore MongoDB instances are grouped into a replica set. It contains one primary and one or more secondaries (see figure 5). The primary receives all write operations and the secondaries apply these operations from the primary to execute the same requests to have the same data set. This is done by logging all the changes to its data sets at the primary. The secondaries replicate this log and apply the operations to their own data sets. If the primary is not available anymore, the replica set will elect a secondary as the new primary. Replica sets provide strict consistency.

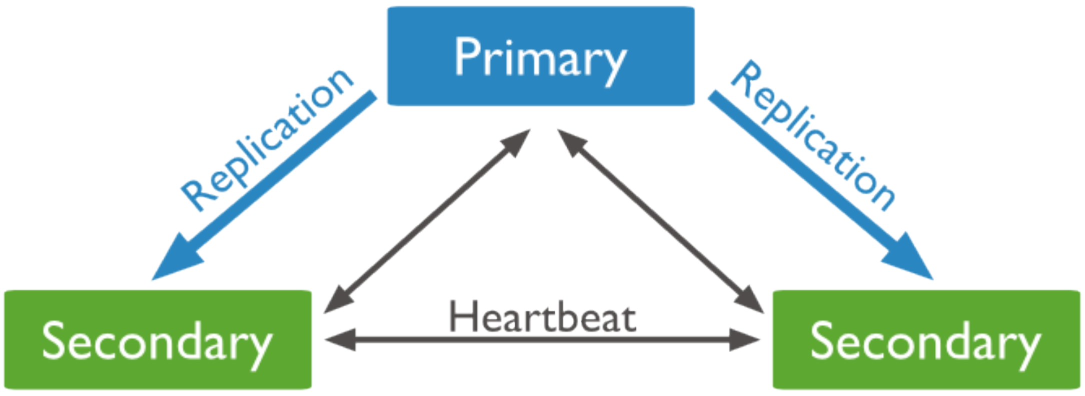


Figure 10 Replica set with a primary and two secondary

To improve the performance of the repository it is possible to make use of vertical scaling and sharding. Vertical scaling adds more CPU and storage resources to increase performance and capacity. Sharding (also called horizontal scaling) is a method for storing data across multiple machines to overcome large data sets and high throughput operations. Sharding means to divide the data set and distribute it over several servers (shards). With that we can reduce the number of operations and the amount of data that each shard has to handle. A sharded cluster architecture in MongoDB (see figure 10) consists of shards, query routers and config servers. The shards store the data. Each shard is a replica set to provide high availability and data consistency. The query routers process the requests and return the results to the clients. In this case we have two routers to divide the request load. The config servers store the cluster's metadata containing the mapping of the cluster's data set to the shards. Production sharded clusters have exactly 3 config servers.

