Core solution description

October 2016

Table of Contents

Introduction

OpenIO is an open source object storage solution introduced in 2015 though its development started 10 years ago. It is built on top of the groundbreaking concept of “Conscience” allowing dynamic behavior and human-free administration.

Ever wondered how to store multiple petabytes of data without getting overwhelmed by complex storage tasks and difficult application development? As object storage is changing the game in enterprise-class storage, most existing solutions may suffer lack of scalability over the years, with evolving hardware and capacity needs that increase dramatically. OpenIO’s Conscience technology deals with this, new hardware is auto-discovered and used right-away at no performance penalty. Heterogeneity of hardware is the norm, the Conscience is aware of the performance of each piece of equipment and distributes the load accordingly. OpenIO solves your storage problem, from 1TB to Exabytes, for the long term.

OpenIO is an already proven technology at massive scale, especially in email storage, with 10+ PetaBytes managed.

Our story

It all began in 2006. Data, in the datacenter of the company we were working for, was exploding at exponential pace and traditional answers with NAS or SAN storage were overwhelming for the teams who ran the platforms.

This is how the idea of an Object Storage solution came up. As most of the data we had was made of immutable objects, it was possible to switch to web-like GET/PUT/DELETE paradigm instead of the complex POSIX Filesystem API. It was also possible to get rid of the complex hardware required to run POSIX-like filesystems in a distributed manner.

Some Object Storage solutions already existed at the time but most of them were designed for the sole purpose of storing relatively few gigantic files, addressing large storage capacities with relatively small amount of metadata. Most of them were based on a single metadata node, avoiding the complexity of a distributed directory. Most of them were also made for non-production environments, and were not resilient enough, especially on the metadata side, with the single metadata node being a single point of failure (SPOF).

Our ambition was to store huge amounts of relatively small files produced by end-users like emails, eventually using a large storage capacity, but always accessed with the lowest latency. Also, there was the need for maximum availability as Service Level Agreements were stricts for these critical end-user services.

In late 2006, we designed our own solution with a massively distributed directory. It enabled a true scale-out design:

- each node would carry a part of the directory, i/o intensive metadata load would be distributed everywhere
- each new node was going to be immediately used, no need to re-dispatch already existing data to get benefits of the new hardware
- data storage was going to be de-correlated from metadata storage, giving true scalability on both axis

The solution had also to be hardware agnostic and support heterogeneous deployments. It had to support multiple hardware vendors and multiple storage technologies, from simple single drive x86 hosts to high-end NAS and SAN arrays that were going to be re-used behind x86 servers.

The first production ready version was built in 2008 and the first massive production of a large scale email system started the year after. Since then, the solution has been used to store 10+ Petabytes, 10+ billions of objects, at 20 Gbps of bandwidth at the peak hour, with low-latency SLAs enforced 24/7.

As the solution was designed as a general purpose object storage solution, it was soon used in multiple environments like email, consumer cloud, archiving (speed control cameras, ...), healthcare systems, voice call recordings, etc.

This Object Storage solution became Open Source in 2012.

Today, OpenIO has the mission to support the open source community, and to make the solution widely available especially for the most demanding use cases.

Technical facts

1. Core architecture

Global single namespace for 100s of petabytes / exabytes

The way OpenIO is designed makes possible to use a unique namespace to store 1000s of PetaBytes. In fact, our distributed directory is a tree-like structure made of two indirection tables. Data volume being 1TB, 100s of TB or 100s of PB, it will remain 3 hops to get down to the object location. In the contrary of many other designs, data query path does not depend on the number of nodes, and performance is not affected by the natural growth of the storage capacity.

Hardware agnostic

Our technology is adaptive. OpenIO is able to detect the efficiency of each node deployed in the infrastructure, and use it at its true capacity. The load can be balanced on heterogeneous nodes, OpenIO will take it into account and will get the best from each of them.

