Red Hat Ceph Storage 1.3
Architecture Guide

Guide on Red Hat Ceph Storage Architecture

Red Hat Ceph Storage Documentation Team

Red Hat Ceph Storage 1.3 Architecture Guide

Guide on Red Hat Ceph Storage Architecture

Legal Notice

Abstract

This document is an architecture guide for Red Hat Ceph Storage.

Table of Contents

CHAPTER 1. OVERVIEW

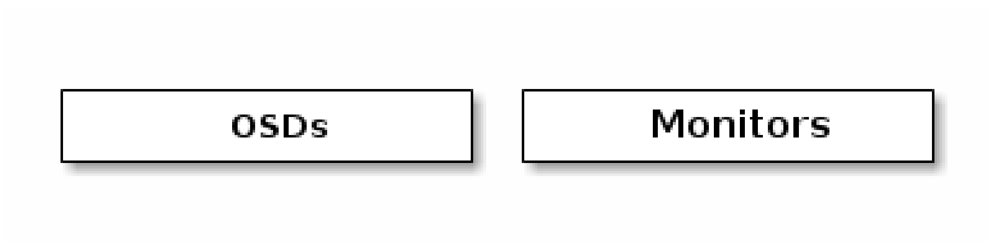
Red Hat Ceph is a distributed data object store designed to provide excellent performance, reliability and scalability. Distributed object stores are the future of storage, because they accommodate unstructured data, and because clients can use modern object interfaces and legacy interfaces simultaneously. For example:

- Native language binding interfaces (C/C++, Java, Python)
- RESTful interfaces (S3/Swift)
- Block device interfaces
- Filesystem interfaces

The power of Red Hat Ceph can transform your organization’s IT infrastructure and your ability to manage vast amounts of data, especially for cloud computing platforms like RHEL OSP. Red Hat Ceph delivers **extraordinary** scalability–thousands of clients accessing petabytes to exabytes of data and beyond.

At the heart of every Ceph deployment is the 'Ceph Storage Cluster.' It consists of two types of daemons:

- **Ceph OSD Daemon:** Ceph OSDs store data on behalf of Ceph clients. Additionally, Ceph OSDs utilize the CPU and memory of Ceph nodes to perform data replication, rebalancing, recovery, monitoring and reporting functions.
- **Ceph Monitor:** A Ceph monitor maintains a master copy of the Ceph storage cluster map with the current state of the storage cluster.

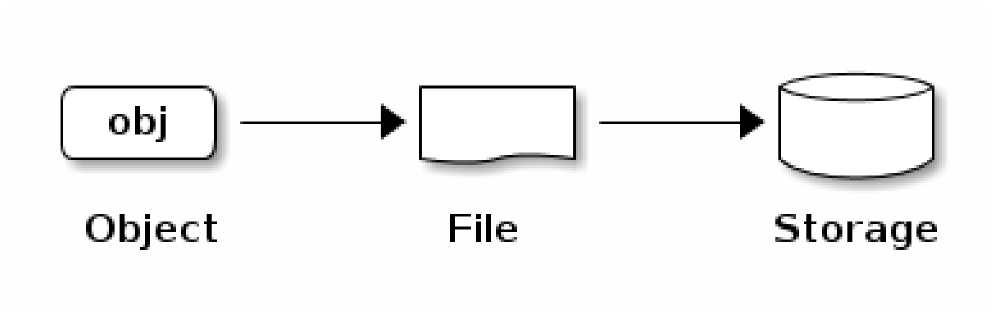


Ceph client interfaces read data from and write data to the Ceph storage cluster. Clients need the following data to communicate with the Ceph storage cluster:

- The Ceph configuration file, or the cluster name (usually **ceph**) and monitor address
- The pool name
- The user name and the path to the secret key.

Ceph clients maintain object IDs and the pool name(s) where they store the objects, but they do not need to maintain an object-to-OSD index or communicate with a centralized object index to look up data object locations. To store and retrieve data, Ceph clients access a Ceph monitor and retrieve the latest copy of the storage cluster map. Then, Ceph clients can provide an object name and pool name, and Ceph will use the cluster map and the CRUSH (Controlled Replication Under Scalable Hashing) algorithm to compute the object placement group and the primary Ceph OSD for storing or retrieving data. The Ceph client connects to the primary OSD where it may perform read and write operations. There is no intermediary server, broker or bus between the client and the OSD.

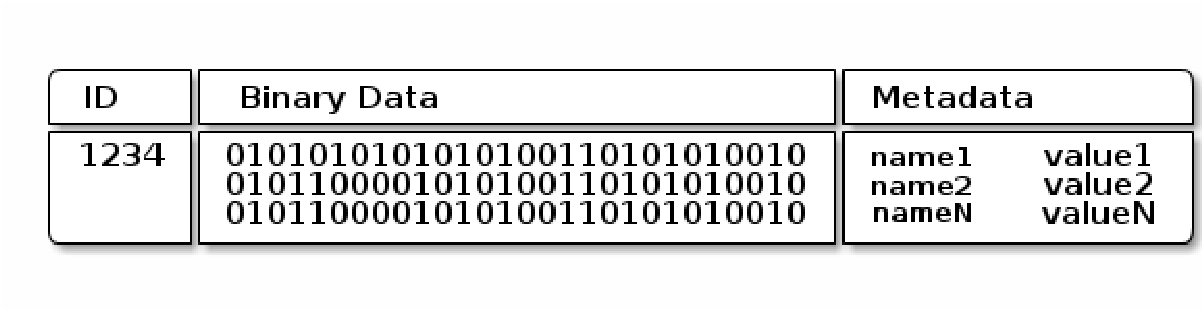
When an OSD stores data, it receives data from a Ceph client—whether the client is a Ceph Block Device, a Ceph Object Gateway or another interface—and it stores the data as an object. Each object corresponds to a file in a filesystem, which is stored on a storage device such as a hard disk. Ceph OSDs handle the read/write operations on the storage device.



Note

An object ID is unique across the entire cluster, not just the local filesystem.

Ceph OSDs store all data as objects in a flat namespace (e.g., no hierarchy of directories). An object has a cluster-wide unique identifier, binary data, and metadata consisting of a set of name/value pairs. The semantics are completely up to Ceph clients. For example, the Ceph block device maps a block device image to a series of objects stored across the cluster.



Note

Objects consisting of a unique ID, data, and name/value paired metadata can represent both structured and unstructured data, as well as legacy and leading edge data storage interfaces.

CHAPTER 2. STORAGE CLUSTER ARCHITECTURE

A Ceph Storage Cluster accommodates large numbers of Ceph nodes for effectively limitless scalability, high availability and performance. Each node leverages commodity hardware and intelligent Ceph daemons that communicate with each other to:

- Store and retrieve data
- Replicate data
- Monitor and report on cluster health (heartbeating)
- Redistribute data dynamically (backfilling)
- Ensure data integrity (scrubbing)
- Recover from failures.

