Alba Architecture

July 17, 2015

1 Introduction

It's important to have an overview of what you're trying to build and why. This document is a good start.

2 Requirements

2.1 Functional requirements

2.1.1 store an object

We need a simple API to store an object that resides in memory or on disk at the client's site. It should work it's way around fluctuating availability of disks.

2.1.2 retrieve an object

We need to be able to retrieve an object into a buffer or file, even when there are failures (within operating range).

2.1.3 partial retrieval

We need an API to retrieve only a part of an object, in an efficient way. Moreover, the ability to specify the retrieval of several parts of an object or objects.

2.1.4 delete an object

Remove an object from the system so that users cannot retrieve it anymore, and that the resources used by the object will *eventually* become available again. A deleted object should not show up in subsequent object listings. It is fine if the object is still retrievable for someone having the metadata, for a short while after the object has been declared deleted.

2.1.5 namespaces

The ability to group objects together, and decide storage policy on the group level iso the individual level. So you want to specify desired redundancy, compressor, encryption options, aso per namespace iso per object.

2.1.6 list objects

We want to be able to retrieve a list of object that are in the same *namespace*. The listing should be authoritative, so if the object's information can be retrieved via the listing API, the object should be retrievable. For the listing itself, a paging based api is required.

2.1.7 list meta objects

like namespaces and policies, available storage devices, defaults, ...

2.1.8 retrieve run time statistics of components

like proxies, storage devices, namespace hosts, ...

2.1.9 snapshot of a namespace?

Cheap marking of the current state of a namespace so we can revisit this state later. This has been a request that keeps coming back, but no other object store is offering this, which means the volume driver needs to do the administration itself for all object stores. Alba could however facilitate additional opaque meta data for objects.

2.1.10 data safety.

For objects, we are interested in the number of disks that can fail before the integrity of the data is compromised. Objects should be dispersed with over disks with a certain number (k) data fragments, a certain number (m) of parity fragments, and a limit (x) on the number of disks from the same node that can be used for a specific object. An upload can be successful, even when not all ($k+m$) fragments were stored. There's a minimum number of fragments (c), with ($k \leq c \leq k+m$) that needs to have been stored, before an upload can be considered successful. The tuple (k, m, c, x) describes a *policy*. But more than one policy can be acceptable. The metadata should have the same policy, and not just a replication based number. We need to know which objects are at risk (policy lower than the desired one, or object impaired by a broken disk).

2.1.11 optional compression.

For some types of objects, compression helps a lot. So we should be able to enable a specific compressor (for example *snappy* or *bzip2*) for these objects.

2.1.12 optional encryption

For some namespaces, we need **plausible deniability** of content knowledge. This means that what needs to be stored needs to have been encrypted *on the client's tier*. This impacts key management and repair

2.1.13 optional privacy of communication

between client and storage system components. Typically *ssl*.[1]

2.1.14 total privacy

Some customers have the additional requirement that their data does not reside on shared disks. They have their own pre-allocated set of disks. This means a mapping from namespace to OSD (Object Storag Device).

2.2 Efficiency

For the type of objects we care about, we need to be better suited than Swift, Ceph, or DSS. In concreto, we want

lower storage cost erasure coding of data. we don't want data replication.

lower read latency No access point proxy.

higher read efficiency ie partial reads, and a light codec, that allows these partial reads without processing the full object.

high write throughput but we don't really care about write *latency*

[1]we'll do this in a later phase

2.2.1 to proxy or not to proxy?

A proxy that functions as an access point is comfortable, but adds latency. However the proxy's proximity to the client has tremendous impact.

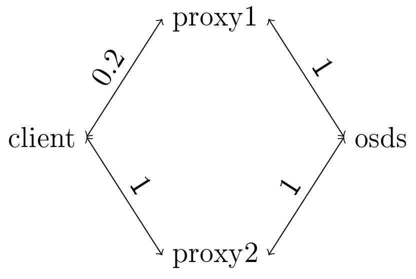


Figure 1: latency for proxies

Suppose, latency is 0.2 hops communication on localhost (rough estimate). For the first proxy in figure 1 we have a total latency of 2.4 hops. while for the second proxy, we incur a latency of 4 hops. The ideal would be of course 2 hops. This simple example shows that *hop count* dwarfs most other things in a distributed system, but that the latency of the hops matters too. If we really need to have a proxy, it better be on the same tier as the client.