No SPOF architecture

Every single service used to serve data is redundant. From the directory's top level, to the chunk of data stored on disk, the information is duplicated. There is no SPOF (Single Point Of Failure), a node can be shut down, it will not affect the overall integrity or availability.

Scale-out architecture

With OpenIO, you can start at very low needs. You can even start with a "micro-cloud" single instance in a VM for testing purpose. But to get full protection for the data, it is recommended to start with 3 servers/VMs.

Scalability is easy and efficient, one can add one or hundreds of nodes, the process will remain the same. Anticipating long term usage to shape the platform is not mandatory. Just deploy as you grow, by small or huge increments, it is up to you.

Tiering

With storage tiering, you are able to configure classes of hardware, and select one of them when you need to store data. For example, you can create a pool of high performance disks (like SSDs), and configure it in a special class to store objects that require low latency accesses. This mechanism is called storage policies. Multiple storage policies can be defined in one particular namespace. They can be defined on hardware categories, but also on old or new objects, on number of replicas needed for a particular dataset, etc.

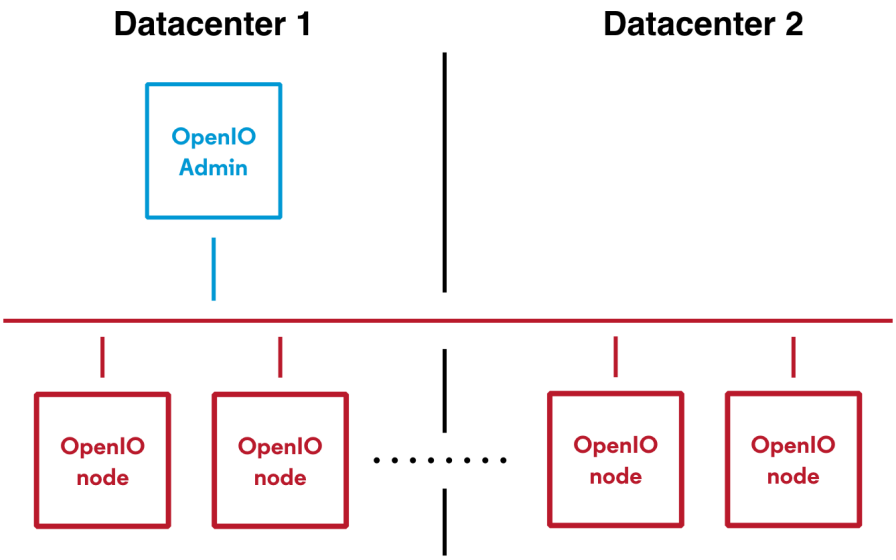
Replication and Erasure Coding

Configurable at each level of the architecture, directory replication secure the namespace integrity. Service directory and containers metadata can be synchronously replicated on other nodes.

To secure your data, data chunks can be replicated in various manners. From simple replication that just creates one copy of your data, to n-replicas for better fault tolerance, you are able to choose, with your own use case, between high security and storage efficiency. To take the best of both worlds, our erasure coding service will split the data in M chunks of data and K chunks of parity (M and K are configurable). Doing so, you prevent the total loss of your data by allowing the loss of part of it (up to K chunks of data) when a failure occurs, the K parity chunks will be used as recovery chunks for the lost data ones. Classic configuration can be 6+3 allowing the loss of 3 data chunks out of 6, but one can also configure it to 14+4 on larger platforms to allow minimizing the parity overhead with a very interesting 4:14 ratio, ~29%. It optimizes the cost of extra storage needed for data protection.

Geo-Redundancy

OpenIO SDS allows storage policies and data to be distributed across multiple datacenters. Depending on distance and latency requirements, data storage cluster can be stretched over multiple locations synchronously or replicated to a different site asynchronously.



2. Core concepts

Hierarchy and multi-tenancy

Multi-tenancy is a core concern of OpenIO's Software Defined Storage. Its "B2B2C" orientation is declined in 2 main organization levels: the account and the container.