To the Ceph client interface that reads and writes data, a Ceph storage cluster looks like a simple pool where it stores data. However, the storage cluster performs many complex operations in a manner that is completely transparent to the client interface. Ceph clients and Ceph OSDs both use the CRUSH (Controlled Replication Under Scalable Hashing) algorithm. The following sections provide details on how CRUSH enables Ceph to perform these operations seamlessly.

2.1. POOLS

The Ceph storage cluster stores data objects in logical partitions called 'Pools.' You can create pools for particular types of data, such as for block devices, object gateways, or simply just to separate one group of users from another.

From the perspective of a Ceph client, the storage cluster is very simple. When a Ceph client reads or writes data (i.e., called an i/o context), it **always** connects to a storage pool in the Ceph storage cluster. The client specifies the pool name, a user and a secret key, so the pool appears to act as a logical partition with access controls to data objects.

In actual fact, a Ceph pool is not only a logical partition for storing object data, it plays a critical role in how the Ceph storage cluster distributes and stores data—yet, these complex operations are completely transparent to the Ceph client. Ceph pools define:

- **Pool Type:** In early versions of Ceph, a pool simply maintained multiple deep copies of an object. Today, Ceph can maintain multiple copies of an object, or it can use erasure coding. Since the methods for ensuring data durability differ between deep copies and erasure coding, Ceph supports a pool type. Pool types are completely transparent to the client.
- **Cache Tier:** In early versions of Ceph, a client could only write data directly to an OSD. Today, Ceph can also support a cache tier and a backing storage tier. The cache tier is a pool consisting of higher cost/higher performance hardware such as SSDs. The backing storage tier is the primary storage pool. Cache tiers speed up read and write operations with high performance hardware. Cache tiers (and their pool names) are completely transparent to the client.
- **Placement Groups:** In an exabyte scale storage cluster, a Ceph pool might store millions of data objects or more. Since Ceph must handle data durability (replicas or erasure code chunks), scrubbing, replication, rebalancing and recovery, managing data on a per-object basis presents a scalability and performance bottleneck. Ceph addresses this bottleneck by sharding a pool into placement groups. CRUSH assigns each object to a placement group, and each placement group to a set of OSDs.

- **CRUSH Ruleset:** High availability, durability and performance are extremely important in Ceph. The CRUSH algorithm computes the placement group for storing an object and the Acting Set of OSDs for the placement group. CRUSH also plays other important roles: namely, CRUSH can recognize failure domains and performance domains (i.e., types of storage media and the nodes, racks, rows, etc. that contain them). CRUSH enables clients to write data across failure domains (rooms, racks, rows, etc.), so that if a large-grained portion of a cluster fails (e.g., a rack), the cluster can still operate in a degraded state until it recovers. CRUSH enables clients to write data to particular types of hardware (performance domains), such as SSDs, hard drives with SSD journals, or hard drives with journals on the same drive as the data. The CRUSH ruleset determines failure domains and performance domains for the pool.

- **Durability**: In exabyte scale storage clusters, hardware failure is an expectation and not an exception. When using data objects to represent larger-grained storage interfaces such as a block device, losing one or more data objects for that larger-grained interface can compromise the integrity of the larger-grained storage entity—potentially rendering it useless. So data loss is intolerable. Ceph provides high data durability in two ways: first, replica pools will store multiple deep copies of an object using the CRUSH failure domain to physically separate one data object copy from another (i.e., copies get distributed to separate hardware). This increases durability during hardware failures. Second, erasure coded pools store each object as **`K+M`** chunks, where **`K`** represents data chunks and **`M`** represents coding chunks. The sum represents the number of OSDs used to store the object and the the **`M`** value represents the number of OSDs that can fail and still restore data should the **`M`** number of OSDs fail.

From the client perspective, Ceph is elegant and simple. The client simply reads from and writes to pools. However, pools play an important role in data durability, performance and high availability.

2.2. AUTHENTICATION

To identify users and protect against man-in-the-middle attacks, Ceph provides its **`cephx`** authentication system to authenticate users and daemons.

Note

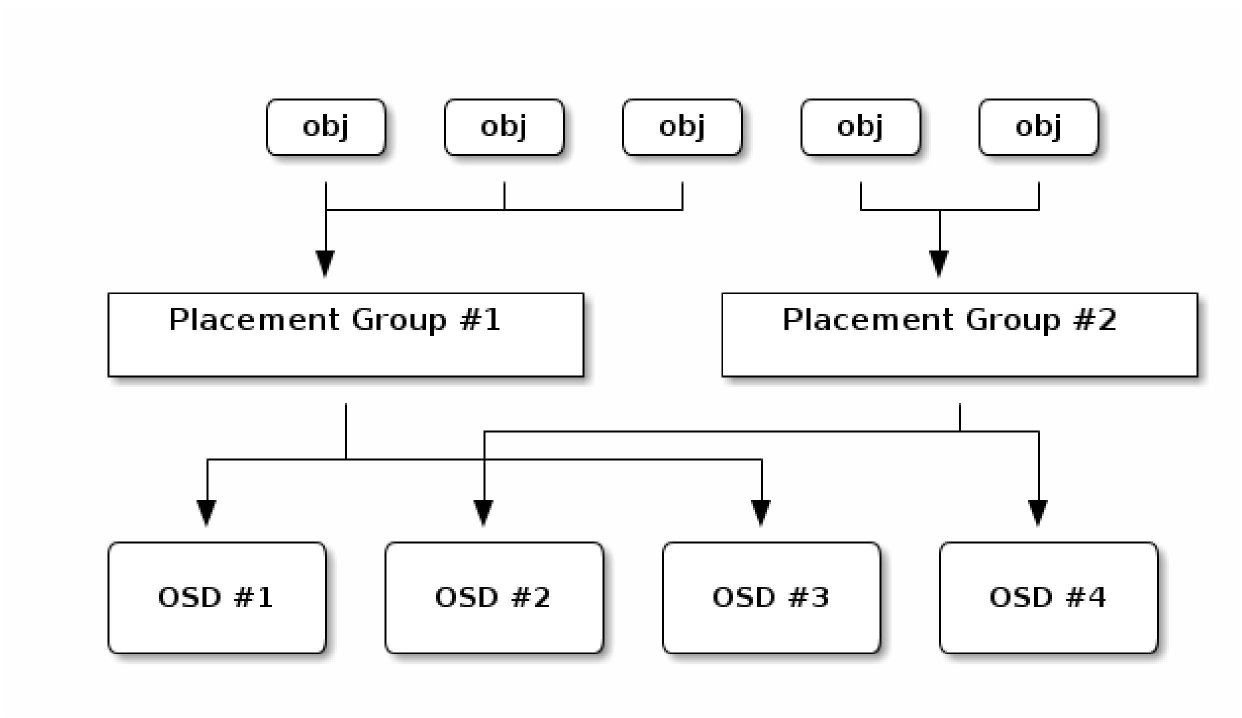
The **`cephx`** protocol does not address data encryption in transport (e.g., SSL/TLS) or encryption at rest.