2.2.2 partial read

Hop counts come close to the truth but you also want to be able to to read only what you want. It's kind of painful when you have to fetch and reconstruct a 4MB object when you are really only interested in 4KB of its contents. Partial reads allow you to fetch what you need.

2.2.3 Reliability

This means the system is self maintaining (up to a point). Automatically

- work around a failing disk.
- heal damaged objects
- know what to do when a disk is added or when a node is added.
- **adapt to changed policies?** (do we want this? for example, change number of parity disks m and apply this to already existing objects?)

2.3 Scalability

We aim to build a storage product for $[12, 100)$ HDDs. Within this range, it should scale rather smoothly. It's acceptable if the minimal number of disks that can be added at a time is somewhat larger than 1. Above and below this size the system's performance may degrade[2] If it doesn't then it's a fortunate accident.

2.3.1 Hardware

We design for little boxes of 4 HDDs that are deployed as a single unit called a **node**. The core idea is that we can work with a crippled unit when 1 of the disks is dead, but that we decommission the unit when 2 disks die. The decommissioned unit can than be taken out and replaced by a fresh one[3]. This also impacts the data safety policies we want to use. Perhaps surprisingly, small deployments are a big challenge. Consider 12 disks in 3 nodes. This means a policy of something like $(4, 2, 6, 2)$ will yield a rather disappointing performance, and ditto redundancy.

We should also be able to run on _Seagate's Kinetic drives$_{\text{[fn}}$::eventually].

2.4 Code Evolution

We cannot hope to hit on the perfect design immediately so we must be able to adapt. *New cryptors* will become fashionable (or necessary) as will *new compressors* or encoding schemes. We will change our minds about suitable *data formats* etc. We must be able to change our preferences for new objects without having to rewrite existing data.

2.5 Deployment Comfort

We want minimal configuration and simple upgrades. We have learned that a single configuration file that needs to be manually maintained across multiple hosts is already a challenge for operations. We desire boot-strappable configuration of clients from 1 working access point. It would be really nice if our own hardware nodes can be shipped with everything installed and can be used without any on site installation.

[2]we currently envision a fully connected architecture, with all proxies connected to all OSDs. That may not be scalable to bigger environments

[3]this is cheaper than catering for individual disk replacements

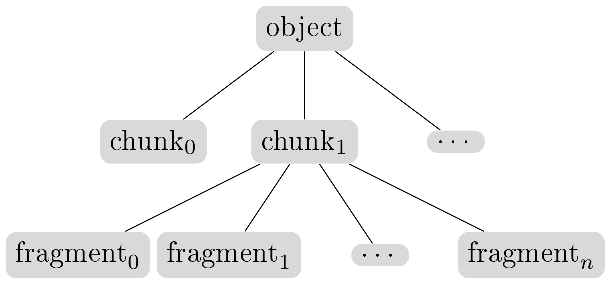


Figure 2: Object structure

2.6 Dashboard

Most storage systems are notoriously bad when it comes to the ability to inspect the current state. We need some relevant pseudo-live statistics, and presentation thereof. This needs is integrated into the OpenVStorage GUI.

3 Concepts

3.1 Object Structure

Alba works on objects, but what exactly is an object? An object is a largish amorphous piece of data for which you *know the size*. Objects can grow nor shrink. An objects's contents cannot be changed. Inside Alba, objects are represented as an *array of chunks*. A chunk is an *array of fragments* and has a nice[4] size. This means the last chunk might need padding. It also means you can retrieve individual chunks, and fragments.

3.2 Volume Driver SCOs

Alba is designed to be a store optimized for Volume driver SCOs. A SCO[5] by definition already has a nice size, so a SCO will be a an object of 1 chunk. Currently that is 4MB, but we need to experiment a but what works best.

3.3 Fragment Placement

Alba transforms chunks into an array of (maybe compressed and encrypted) fragments. The transformation will turn k data fragments into $n = k + m$ coded fragments. These n fragments need to be stored on our nodes and

[4] well aligned with buffer sizes, etc
[5] superblock container object(!?)