More generally, data objects are stored within the following hierarchy: Namespace/Account/Container/Object.

There is no classic subdirectory tree. Objects are stored in a flat structure at the container level. As many other Object Storage solutions, one can emulate a filesystem tree but it has no physical reality.

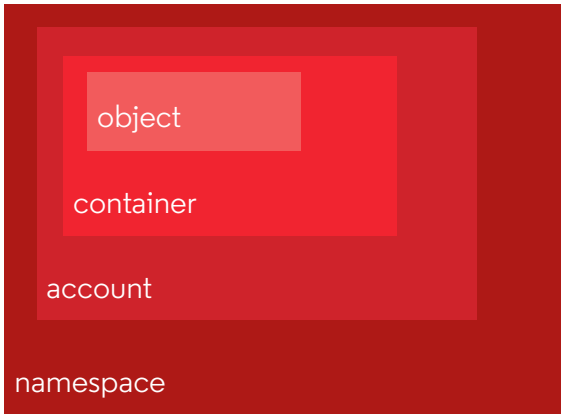


Fig.1 data organization

Namespace

The namespace gathers elements of a physical infrastructure. It hosts services and provides global features such as services configuration, platform monitoring.

Account

An account is the top level of our organisation. It is assimilated to a tenant, managing a central configuration, a global quota on disk space usage, and the billing features accompanying the quota. Each account owns and manages a collection of containers.

Container

A container belongs to an account. It is a collection of objects, and is identified by a name which is unique within the account. The container carries a bunch of options describing how to manage its objects: how to secure them (replicate or not), how many revisions are allowed for an object, etc.

Object

An object is a named BLOB with metadata. Several objects with the same name can be stored in a container, and are considered subsequent revisions. By default, PUT/GET/DELETE operations will manage the latest revision of an object.

An object can be arbitrarily large, but if its size is higher than a limit configured at the namespace level, it will be split into several chunks of data. Though it is recommended to avoid excessive chunking (to preserve a low latency to the data for most use cases), this principle allows to never face the case of a full device that would deny the upload of a new object.

Chunk

An object is not stored as-is but rather in chunks of data. A chunk is an immutable BLOB stored as a file on a classic filesystem. It also makes extensive use of filesystem extended attributes to store metadata such as object's size, position of the chunk inside the object, number of chunks, the path of the object it belongs to, etc. This allows rebuilding of a lost container. This feature can be assimilated to a reversed directory.

Chunks are of limited size, which is not of fixed size. A small object will use only one chunk. When an object size is more than a chunk's size limit, the object will be split into multiple chunks. This allows capacity optimization, as the solution always knows that a chunk will fit in the remaining space of a filesystem. It also allows for distributed reads of one object, which could be particularly useful for high-speed video streaming of large medias for instance.

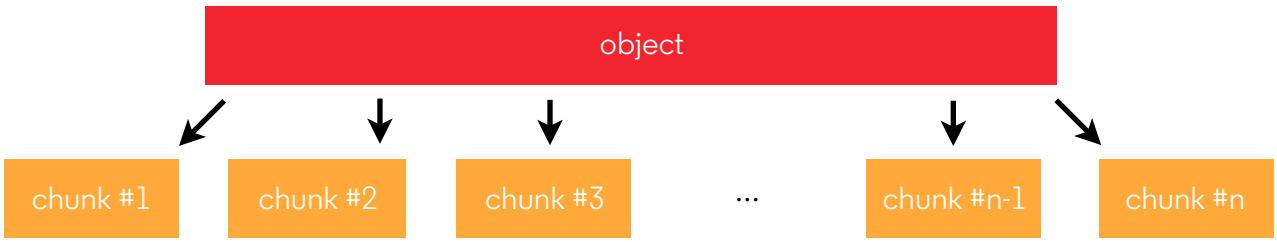


Fig.2 object split in chunks

3. Features

Items isolation

After years in production, one principle particularly helped: items isolation. Each container is stored in a separated file (i.e. they are not federated in one unique data structure), and each chunk is also stored as a file. This greatly improves the overall robustness of the solution, and limits the impact of a corruption or a loss of a single item.