Cephx uses shared secret keys for authentication, meaning both the client and the monitor cluster have a copy of the client’s secret key. The authentication protocol is such that both parties are able to prove to each other they have a copy of the key without actually revealing it. This provides mutual authentication, which means the cluster is sure the user possesses the secret key, and the user is sure that the cluster has a copy of the secret key.

2.3. PLACEMENT GROUPS (PGS)

Ceph shards a pool into placement groups distributed evenly and pseudo-randomly across the cluster. The CRUSH algorithm assigns each object to a placement group, and assigns each placement group to an Acting Set of OSDs—creating a layer of indirection between the Ceph client and the OSDs storing the copies of an object. If the Ceph client "knew" which Ceph OSD had which object, that would create a tight coupling between the Ceph client and the Ceph OSD. Instead, the CRUSH algorithm dynamically assigns each object to a placement group and then assigns each placement group to a set of Ceph OSDs. This layer of indirection allows the Ceph storage cluster to re-balance dynamically when new Ceph OSD come online or when Ceph OSDs fail. By managing millions of objects within the context of hundreds to thousands of placement groups, the Ceph storage cluster can grow, shrink and recover from failure efficiently.

The following diagram depicts how CRUSH assigns objects to placement groups, and placement groups to OSDs.



If a pool has too few placement groups relative to the overall cluster size, Ceph will have too much data per placement group and won’t perform well. If a pool has too many placement groups relative to the overall cluster, Ceph OSDs will use too much RAM and CPU and won’t perform well. Setting an appropriate number of placement groups per pool, and an upper limit on the number of placement groups assigned to each OSD in the cluster is critical to Ceph performance.

2.4. CRUSH

Ceph assigns a CRUSH ruleset to a pool. When a Ceph client stores or retrieves data in a pool, Ceph identifies the CRUSH ruleset, a rule and the top-level bucket in the rule for storing and retrieving data. As Ceph processes the CRUSH rule, it identifies the primary OSD that contains the placement group for an object. That enables the client to connect directly to the OSD and read or write data.

To map placement groups to OSDs, a CRUSH map defines a hierarchical list of bucket types (i.e., under **`types`** in the generated CRUSH map). The purpose of creating a bucket hierarchy is to segregate the leaf nodes by their failure domains and/or performance domains, such drive type, hosts, chassis, racks, power distribution units, pods, rows, rooms, and data centers.

With the exception of the leaf nodes representing OSDs, the rest of the hierarchy is arbitrary, and you may define it according to your own needs if the default types don’t suit your requirements. CRUSH supports a directed acyclic graph that models your Ceph OSD nodes, typically in a hierarchy. So you can support multiple hierarchies with multiple root nodes in a single CRUSH map. For example, you can create a hierarchy of SSDs for a cache tier, a hierarchy of hard drives with SSD journals, etc.

2.5. I/O OPERATIONS

Ceph clients retrieve a 'Cluster Map' from a Ceph monitor, and perform i/o on objects within pools. The pool’s CRUSH ruleset and the number of placement groups are the main factors that determine how Ceph will place the data. With the latest version of the cluster map, the client knows about all of the monitors and OSDs in the cluster. **However, it doesn’t know anything about object locations.**

The only inputs required by the client are the object ID and the pool name. It’s simple: Ceph stores data in named pools (e.g., "liverpool"). When a client wants to store a named object (e.g., "john," "paul," "george," "ringo", etc.) in a pool it takes the object name, a hash code, the number of PGs in the pool and the pool name as inputs and CRUSH (Controlled Replication Under Scalable Hashing) calculates the ID of the placement group and the primary OSD for the placement group.

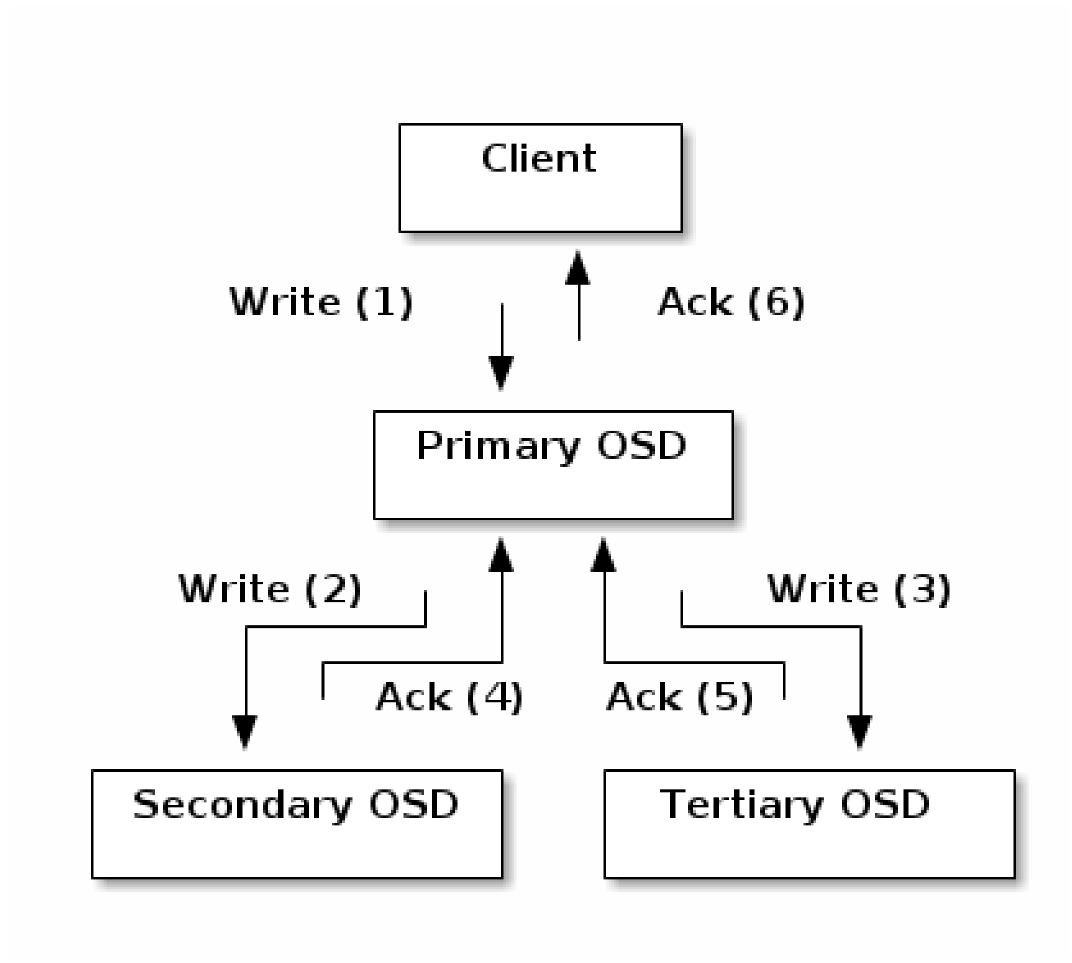
Ceph clients use the following steps to compute PG IDs.