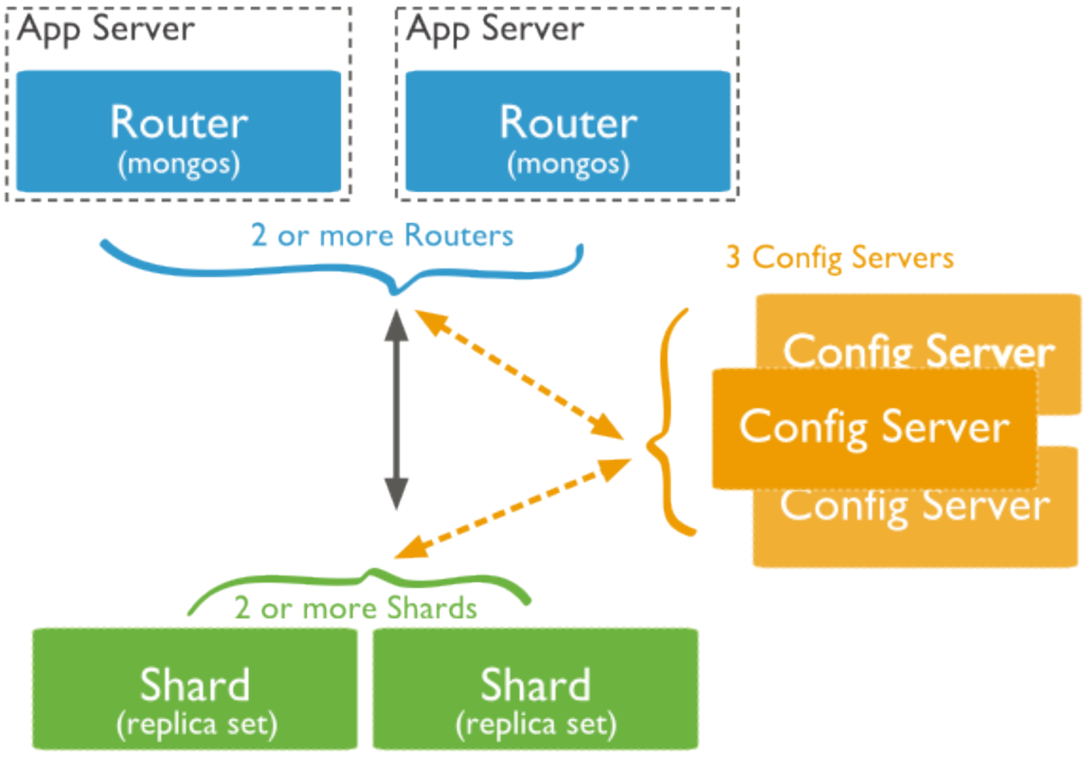


Figure 11 Shared cluster architecture

6.2. GridFS details

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default GridFS limits chunk size to 255k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.
When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file.
GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory.
To store and retrieve files using GridFS, use either of the following:

- A MongoDB driver. (Java, C, C++, C#, Python, Javascript, Ruby and so on)
- The mongofiles command-line tool in the mongo shell.

The mongofiles utility makes it possible to manipulate files stored in your MongoDB instance in GridFS objects from the command line. It is particularly useful as it provides an interface between objects stored in your file system and GridFS.
As previously said, GridFS stores files in two collections:

- chunks stores the binary chunks.
- files stores the file’s metadata.

Each document in the chunks collection represents a distinct chunk of a file as represented in the GridFS store. The following is a prototype document from the chunks collection:

{

"_id" : <ObjectId>,	The unique ObjectId of the chunk.
"files_id" : <ObjectId>,	The _id of the "parent" document, as specified in the files collection.
"n" : <num>,	The sequence number of the chunk. GridFS numbers all chunks, starting with 0.
"data" : <binary>	The chunk's payload as a BSON binary type.

}

Each document in the files collection represents a file in the GridFS store. Consider the following prototype of a document in the files collection:

{

"_id" : <ObjectId>,	The unique ID for this document. The _id is of the data type you chose for the original document. The default type for MongoDB documents is BSON ObjectId.
"length" : <num>,	The size of the document in bytes.
"chunkSize" : <num>,	The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 255 kilobytes.
"uploadDate" : <timestamp>,	The date the document was first stored by GridFS. This value has the Date type.
"md5" : <hash>,	An MD5 hash returned by the filemd5 command. This value has the String type.
"filename" : <string>,	Optional. A human-readable name for the document
"contentType" : <string>,	Optional. A valid MIME type for the document.
"aliases" : <string array>,	Optional. An array of alias strings.
"metadata" : <dataObject>,	Optional. Any additional information you want to store.

```
}
```

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by fs bucket:

- fs.files
- fs.chunks

You can choose a different bucket name than fs, and create multiple buckets in a single database.
Each document in the chunks collection represents a distinct chunk of a file as represented in the GridFS store. Each chunk is identified by its unique ObjectId stored in its _id field.
GridFS uses a unique, compound index on the chunks collection for the files_id and n fields. The files_id field contains the _id of the chunk's "parent" document. The n field contains the sequence number of the chunk. GridFS numbers all chunks, starting with 0.

7. Final decision: MongoDB

MongoDB already meets the requirements described in the previous section. And it was chosen accordingly with WP4.

7.1. MongoDB offers CRUD operations

7.1.1. Query Interface

db.collection.find() is equivalent to a SELECT * FROM <DATABASE>. It is possible to add criteria, for instance

```
db.users.find(
        {
                age: {
                      $gt: 18
                }
        }, {
                name: 1,
                address: 1
        }
).limit(5)
```

will return only the name and address of only 5 objects in the collection users with the field age greater that 18.

7.1.2. Insert

db.collection.insert() is equivalent to INSERT INTO <DATABASE>.
For example:

```
db.users.insert(
  {
    name: "sue",
    age: 26,
    status: "A"
  }
)
```

will insert a document in the collection users with the specified fields. It is possible to update an existing document running the command:

```
db.users.update(
  { age: { $gt: 18 } },
  { $set: { status: "A" } },
  { multi: true }
)
```

This command will set the status to “A” to all the users with age greater than 18.