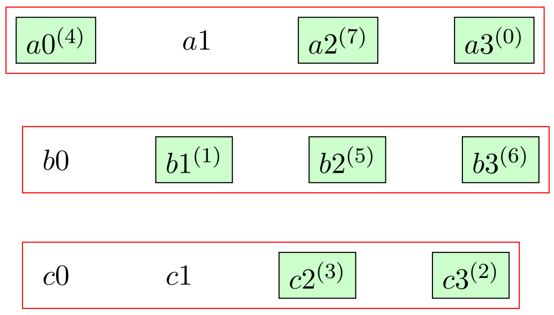


Figure 3: example fragment placement. (Superscripts show the order in which disks were chosen)

will end up on HDDs. Obviously, putting all fragments on a single disk is sub optimal as the objects life is that of this single HDD. We assume nodes to fail independently. So, ideally you want to use as many nodes as there are fragments n. If for some reason, there are less nodes available it's still advantageous to spread as wide as possible. When storing or repairing a chunk, an Alba client will use the hardware layout structure to do a *best effort* in finding disks for fragments. This calculation is done *on-the-fly* and can use available information. A disk that's known to be dead will not be picked. As you can see from a simple example in figure 3, where we show a choice for 8 fragments over 3 nodes (a,b,c) with 4 disks each, perfect balance is not always achieved. Our placement algorithm will pick randomly, while being as balanced as possible. The algorithm is generic so adding levels (racks, ...) is possible.

Alba uses fill rates of the devices as well, so that fuller OSDs have a slightly lower probability to be picked than emptier ones.

A special extra constraint is the maximum number of disks that can be picked from the same node. This is a simple and elegant way to pay attention to the fact nodes can fail (or be decommissioned) too.

3.4 OSDs

3.4.1 Seagate's Kinetic Drives

Seagate offers a hardware component that does almost exactly what we need. It's more or less[6] a TCP/IP socket server with *leveldb* behind it, combined in one package: HDD + ARM CPU + ethernet port. Keys are limited to

[6]The simulator is *exactly* that

4KB, and values are limited to 1MB. The best througput is achieved for the largest values. They provide *atomic multi-updates*, and a `getRange` call. The drives advertise their availability through a UDP multicast of their json config.

3.4.2 ASD

When the Kinetic drives are not available, we use our own pure software based implementation of the same concept: A Socket Server on top of RocksDb, offering persistence for meta data and small values, while large values reside on the file system. Every atomic multi-update is synced before the acknowledgement is returned. To have acceptable performance, the updates for multiple clients are batched and at the end of the batch, the ASD calls `syncfs` to *sync* the whole filesystem (files, directories, and rocksdb) at once. This is effectively a tradeof, increasing latency for throughput, which is perfectly acceptable for our use case.

ASD's follow the same strategy as the Kinetic drives, advertising themselves via a UDP multicast of a json config file. A single parser can be used to retrieve connection information of for ASD and Kinetic drive alike.

3.4.3 Generic OSD

We've pulled a generic abstraction over ASD and Kinetic Drive, but this has downsides too. For example, ASDs have to be kept ignorant about object structure or reverse mappings. We cannot start running maintenance tasks on the ASDs for the same reason.

3.5 Inter Component Messaging

Alba is a distributed system, and information needs to flow between components, but most of the components are passive: They will not initiate an RPC call or set up a connection. So some components need to act as a facilitator. This task is performed by the proxies and maintenance agents, often as part of an admin call. An example illustrates what's going on.

3.5.1 Creating a Namespace.

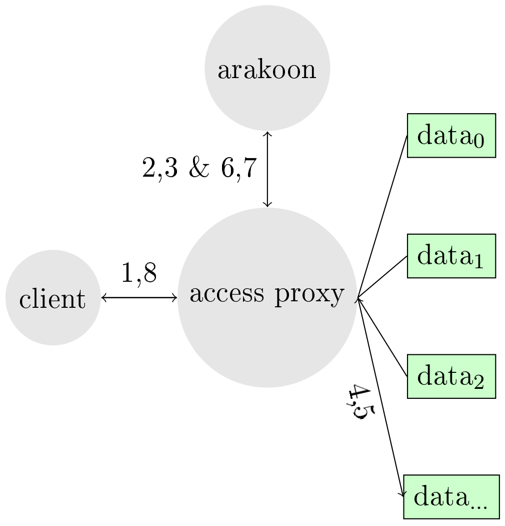
Creating a namespace first adds the information to the alba manager. The alba manager records which namespace host will keep the metadata of that namespace. Also, the OSDs to be used for that namespace need some setup information. However, the alba manager does not contact the namespace manager directly. Instead, the client doing the `create_namespace` call (or a proxy, or a maintenance agent) will pull messages from the alba manager for the namespace host and deliver them. Upon successful delivery, the delivering agent will mark these messages delivered on the alba manager. The same happens for the messages that need to go from alba manager to OSDs: The client pulls them, delivers them and marks them as delivered after successful delivery. Of course, all kinds of things can go wrong, like double delivery, races and networking errors.