1. The client inputs the pool ID and the object ID. (e.g., pool = "liverpool" and object-id = "john")
2. CRUSH takes the object ID and hashes it.
3. CRUSH calculates the hash modulo of the number of PGs. (e.g., **`58`**) to get a PG ID.
4. CRUSH calculates the primary OSD corresponding to the PG ID.
5. The client gets the pool ID given the pool name (e.g., "liverpool" = **`4`**)
6. The client prepends the pool ID to the PG ID (e.g., **`4.58`**).
7. The client performs an object operation (e.g., write, read, delete, etc.) by communicating directly with the Primary OSD in the Acting Set.

The topology and state of the Ceph storage cluster are relatively stable during a session (i/o context). Empowering a Ceph client to compute object locations is much faster than requiring the client make a query to the storage over a chatty session for each read/write operation. The CRUSH algorithm allows a client to compute where objects *should* be stored, and **enables the client to contact the primary OSD in the acting set directly** to store or retrieve the objects. Since a cluster at the exabyte scale has thousands of OSDs, network over subscription between a client and a Ceph OSD is not a significant problem. If the cluster state changes, the client can simply request an update to the cluster map from the Ceph monitor.

2.5.1. Replicated I/O

Like Ceph clients, Ceph OSDs can contact Ceph monitors to retrieve the latest copy of the cluster map. Ceph OSDs also use the CRUSH algorithm, but they use it to compute where replicas of objects should be stored. In a typical write scenario, a Ceph client uses the CRUSH algorithm to compute the placement group ID and the primary OSD in the Acting Set for an object. When the client writes the object to the primary OSD, the primary OSD finds the number of replicas that it should store (i.e., **`osd_pool_default_size = n`**). Then, the primary OSD takes the object ID, pool name and the cluster map and uses the CRUSH algorithm to calculate the ID(s) of secondary OSD(s) for the acting set. The primary OSD writes the object to the secondary OSD(s). When the primary OSD receives an acknowledgment from the secondary OSD(s) and the primary OSD itself completes its write operation, it acknowledges a successful write operation to the Ceph client.



With the ability to perform data replication on behalf of Ceph clients, Ceph OSD Daemons relieve Ceph clients from that duty, while ensuring high data availability and data safety.

Note

The primary OSD and the secondary OSDs are typically configured to be in separate failure domains (i.e., rows, racks, nodes, etc.). CRUSH computes the ID(s) of the secondary OSD(s) with consideration for the failure domains.

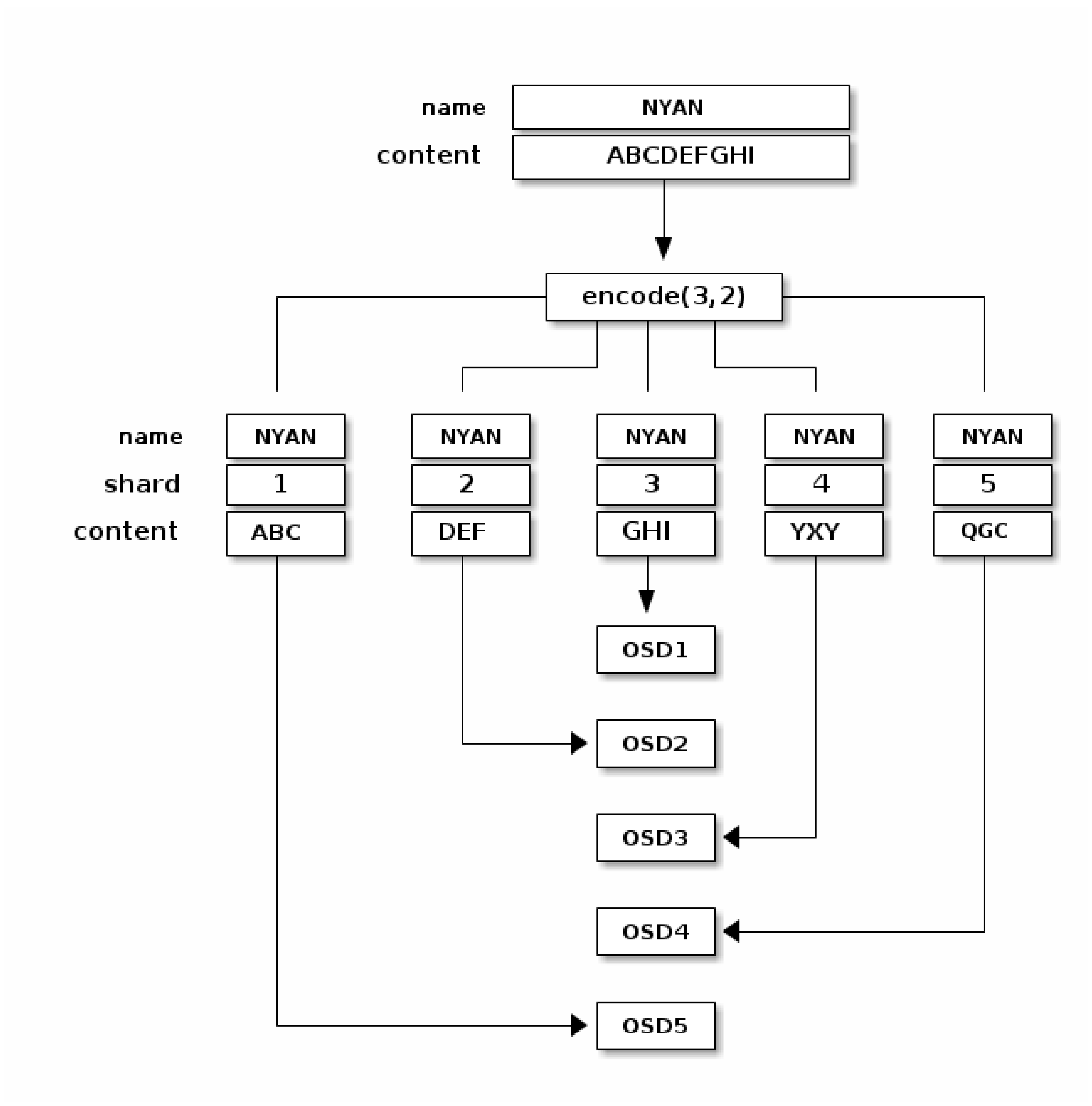
2.5.2. Erasure-coded I/O

Erasure code is actually a forward error correction (FEC) code which transforms a message of K chunks into a longer message (code word) with N chunks such that the original message can be recovered from a subset of the N chunks. Over the time, different algorithms have developed for erasure coding out of which one of the earliest and most commonly used is **`Reed-Solomon`** algorithm. You can understand it better with the equation **`N = K+M`** where the variable **`K`** is the original amount of data chunks, variable **`M`** stands for the extra or redundant chunks that are added to provide protection from failures and the variable **`N`** is the total number of chunks created after the **`erasure coding`** process. The value of **`M`** is simply **`N-K`** which means that **`N-K`** redundant chunks are computed from **`K`** original data chunks. This approach guarantees that all the original data are accessible as long as **`K`** among the **`N`** chunks are functional. That is, the system is resilient to arbitrary **`N-K`** failures. For instance, in a 10 (K) of 16 (N) configuration, or erasure coding 10/16, six extra chunks **`M = K-N`** i.e, **`16-10 = 6`** would be added to the 10 base chunks (K). The 16 data chunks (N) would be spread across 16 OSDs. The original file could be reconstructed from the 10 verified data chunks even if 6 OSDs fail. It means that there won’t be any loss of data. Thus, it ensures a very high level of fault tolerance.