7.1.3. Remove

The command **db.collection.remove()** will remove from the collection the matching documents. For instance:

```
db.users.remove(
  { status: "D" }
)
```

will remove all the users with status set to “D”.

7.2. MongoDB Driver

The easiest way to communicate with MongoDB is using the provided Drivers. MongoDB provides different Drivers for most programming languages:

- C
- C++
- C#
- Java
- Python
- …

In Java for example, after having downloaded and included the appropriate Driver, is possible to create a MongoClient like this:

```
MongoCredential credential =
     MongoCredential
          .createMongoCRCredential(
               userName,
               database,
               password
          );
MongoClient mongoClient =
   New MongoClient(
     new ServerAddress(“127.0.0.1”, 27018),
     Arrays.asList(credential)
   );
```

and then retrieving the Database with:

```
DB db = mongoClient.getDB( "mydb" );
```

On the db object, all the CRUD operations are available.

7.3. Authentication

In MongoDB an authentication system is already present. It is possible to define Users in order to access to the mongo server through username and password. To add a new user this command has to be run:

```
use admin
```

This will set the collection admin as collection in use. Then create the User as

```
db.createUser(
    {
      user: "superuser",
      pwd: "12345678",
      roles: [ "root" ]
    }
)
```

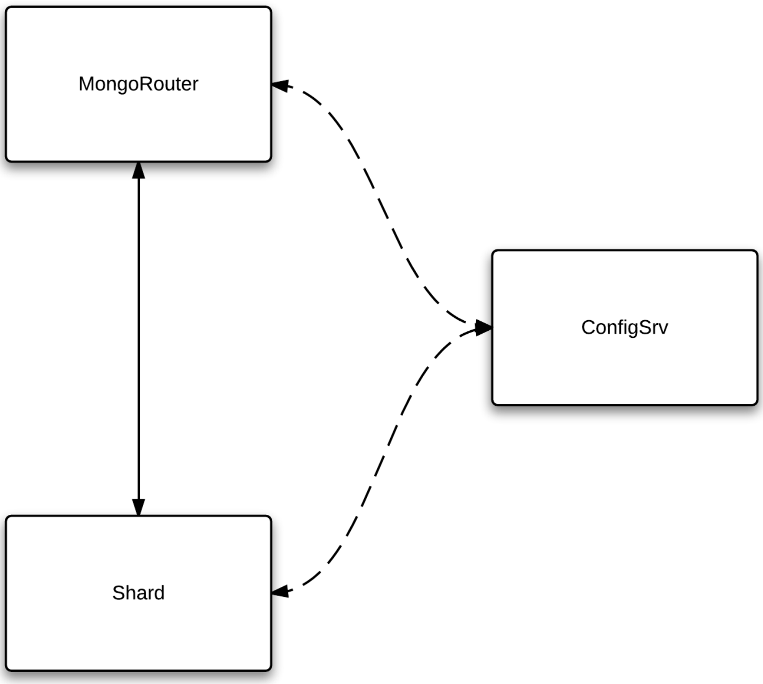
The new user “superuser” has the role “root”. This means that is has unrestricted access to all the collection in the System. It is possible to create custom roles:

```
db.createRole(
    {
      role: "manageOpRole",
      privileges: [
      { resource: {
            cluster: true
      }, actions: [ "killop", "inprog" ]
      },
      { resource: {
                  db: "",
                  collection: ""
      }, actions: [ "killCursors" ]
      }
      ],
      roles: []
    }
```

This role, for example, will grant the privilege of killing any operations.
There are different type of Authentications available, for more details on it please see [6].
In this way MongoDB offer a good authentication protocol already out of the box.

8. MongoDB Cluster

A MongoDB cluster is present inside the Virtual Infrastructure. At the moment it has restricted functionalities, but it can be easily improved. The actual configuration consists in three VMs:



The Shard VM is the real place where the Media Data is stored. It is necessary to add more Shards in order to have a real distribution of the data and some of the Shard can also be configured as a Replica set in order to provide High Availability. In the current version there is only one Config Server but for production environments at least three are necessary. This consideration has to be applied also to the Mongo Router. At least 2 are necessary in a production environment.