4 Primary Operations

4.1 Performance

So why can have a solution with lower latencies than other object storage solutions?

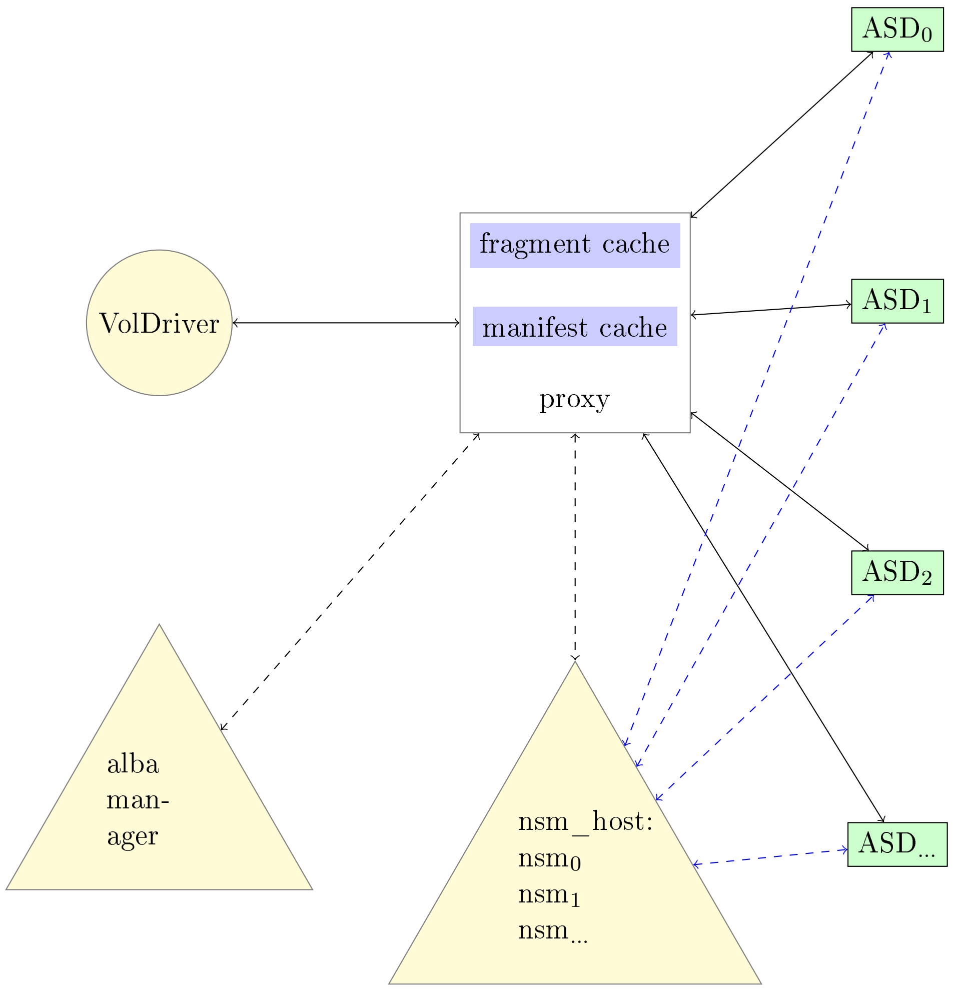
4.1.1 DSS (Simplified)



The figure shows a simplified schema of DSS. The proxy is inserted to decouple the front end interface architecture from the back end.

Also it makes integration easy as they can talk 1 (proprietary) protocol on the back end, and multiple protocols (S3,REST, ftp, ...) on the front end side of the proxy. The metadata is stored in an Arakoon fork. The retrieval of an object causes a metadata lookup in Arakoon, then fetching the data, reconstructing the object and sending it to the client. All in all 6 hops. However, it's quite cheap to cache the metadata, so if your storage system isn't too big, it might be 4 hops. (Writing with DSS even incurs 8 hops, as there is a metadata update before and after the write of the data.)