Like replicated pools, in an erasure-coded pool the primary OSD in the up set receives all write operations. In replicated pools, Ceph makes a deep copy of each object in the placement group on the secondary OSD(s) in the set. For erasure coding, the process is a bit different. An erasure coded pool stores each object as **`K+M`** chunks. It is divided into **`K`** data chunks and **`M`** coding chunks. The pool is configured to have a size of **`K+M`** so that each chunk is stored in an OSD in the acting set. The rank of the chunk is stored as an attribute of the object. The primary OSD is responsible for encoding the payload into **`K+M`** chunks and sends them to the other OSDs. It is also responsible for maintaining an authoritative version of the placement group logs.

For instance an erasure coded pool is created to use five OSDs (**`K+M = 5`**) and sustain the loss of two of them (**`M = 2`**).

When the object **NYAN** containing **`ABCDEFGHI`** is written to the pool, the erasure encoding function splits the content into three data chunks simply by dividing the content in three: the first contains **`ABC`**, the second **`DEF`** and the last **`GHI`**. The content will be padded if the content length is not a multiple of **`K`**. The function also creates two coding chunks: the fourth with **`YXY`** and the fifth with **`GQC`**. Each chunk is stored in an OSD in the acting set. The chunks are stored in objects that have the same name (**NYAN**) but reside on different OSDs. The order in which the chunks were created must be preserved and is stored as an attribute of the object (**`shard_t`**), in addition to its name. Chunk 1 contains **`ABC`** and is stored on **OSD5** while chunk 4 contains **`YXY`** and is stored on **OSD3**.



When the object **NYAN** is read from the erasure coded pool, the decoding function reads three chunks: chunk 1 containing **`ABC`**, chunk 3 containing **`GHI`** and chunk 4 containing **`YXY`**. Then, it rebuilds the original content of the object **`ABCDEFGHI`**. The decoding function is informed that the chunks 2 and 5 are missing (they are called 'erasures'). The chunk 5 could not be read because the **OSD4** is out. The decoding function can be called as soon as three chunks are read: **OSD2** was the slowest and its chunk was not taken into account.

This splitting of data into chunks is independent from object placement. Your CRUSH ruleset along with the erasure-coded pool profile determines the placement of chunks on the OSDs. For instance, using the Locally Repairable Code (**`lrc`**) plugin in your profile creates additional chunks and requires fewer OSDs to recover from. You have this **`lrc`** profile configuration: **`K=4, M=2, and L=3`**. Six chunks are created (**`K+M`**), just as the **`jerasure`** plugin would, but the locality value (**`L=3`**) requires 2 more chunks to be created locally. The additional chunks are calculated as such, **`(K+M)/L`**. If the OSD containing chunk 0 fails, this chunk can be recovered by using chunks 1, 2 and the first local chunk. In this case, only 3 chunks instead of 5 chunks are required for recovery. For more information about CRUSH, the erasure-coding profiles and plugins, please see the Ceph Storage Strategies Guide.

Note

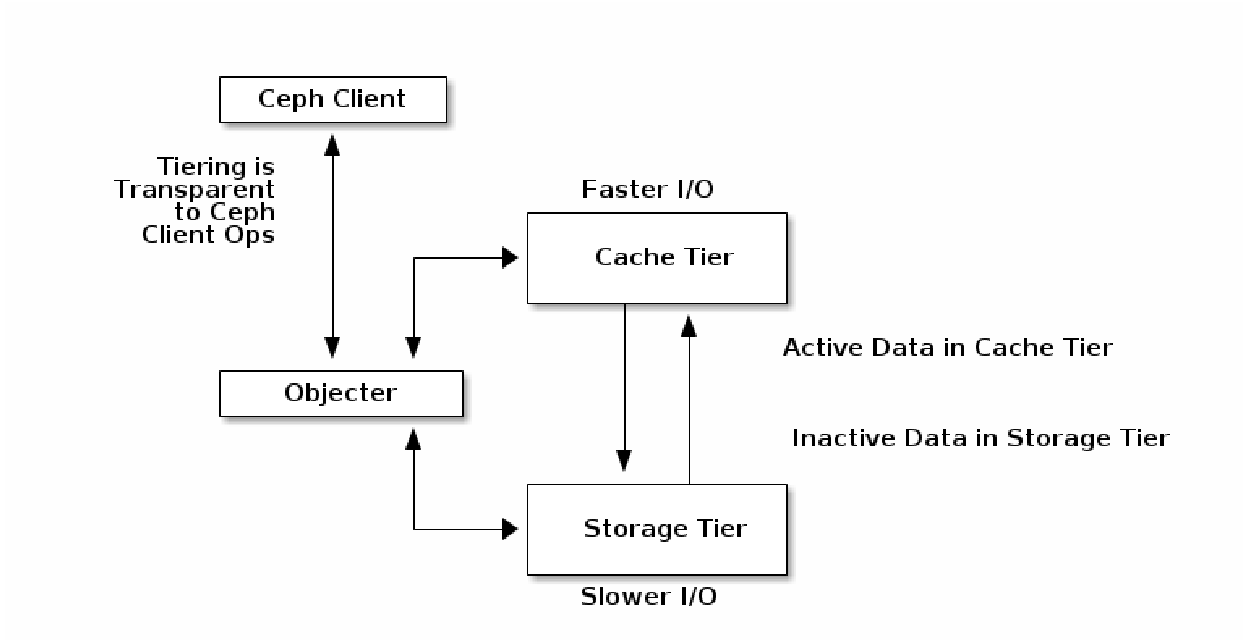
Object Map for erasure-coded pools is disabled and can not be enabled. For more details on Object Map, please see the Object Map section below.

Note

Erasure-coded pools are only supported with the RADOS Gateway (RGW). Using erasure-coded pools with a RADOS Block Device (RBD) is not supported.