Massively distributed 3-level directory

Container and objects are stored in a persistent 3-level distributed directory (Meta-0, Meta-1, Meta-2). OpenIO's allows to store hundreds of services for each of the hundred of millions of containers, with strong consistency and low latency, especially on read operations.

The directory has the form of a hash table mapping containers' UUID to their services. To handle the number of items, a first level of indirection splits the hash table into 64k slots. Every level is synchronously replicated.

Higher levels of the directory (the indirection tables) are quite stable, and get full benefits from implemented cache mechanisms. Caches are everywhere, especially inside directory gateways and are readily available on the client side of the solution.

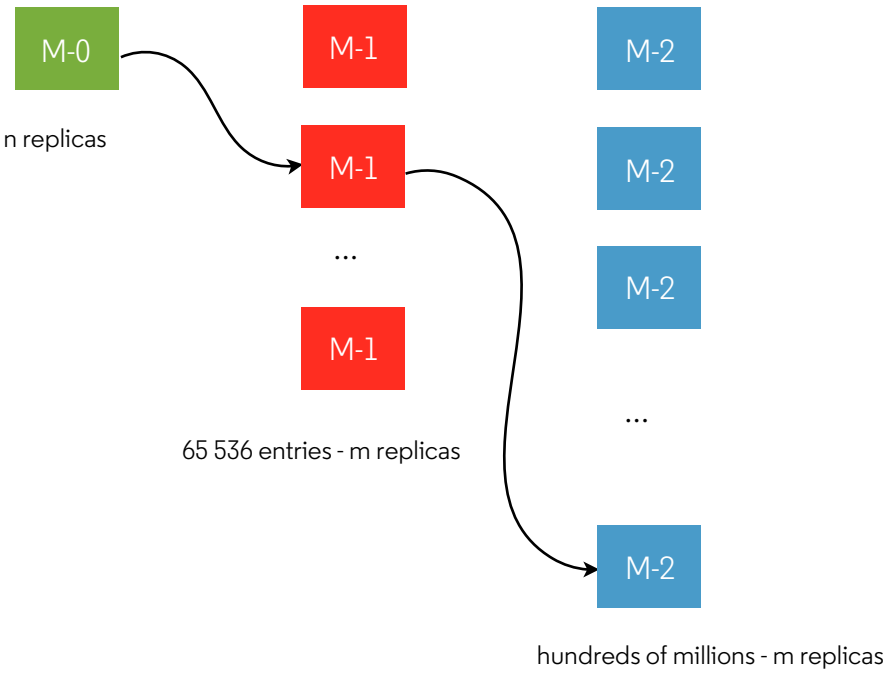


Fig.3 directory indirection tables

Conscience - advanced write load-balancing

For data placement, OpenIO SDS ships a system responsible for “best match making” between requests and services. It is called the “conscience”. The conscience takes into account constraints set by the request (e.g. respect of storage policies) and the quality of each service available. The quality is calculated from metrics coming from the nodes themselves, they are shared among the grid of nodes to the « conscience ». It is called the “score”. Through this feedback loop, each node knows in realtime what are the best nodes with the highest scores to handle a subsequent request.