Alba tried to suppress the proxy entirely and to provide a c++ client lib, but that turned out to be too difficult, and currently operates as a proxy on the client's tier. So Alba is a proxy service, residing on the client's tier. This



proxy bootstraps itself from an alba manager.

$$\begin{tikzcd}
 & & \text{compress} \arrow[r] & \text{encrypt} \arrow[r] & \text{coords} \arrow[r] & \text{storage} \\
\text{chunking} \arrow[r] & \text{encoder} \arrow[r] \arrow[ru] \arrow[rd] & \text{compress} \arrow[r] & \text{encrypt} \arrow[r] & \text{coords} \arrow[r] & \text{storage} \\
 & & \text{compress} \arrow[r] & \text{encrypt} \arrow[r] & \text{coords} \arrow[r] & \text{storage}
\end{tikzcd}$$

4.2 Put

The flow is as follows. Objects are split in chunks sequentially[7]. Each chunk is encoded into fragments. The first k fragments comprise the original chunk, while the codec generates m extra fragments that can be used in case of erasures.

We may want to compress and encrypt fragments. After that, coordinates are assigned and the packed fragment is stored. All fragments are handled independently of one another, so this can happen in parallel. At the end of a put, Alba also writes a **manifest** that contains the tree information, as well as the choices made when the object was written (k, m, cryptor,. . .).

Manifest are stored in the namespace manager for that volume.

```
void write_object_fs(const string &namespace_,
                     const string &object_name,
                     const string &input_file,
                     const allow_overwrite,
                     const Checksum *checksum);
```

All the information needed to process the object is derived from the namespace's **preset**. A preset contains:

- a list of acceptable policies (k, m, c, x)
- choice of compressor
- choice of encryption
- choice of object checksum algorithm
- choice of fragment checksum algorithm
- acceptable OSDs
- . . .

[7]remember, for Volumedriver there will be only 1 chunk

The list of acceptable policies allows Alba to decide which policy to use, taking the current state of the system into account. For example, The first policy in the list might be very wide (large $k + m$, low x) and currently unavailable due to a node that is inavailable. Alba will then move to the next policy on the list. Repair will eventually try to move the object from the current policy to the best policy.

4.3 Get

Get works the other way around: First retrieve the manifest. We hope it's in the manifest cache, but if not, we retrieve it from the namespace host (nsm). The manifest contains everything we need to retrieve the rest.

Retrieval of the fragments, decryption and decompression is done in parallel. The resulting fragments are fed into the decoder, yielding a chunk. We don't want to retrieve all fragments, as chances are we don't need them, but if we don't fetch enough, we incur an extra round trip. If we only want an extra roundtrip in 1% of the reads, and have an idea of network and disk failure rates, We gamble a bit, and ask to retrieve $k + \epsilon$ fragments, hoping we get at least k. [8]

```
void read_object_fs(const string &namespace_,
                    const string &object_name,
                    const string &dest_file,
                    const consistent_read,
                    const should_cache);
```

The API call for the C++ proxy client has some additional cache instructions. The `consistent_read` parameter tells Alba whether to use the manifest cache for lookups, and the `should_cache` parameter tells Alba if the manifest should be added to the proxy's manifest cache afterwards.

4.4 Partial Read

Erasure coding typically works with an encoding matrix that transforms the data. suppose the data is $\vec{x} = [x_1 \ldots x_k]$ the transformation is A and the

[8] we don't do this yet, and voldriver never retrieves full objects as it prefers partial reads

stored data is $\vec{b} = [b_1 \ldots b_{k+m}]$.

$$\left[\begin{array}{cccc} a_{1,1} & a_{1,2} & \dots & a_{1,k} \\ a_{2,1} & \dots & & a_{2,k} \\ \dots & \dots & \dots & \dots \\ a_{k,1} & \dots & & a_{k,k} \\ \hline \\ a_{k+1,1} & \dots & \dots & a_{k+1,k} \\ \dots & & & \dots \\ a_{k+m,1} & & & \end{array}\right] \cdot \left[\begin{array}{c} x_1 \\ x_2 \\ \dots \\ x_k \end{array}\right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \dots \\ b_k \\ \hline \\ b_{k+1} \\ \dots \\ b_{k+m} \end{array}\right]$$

The matrix A is designed so that all subsystems with n equations are solvable. If we add the constraint that $A_{11..kk}$ is I then \$b$_1$ = x$_1$, b$_2$ = x$_2$, …\$ Alba uses this scheme. Partial object retrieval is also the reason Alba first encodes, and only then compresses and encrypts.