2.5.3. Cache-Tiering I/O (Tech Preview)

A cache tier provides Ceph clients with better I/O performance for a subset of the data stored in a backing storage tier. Cache tiering involves creating a pool of relatively fast/expensive storage devices (e.g., solid state drives) configured to act as a cache tier, and a backing pool of either erasure-coded or relatively slower/cheaper devices configured to act as an economical storage tier. The Ceph objecter handles where to place the objects and the tiering agent determines when to flush objects from the cache to the backing storage tier. So the cache tier and the backing storage tier are completely transparent to Ceph clients.


Ceph Client
Tiering is Transparent to Ceph Client Ops
Faster I/O
Cache Tier
Active Data in Cache Tier
Objecter
Inactive Data in Storage Tier
Storage Tier
Slower I/O

2.6. SELF-MANAGEMENT OPERATIONS

Ceph clusters perform a lot of self monitoring and management operations automatically. For example, Ceph OSDs can check the cluster health and report back to the Ceph monitors; and by using CRUSH to assigning objects to placement groups and placement groups to a set of OSDs, Ceph OSDs can use the CRUSH algorithm to rebalance the cluster or recover from OSD failures dynamically. The following sections describe some of the operations Ceph conducts for you.

2.6.1. Heartbeating

Ceph OSDs join a cluster and report to Ceph monitors on their status. At the lowest level, the Ceph

OSD status is **`up`** or **`down`** reflecting whether or not it is running and able to service Ceph client requests. If a Ceph OSD is **`down`** and **`in`** the Ceph storage cluster, this status may indicate the failure of the Ceph OSD. If a Ceph OSD is not running (e.g., it crashes), the Ceph OSD cannot notify the Ceph monitor that it is **`down`**. The Ceph monitor can ping a Ceph OSD daemon periodically to ensure that it is running. However, Ceph also empowers Ceph OSDs to determine if a neighboring OSD is **`down`**, to update the cluster map and to report it to the Ceph monitor(s). This means that Ceph monitors can remain light weight processes.

2.6.2. Peering

Ceph OSD daemons perform 'peering', which is the process of bringing all of the OSDs that store a Placement Group (PG) into agreement about the state of all of the objects (and their metadata) in that PG. Peering issues usually resolve themselves.

Note

When Ceph monitors agree on the state of the OSDs storing a placement group, that does not mean that the placement group has the latest contents.

When Ceph stores a placement group in an acting set of OSDs, refer to them as *Primary*, *Secondary*, and so forth. By convention, the *Primary* is the first OSD in the *Acting Set*, and is responsible for coordinating the peering process for each placement group where it acts as the *Primary*, and is the **ONLY** OSD that that will accept client-initiated writes to objects for a given placement group where it acts as the *Primary*.

When a series of OSDs are responsible for a placement group, that series of OSDs, we refer to them as an *Acting Set*. An *Acting Set* may refer to the Ceph OSD Daemons that are currently responsible for the placement group, or the Ceph OSD Daemons that were responsible for a particular placement group as of some epoch.

The Ceph OSD daemons that are part of an *Acting Set* may not always be **`up`**. When an OSD in the *Acting Set* is **`up`**, it is part of the *Up Set*. The *Up Set* is an important distinction, because Ceph can remap PGs to other Ceph OSDs when an OSD fails.

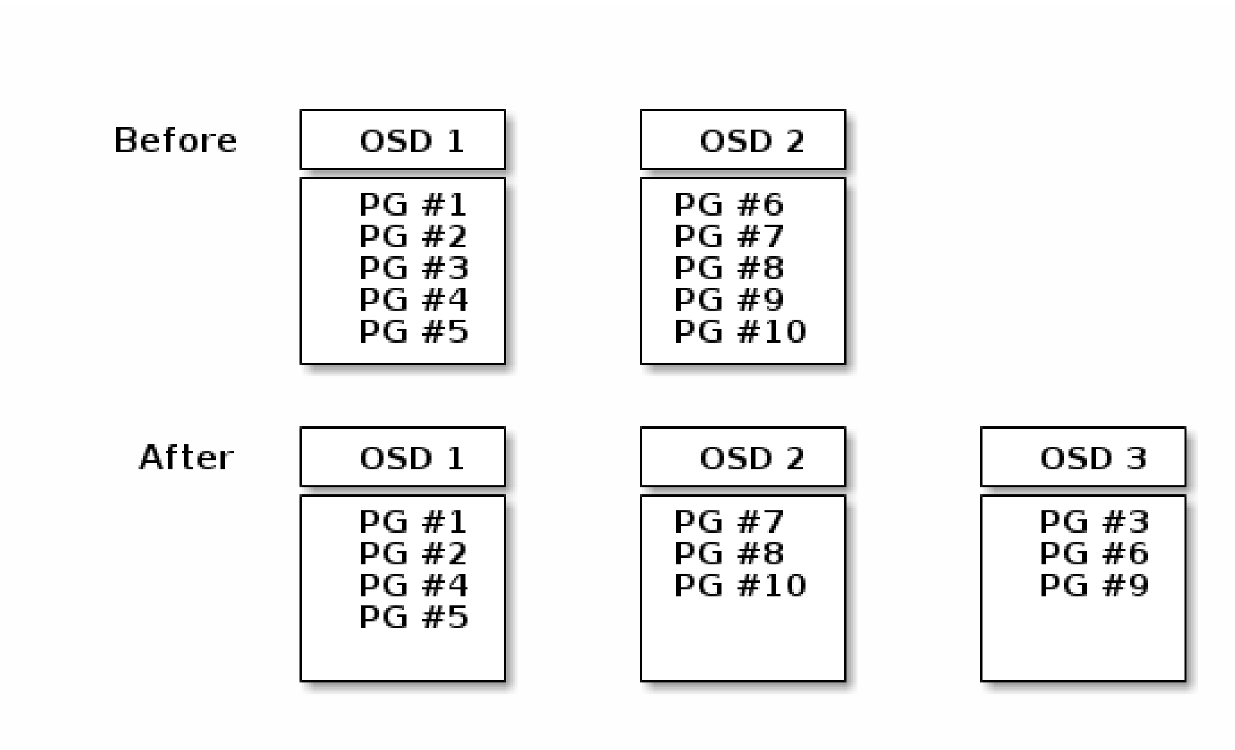
Note

In an *Acting Set* for a PG containing **`osd.25`**, **`osd.32`** and **`osd.61`**, the first OSD, **`osd.25`**, is the *Primary*. If that OSD fails, the Secondary, **`osd.32`**, becomes the *Primary*, and **`osd.25`** will be removed from the *Up Set*.

2.6.3. Rebalancing and Recovery

When you add a Ceph OSD to a Ceph storage cluster, the cluster map gets updated with the new OSD. This changes the cluster map. Consequently, it changes object placement, because it changes an input for the CRUSH calculations. CRUSH places data evenly, but pseudo randomly. So only a small amount of data moves when you add a new OSD. The amount of data is usually the the number of OSDs you add divided by the total amount of data in the cluster (e.g., in a cluster with 50 OSDs, 1/50th or 2% of your data might move when adding an OSD).