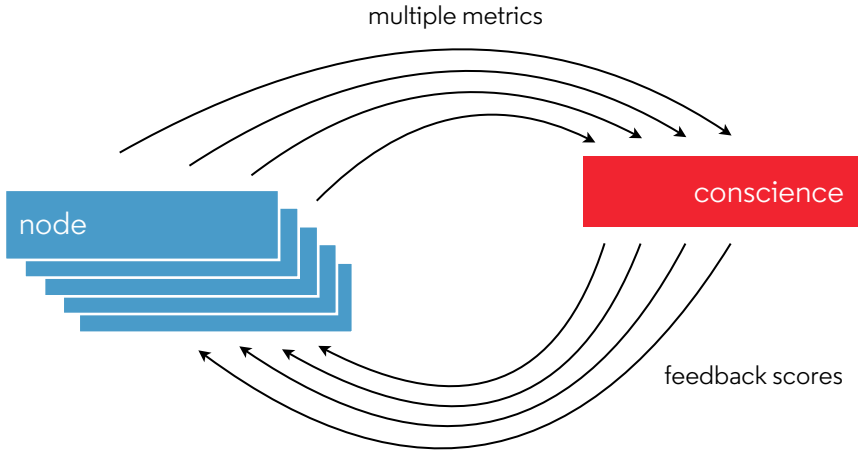


Fig.4 conscience feedback loop

Rebalancing

No consistent hashing is present in OpenIO SDS solutions, we rather play with directories. Thus becomes possible fine grained rebalancing on any distributed item instead of big slices of items at a time.

OpenIO SDS provides tools to automate chunks rebalancing. This is useful to rebalance the data at slow pace so that older data and newer data get always mixed together. As soon as the rebalancing starts, free space become eligible to load-balancing, thus enabling a maximal amount of disks axis ready to manage your I/O ops.

OpenIO SDS also provides tools to rebalance containers. With the same benefits as for data chunks: the newly freed container services become ready for new containers creations, enabling mix of old and new containers on the same hardware.

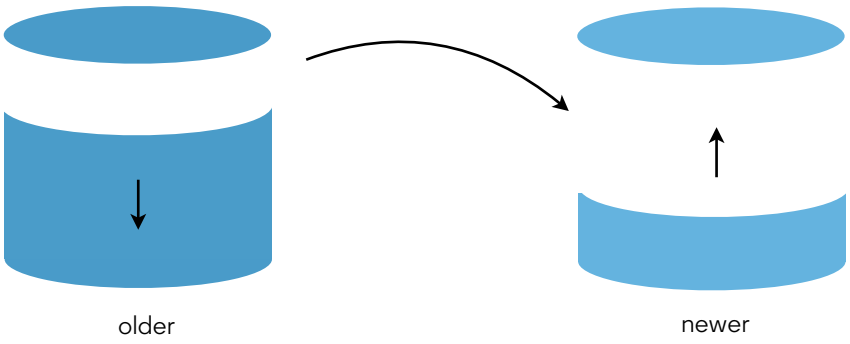


Fig.5 rebalancing from older to newer storage

Self-healing / Integrity loop

Each directory has a counterpart, reversed. *I.e.* each item knows its parents: a container is aware of its account, and a chunk is aware of the object and container it belongs to. That technique allows to rebuild a directory with a simple crawl of the items present on the storage nodes. Even if a container is lost, it is still possible to reconstruct it directly from the data. Usually, containers are replicated and rebuild occurs without the need for reversed directory crawl.

The storage nodes are also periodically crawled to trigger action on each item met. *E.g.* one can crawl a container and check the object's integrity and/or the chunks' accessibility. Another can crawl the chunks and checks whether its is well referenced in its container.

All those administratives tasks are accessible through REST APIs, so that on-demand actions are also easy to perform for maintenance purposes.