The alternative would be to first compress, then encrypt and only then encode, but since compression is not very predictable, it's too difficult to calculate where the piece of data you need ended up.

The current scheme, however makes it very easy to retrieve a part of the object. Sometimes, the slice borders don't align well with the fragment borders, and we need to fetch multiple fragments for one slice. The api currently offers a call that allows you to specify multiple slices from multiple objects:

```
struct SliceDescriptor {
  byte *buf;
  const uint64_t offset;
  const uint32_t size;
};

struct ObjectSlices {
  const string &object_name;
  const vector<SliceDescriptor> slices;
};

void read_objects_slices(const string &namespace_,
                         const vector<ObjectSlices> &,
                         const consistent_read);
```

4.4.1 Fragment cache

Each proxy has a local, on disk fragment cache. When fragments are retrieved from the OSDs they are normally added to this cache. Further (partial) reads might need them too. The fragments are in *packed* state (possibly compressed and encrypted). The cache currently uses a simple LRU victim selection algorithm.

4.5 Replication

Replication can be seen as a special case of erasure coding. With a replication factor r we have $k = 1, m = r - 1$. However Without special casing the code, this would be rather inefficient as the same chunk, would be compressed and encrypted r times. The resulting 'fragments' could be different as the encryption's initialization vectors might differ. However, it's not difficult to make the replication case efficient. In Alba, a policy where $k = 1$ effectively means **replication**.

4.6 Delete

In Alba, delete is a synchronous metadata operation, combined with a detached maintenance action. So it is possible that another proxy, whose manifest cache still contains the necessary information is still able to retrieve the object.

```
void delete_object(const string &namespace_,
                   const string &object_name,
                   const may_not_exist);
```

4.7 Namespaces

a namespace is encoded as an extra part of the name (before hashing).

4.8 List objects in a namespace

Alba provides an API call to alphabetically list the objects in a namespace.

```
tuple<vector<string>, has_more>
list_objects(const string &namespace_,
             const string &first, const include_first,
             const optional<string> &last,
             const include_last, const int max,
             const reverse reverse = reverse::F);
```

Essentially, this is a range query on the namespace host. As with all range queries on Alba metadata, the caller can say how many items (s)he wants to process as a result, but the callee ultimately decides how many items it returns and if there are more to come.

4.9 Configuration

Alba storage devices (and Kinetic drives) announce themselves via an UDP multicast. They need to be claimed by an Alba instance, before they can be used. When a device is claimed, the alba mangager knows the coordinates of the device. The alba manager also knows which namespace host is responsible for a certain namespace. A proxy just needs coordinates of the alba manager to be able to bootstrap everything.

4.10 Evolution of code and Architecture

The strategy is to make all choices explicit and record the choices in everything stored. (The trick here is not to forget anything)

4.11 Adding a disk

4.12 Dead disk, now what?

4.13 Total Privacy

4.14 Maintenance

There are various tasks:

- clean obsolete fragments
- garbage collection
- repair OSDs
- repair objects
- rebalancing

4.14.1 Clean Obsolete Fragments

When objects are deleted, or when namespaces are deleted, obsolete fragments are cleaned up out of band, by the maintenance agent. The namespace manager knows for each OSD which key value pairs need to be deleted. So the maintenance agent fetches for a particular namespace, a batch of key value pairs to be deleted, creates a big delete sequence, and applies this on the OSD. Afterwards, it *marks the keys deleted* on the namespace manager. The granularity is per namespace and per OSD.

4.14.2 TODO Garbage collection

When things go wrong, for example a failed upload, there might be some fragments left behind. Garbage collection removes the superfluous fragments.

4.14.3 Repair Objects

If an object is missing fragments, for example due to a dead OSD, it needs to be repaired. A missing fragment can be regenerated (typically on another OSD), and the manifest can be updated. This is the simplest case, and repair will be *surgical* as the object's structure and most of its fragments remain as they were. Other cases exist: Namespace presets typically have multiple acceptable policies and the best policy might not be available during the time of the upload of an object. Objects that have been written using a lesser policy run a higher risk of data loss. Therefore, these will be rewritten by the maintenance process if a better policy becomes available. In this case, the object's structure changes and it will have to be rewritten completely.