8.1. Installation

The environment where MongoDB were installed is composed by three VMs running Ubuntu 14.04.01 with 2GB RAM, 1 VCPU and 6.0GB of Disk space.
On each of them the main installation of MongoDB has to be executed:
Adding the key:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv 7F0CEB10
```

Adding one entry in the source list for MongoDB:

```
echo'deb http://downloads-distro.mongodb.org/repo/ubuntu-
upstart dist 10gen' | sudo tee
/etc/apt/sources.list.d/mongodb.list
```

Reload the packages list:

```
sudo apt-get update
```

And install MongoDB:

```
sudo apt-get install -y mongodb-org
```

This will automatically start the mongod service and it has to be done on all the VMs in order to provide all the tools necessary.
The configuration file location is by default **/etc/mongod.conf** and there can be set different value for configuration parameters, like different ports. In our installation all the default values are used.
In order to stop the mongod service, this command has to be run:

```
sudo service mongod stop
```

After having installed MongoDB on all the VMs it is needed to initialize the Config Server. So on the Config Server VM this command has to be run:

```
mongod --configsvr --port 27019
```

this command will run a process mongod configured as config server and listening on the port 27019.
In case of multiple Config Servers, this action has to be repeated on all of them.
Later, the Router Server has to be configured, stopping the mongod service and running a mongos instance:

```
sudo service mongod stop
mongos --configdb <ip_of_configsrv>:27019
```

Also in this case, when more Router Servers are involved in the environment, this command as to be run in all of the Router servers.
The last step is to add one or more Shard to the system. This involves two steps: first add the shard to the Router server then enable the Shard in the Config Server.
So run the mongo command to access the Router server:

```
mongo --host <ip_of_the_routersrv> --port 27017
```

Than add the Shard with:

```
sh.addShard( "<ip_of_the_shardsrv>:27017" )
```

Repeat the last operation for each shard that has to be added.
Then run the mongo command in order to access the Config server:

```
mongo --host <ip_of_the_configsrv> --port 27019
```

and now it is necessary to enable the sharding on a collection, for example:

```
sh.enableSharding("<name_of_the_collection>")
```

Finally check if everything went well:

```
> use config
> db.databases.find()

>{ "_id" : "admin", "partitioned" : false, "primary" :
"config" }

>{ "_id" : "test_db", "partitioned" : true, "primary" :
"shard0000" }
```

Than, as an example, it is possible to “shard” the collection on the _id:

```
> use test_db
> db.test_collection.ensureIndex( { _id : "hashed" } )

> sh.shardCollection("test_db.test_collection", { "_id":
"hashed" } )
```

And insert the data in the sharded collection:

```
> use test_db
> for (var i = 1; i <= 500; i++)
     db.test_collection.insert( { x : i } )
```

To have information regarding the state of the system:

```
 > sh.status()

 --- Sharding Status ---
     sharding version: {
           "_id" : 1,
           "version" : 3,
           "minCompatibleVersion" : 3,
           "currentVersion" : 4,
           "clusterId":
                ObjectId("529cae0691365bef9308cd75")
      }

      shards:
      {
           "_id" : "shard0000",
           "host" : "<ip_of_shard>:27017"
      }
      {
           "_id" : "shard0001",
           "host" : "<ip_of_shard>:27017"
      }
      . . .
```

9. Future works

The following phase that regards the Cloud Repository foresees the integration with the Virtual Infrastructure. This will turn the Cloud Repository from a static deployment to a dynamic one. This will be possible through the Elastic Media Manager. In the deployment phase the Elastic Media Manager will include the creation of the necessary VMs on which the Cloud Repository will be installed. This step will include the creation of new images that already contain the code of MongoDB and the Elastic Media Manager will trigger the installation.
Another goal that will be achieved is the improvement of the number of Shards, Routers and Config Servers, in order to create a real production environment. At least three Config Servers will be needed and two Router Servers. The number of Shard Servers will be reconsidered during the next release accordingly with the NUBOMEDIA applications developers and the application requirements. Obviously also the disk space of the actual VMs has to be improved.
It is necessary to create a general interface to the Cloud Repository in order to provide generic API that can be used in multiple and heterogeneous contexts. This will improve the reusability and the generalization of the Cloud Repository.

10. References

[1] Ceph. THE FUTURE OF STORAGE™. See http://ceph.com/.
[2] GlusterFS, Gluster. See http://www.gluster.org/
[3] MogileFS. See https://mogilefs.pbworks.com/w/page/13765356/FrontPage
[4] OpenStack Swift. OpenStack Swift. See swift.openstack.org/
[5] MongoDB, an open-source document database, and the leading NoSQL database See http://www.mongodb.org/.
[6] MongoDB Authentication possibilities. See http://docs.mongodb.org/manual/core/authentication/.
[7] FUSE. Filesystem in Userspace. a fully functional filesystem in a userspace program. See http://fuse.sourceforge.net/.