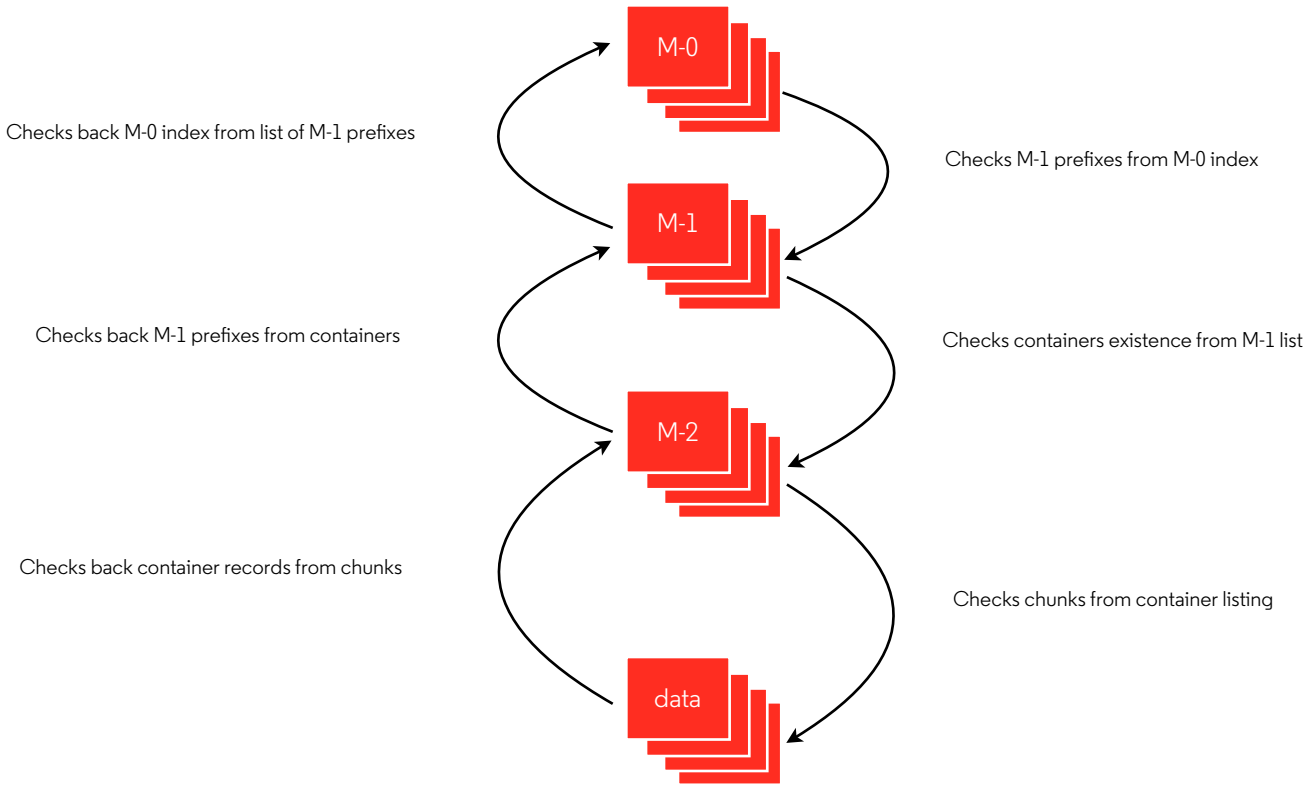


Fig.6 integrity loop checks for inconsistances

4. Data storage

Storage policies

Storage policies are OpenIO's way to manage storage tiering. They consist in triplets describing constraints set by the requestor: which storage class has to be used (the kind of device with its fallback. *E.g.* fast SSD, pretty good SAS drive, slow tape, etc.), if the data has to be secured (simple replication? Sophisticated Erasure encoding?), if the data has to be processed (compressed, encrypted, ...).

All the possible storage policies are namespace-wide, with a well-known default value. That default can be overloaded with each account, and furthermore on a per-container basis.

Versioning

A container can keep several versions of an object. This is configured at the container-level, for all the objects at once. The setting is set at the container's creation. It may be activated during the container's life. If no value is specified, the namespace default value is considered.

When versioning is disabled, pushing a new version of an object overrides the former version, and deleting an object marks it for removal. When versioning is enabled, pushing an object creates a new version of the object. Previous versions of an object can be listed and restored.

The semantics of objects versioning has been designed to be compliant with both Amazon S3 and Swift APIs.

Compression

Applied to chunks of data, this reduces overall storage cost. Decompression is made live during chunk download, with an acceptable extra latency and CPU usage. Compression is usually an asynchronous job not to hurt performance, and you can specify which data you want to compress (data ages, mime-types, users...).