4.14.4 Rebalancing

Alba slightly prefers emptier OSDs for fragments on new writes, and thus tries to achieve a good balance. This is not enough. For example, after replacing a defect drive in a well filled Alba, the new drive will be empty and way below the average fill rate. So the maintenance agents actively rebalance the drives.

Currently the strategy is this. The drives are categorized to be in one of three buckets: *low*, *ok*, *high*.

$$\begin{aligned}
\text{low} &= \{\text{osd}|\text{fr}_{osd} < \overline{fr} - \alpha\} \\
\text{high} &= \{\text{osd}|\text{fr}_{osd} \leqslant \overline{fr} + \alpha\} \\
\text{ok} &= \{\text{osd}|\overline{fr} - \alpha \leqslant \text{fr}_{osd} < \overline{fr} + \alpha\} \\
&\text{where} \\
&\alpha = \sigma_{fr} + 0.01
\end{aligned}$$

Then, a batch of random manifests are fetched. For each of the manifests, the rebalancer tries to find a move of the fragments on *osd* in *high* towards an *osd* from *low* (notation: $\{\text{high} \rightsquigarrow \text{ok}\}$). In absence of such a move, a less ambitious move $\{\text{high} \rightsquigarrow \text{ok}\}$ or $\{\text{ok} \rightsquigarrow \text{low}\}$ is proposed for that manifest. The batch is sorted according to the possible moves, and only a small fraction of moves (the very best ones) is actually executed. Then the process is repeated until the fill rates are acceptable. Essentially, until $\text{low} = \text{high} = \emptyset$.

The reasoning behind it is that it is rather cheap to fetch manifests and do some calculations, but it is expensive to move fragments and update the manifests accordingly. We really want a move to count. Also we want to avoid a combination of $\{\text{high} \rightsquigarrow \text{ok}\}$ and $\{\text{ok} \rightsquigarrow \text{low}\}$ where a single $\{\text{high} \rightsquigarrow \text{low}\}$ would have been possible.

4.14.5 Scaling maintenance

It's possible to run multiple maintenance agents in parallel. To avoid these agents interfere and for example race to repair the same object, they can be assigned a team position. Agent 7/9 will only perform maintenance on $\{\text{namespace} \mid \text{id} \bmod 9 = 7\}$. This keeps them out of each other's way, while allowing various team sizes.

4.14.6 Scheduling

Maintenance agents can run in different flavours:

- `all-in-one` The agent runs all tasks for the assigned namespaces
- `no-rebalance` The agent runs all tasks except rebalancing
- `only-rebalance` only perform rebalancing on the assigned namespaces

This allows scheduled rebalancing. Rebalancing is rather hard on the OSDs and one wants to be able to schedule it (or more of it) during slow hours, while calming down during known peak periods.

5 System Layout

5.1 Alba Manager

An alba manager is an arakoon cluster running with the albamgr plugin. It's the entry point for all clients. It knows which disks belong to this alba instance. It knows which namespaces exist and on which nsm hosts they can be found.

5.2 NSM Host

An nsm host is an Arakoon cluster running with the `nsm_host` plugin. It is registered with the alba manager and can host namespace managers.

5.3 TODO Namespace Manager

The namespace manager is the remote API offered by the NSM host to manipulate most of the object metadata during normal operation. Its coordinates can be retrieved from the albamgr by (proxy) clients and maintenance agents.

```
method put_object:
   overwrite -> gc_epoch -> Manifest.t ->
   Manifest.t option Lwt.t

method get_object_manifest_by_name:
   object_id -> Manifest.t option Lwt.t

method list_objects :
   first:object_name
   finc: bool ->
   last: (object_name * bool) option ->
  max:int -> reverse:bool ->
   object_name counted_list_more Lwt.t

method update_manifest :
   object_name -> object_id ->
   (chunk_id * fragment_id * osd_id) list ->
   gc_epoch -> version_id -> unit Lwt.t

method list_device_objects
   osd_id ->
   first:object_id -> finc:bool ->
   last:(object_id * bool option) ->
  max:int ->
   reverse:bool ->
   Manifest.t counted_list_more Lwt.t
 ...
```

An excerpt of the client API shows some typical methods: `put_object`, `get_object_manifest_by_name`, `list_objects`, ... used by the proxy-client or the maintenance agent. `list_device_objects` is just one of many methods to query the database of object manifests.