The following diagram depicts the rebalancing process (albeit rather crudely, since it is substantially less impactful with large clusters) where some, but not all of the PGs migrate from existing OSDs (OSD 1, and OSD 2) to the new OSD (OSD 3). Even when rebalancing, CRUSH is stable. Many of the placement groups remain in their original configuration, and each OSD gets some added capacity, so there are no load spikes on the new OSD after rebalancing is complete.



2.6.4. Scrubbing

As part of maintaining data consistency and cleanliness, Ceph OSD Daemons can scrub objects within placement groups. That is, Ceph OSD Daemons can compare object metadata in one placement group with its replicas in placement groups stored on other OSDs. Scrubbing (usually performed daily) catches bugs or filesystem errors. Ceph OSD Daemons also perform deeper scrubbing by comparing data in objects bit-for-bit. Deep scrubbing (usually performed weekly) finds bad sectors on a drive that weren’t apparent in a light scrub.

2.7. HIGH AVAILABILITY

In addition to the high scalability enabled by the CRUSH algorithm, Ceph must also maintain high availability. This means that Ceph clients must be able to read and write data even when the cluster is in a degraded state, or when a monitor fails.

2.7.1. Data Copies

In a replicated storage pool, Ceph needs multiple copies of an object to operate in a degraded state. Ideally, a Ceph storage cluster enables a client to read and write data even if one of the OSDs in an acting set fails. For this reason, Ceph defaults to making three copies of an object with a minimum of two copies clean for write operations. Ceph will still preserve data even if two OSDs fail; however, it will interrupt write operations.

In an erasure-coded pool, Ceph needs to store chunks of an object across multiple OSDs so that it can operate in a degraded state. Similar to replicated pools, ideally an erasure-coded pool enables a Ceph client to read and write in a degraded state. For this reason, we recommend **`K+M=5`** to store chunks across 5 OSDs with **`M=2`** to allow the failure of two OSDs and retain the ability to recover data.

2.7.2. Monitor Cluster

Before Ceph Clients can read or write data, they must contact a Ceph Monitor to obtain the most recent copy of the cluster map. A Ceph Storage Cluster can operate with a single monitor; however, this introduces a single point of failure (i.e., if the monitor goes down, Ceph Clients cannot read or write data).

For added reliability and fault tolerance, Ceph supports a cluster of monitors. In a cluster of monitors, latency and other faults can cause one or more monitors to fall behind the current state of the cluster. For this reason, Ceph must have agreement among various monitor instances regarding the state of the cluster. Ceph always uses a majority of monitors (e.g., 1, 2:3, 3:5, 4:6, etc.) and the Paxos algorithm to establish a consensus among the monitors about the current state of the cluster. Monitors hosts require NTP to prevent clock drift.

2.7.3. CephX

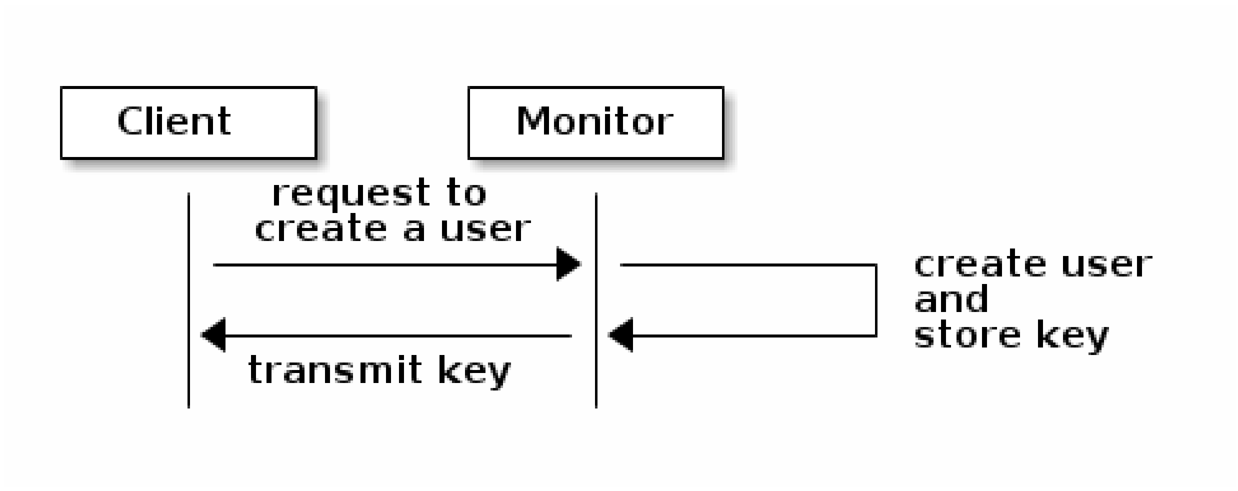
The **`cephx`** authentication protocol operates in a manner with behavior similar to Kerberos.

A user/actor invokes a Ceph client to contact a monitor. Unlike Kerberos, each monitor can authenticate users and distribute keys, so there is no single point of failure or bottleneck when using **`cephx`**. The monitor returns an authentication data structure similar to a Kerberos ticket that contains a session key for use in obtaining Ceph services. This session key is itself encrypted with the user’s permanent secret key, so that only the user can request services from the Ceph monitor(s). The client then uses the session key to request its desired services from the monitor, and the monitor provides the client with a ticket that will authenticate the client to the OSDs that actually handle data. Ceph monitors and OSDs share a secret, so the client can use the ticket provided by the monitor with any OSD or metadata server in the cluster. Like Kerberos, **`cephx`** tickets expire, so an attacker cannot use an expired ticket or session key obtained surreptitiously. This form of authentication will prevent attackers with access to the communications medium from either creating bogus messages under another user’s identity or altering another user’s legitimate messages, as long as the user’s secret key is not divulged before it expires.

To use **`cephx`**, an administrator must set up users first. In the following diagram, the **`client.admin`** user invokes **`ceph auth get-or-create-key`** from the command line to generate a username and secret key. Ceph’s **`auth`** subsystem generates the username and key, stores a copy with the monitor(s) and transmits the user’s secret back to the **`client.admin`** user. This means that the client and the monitor share a secret key.

Note

The **`client.admin`** user must provide the user ID and secret key to the user in a secure manner.


Client
Monitor
request to create a user
create user and store key
transmit key

CHAPTER 3. CLIENT ARCHITECTURE

Ceph clients differ in their materially in how they present data storage interfaces. A Ceph block device presents block storage that mounts just like a physical storage drive. A Ceph gateway presents an object storage service with S3-compliant and Swift-compliant RESTful interfaces with its own user management. However, all Ceph clients use the Reliable Autonomic Distributed Object Store (RADOS) protocol to interact with the Ceph storage cluster; and, they all have the same basic needs:

- The Ceph configuration file, or the cluster name (usually **`ceph`**) and monitor address
- The pool name
- The user name and the path to the secret key.