Directory replication

Configurable at each level of the architecture, directory replication secure the namespace integrity. Service directory and containers metadata can be synchronously replicated on other nodes.

Data duplication

To secure your data, data chunks can be replicated in various manners. From simple replication that just makes one copy of your data, to n-replicas for better fault tolerance, you are able to choose, with your own use case, between high security and storage efficiency.

To take the best of both worlds, our erasure coding service (e.g. Cauchy-Reed-Solomon) will split the data in n chunks of data and k chunks of parity (n and k are configurable). In this way, you prevent your data accessibility from fault and optimize cost of extra storage needed by data securitization.

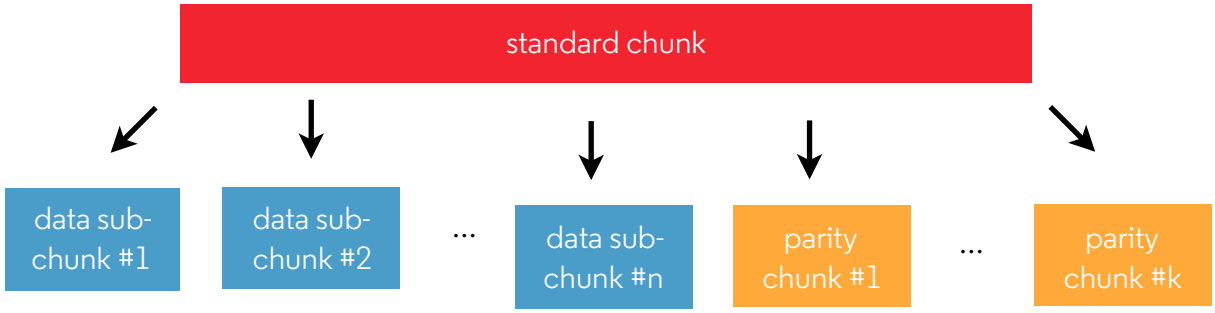


Fig.7 erasure coding split

5. Native APIs

Proxified directory

Whichever protocol is in use by the internal services (directories), all the clients rely on a layer of gateways that provides a simple REST API for metada accesses. This API provides high-level operations that encapsulates all the underlying logic behind the management of the accounts, containers and contents. These gateways are also an ideal place for shared cache mechanisms. Think of it as a Name Service Cache Daemon on Linux hosts.

Client sdks

With the benefit of the gateways, clients are never-so-easy to write. All make use of the gateway layer and have just to manage efficiently the data streams. The following implementations are currently available:

- C : Slick and lean, efficient
- Python : Pure pythonic API, no ugly binding is necessary
- Java : Pure Java API, to get access to the Software Defined Storage. POJO only, no need of JNI or any humongous framework.

These clients are considered “low-level”, since they take part in data placement too and are close to the remote services. Technically, they require access to the whole grid of nodes. In other words, they are a part of it. Another option is to deploy a REST gateway to access data from the outside, such as our implementation of Amazon® S3 or OpenStack® SWIFT.

Command line

A central tool wraps to the CLI the Python native API. Because the command requires the same network accesses to the grid, this means the “oio” command is meant to be ran on grid nodes or its allowed clients.

“oio *action* $[container_id]/$[objectid]”

6. Rest apis

S3 & Openstack Swift API

OpenIO SDS is compatible with the Amazon® S3 API and the OpenStack® Object Storage REST API. Application integration becomes easy with these standard REST protocols. Since these APIs play nicely with SDS, you have access to all advanced storage features of OpenIO SDS and you do not have to make major trade offs.

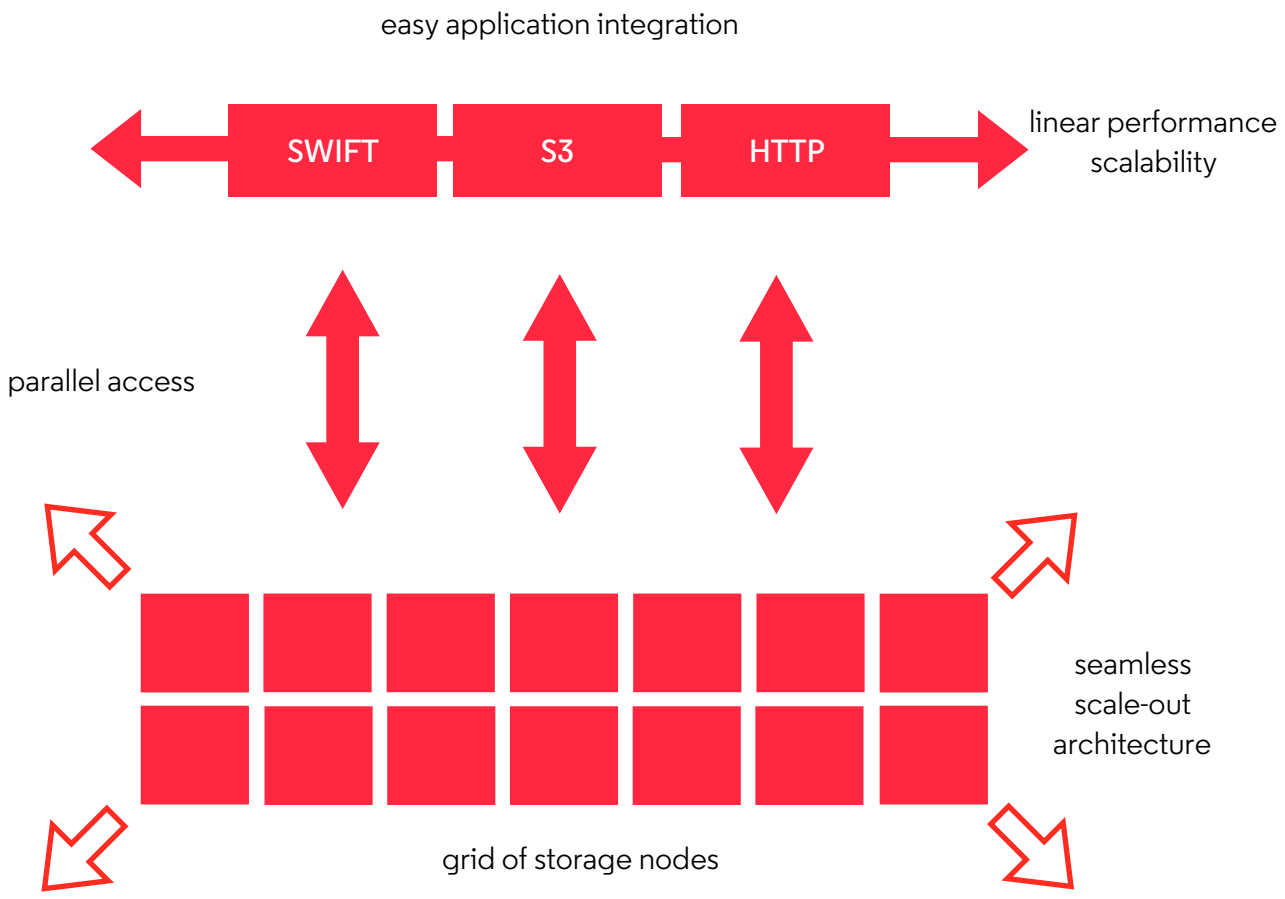


Fig.8 access gateways scalability

app-aware storage software

openio.io

FR
3 avenue Antoine Pinay
Parc d'Activité des 4 Vents
59510 Hem

US
180 Sansome Street, Fl4
San Francisco, 94104 CA

JP
1-35-2 Grains Bldg. #61
Nihonbashi-Kakigara-cho,
Chuo-ku, Tokyo , Japan 103-0014