Ceph clients tend to follow some similar patters, such as object-watch-notify and striping. The following sections describe a little bit more about RADOS, librados and common patterns used in Ceph clients.

3.1. NATIVE PROTOCOL AND `LIBRADOS`

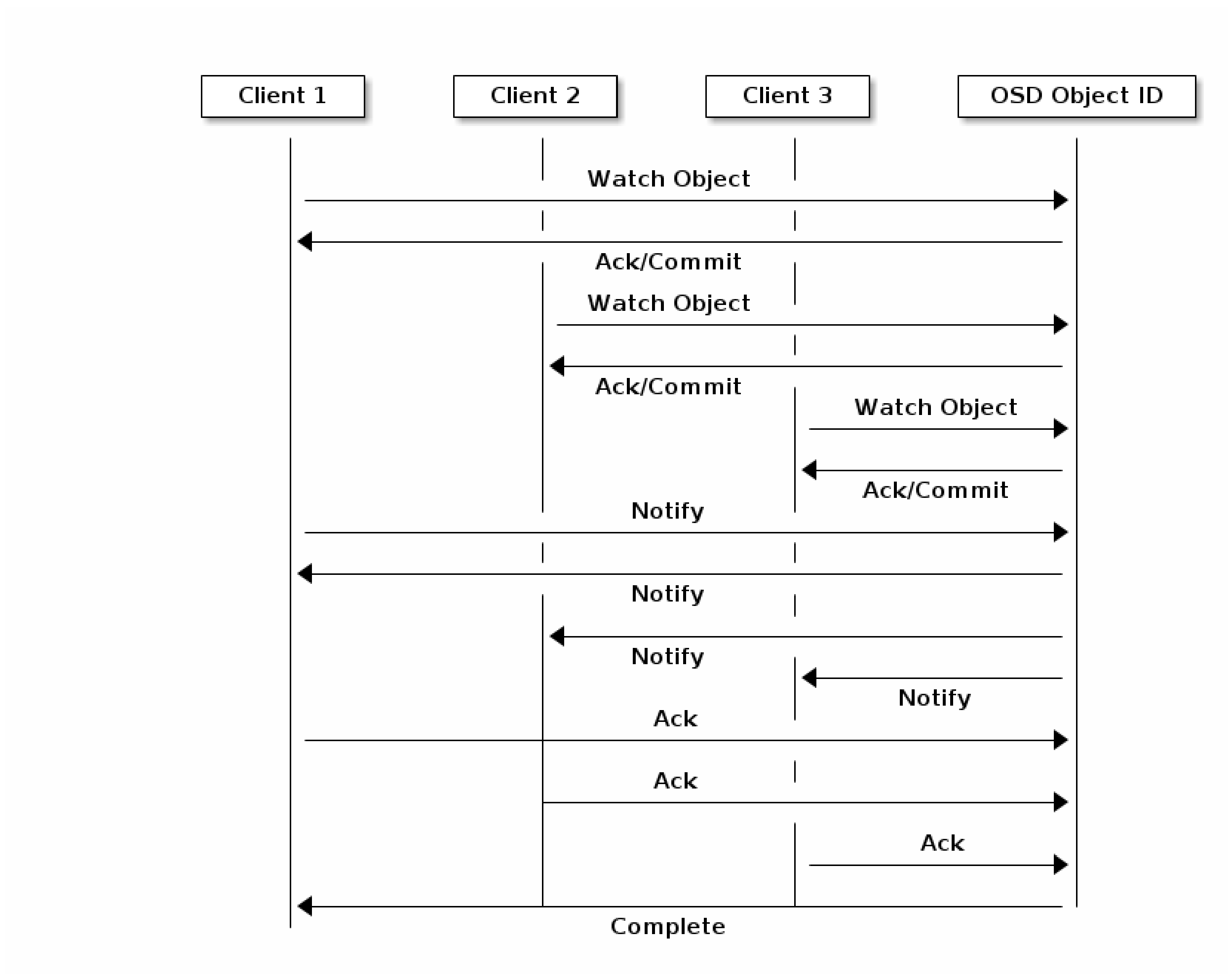
Modern applications need a simple object storage interface with asynchronous communication capability. The Ceph Storage Cluster provides a simple object storage interface with asynchronous communication capability. The interface provides direct, parallel access to objects throughout the cluster.

- Pool Operations
- Snapshots
- Read/Write Objects
  - Create or Remove
  - Entire Object or Byte Range
  - Append or Truncate
- Create/Set/Get/Remove XATTRs
- Create/Set/Get/Remove Key/Value Pairs
- Compound operations and dual-ack semantics

3.2. OBJECT WATCH/NOTIFY

3.3. OBJECT WATCH/NOTIFY

A Ceph client can register a persistent interest with an object and keep a session to the primary OSD open. The client can send a notification message and payload to all watchers and receive notification when the watchers receive the notification. This enables a client to use any object as a synchronization/communication channel.



3.4. MANDATORY EXCLUSIVE LOCKS

Mandatory Exclusive Locks is a feature that locks an RBD to a single client, if multiple mounts are in place. This helps address the write conflict situation when multiple mounted client try to write to the same object. This feature is built on **`object-watch-notify`** explained in the previous section. So, when writing, if one client first establishes an exclusive lock on an object, another mounted client will first check to see if a peer has placed a lock on the object before writing.

With this feature enabled, only one client can modify an RBD device at a time, especially when changing internal RBD structures during operations like **`snapshot create/delete`**. It also provides some protection for failed clients. For instance, if a virtual machine seems to be unresponsive and you start a copy of it with the same disk elsewhere, the first one will be blacklisted in Ceph and unable to corrupt the new one.

Mandatory Exclusive Locks is not enabled by default. You have to explicitly enable it with **`--image-features`** parameter when creating an image. For example:

```
rbd -p mypool create myimage --size 102400 --image-features 5
```

Here, the numeral **`5`** is a summation of **`1`** and **`4`** where **`1`** enables layering support and **`4`** enables exclusive locking support. So, the above command will create a 100 GB rbd image, enable layering and exclusive lock.

Mandatory Exclusive Locks is also a prerequisite for **`object map`**. Without enabling exclusive locking support, object map support cannot be enabled.

Mandatory Exclusive Locks also does some ground work for mirroring.

3.5. OBJECT MAP

Object map is a feature that tracks the presence of backing RADOS objects when a client writes to an rbd image. When a write occurs, that write is translated to an offset within a backing RADOS object. When the object map feature is enabled, the presence of these RADOS objects is tracked. So, we can know if the objects actually exist. Object map is kept in-memory on the librbd client so it can avoid querying the OSDs for objects that it knows don't exist. In other words, object map is an index of the objects that actually exists.

Object map is beneficial for certain operations, viz:

- Resize
- Export
- Copy
- Flatten
- Delete
- Read

A shrink resize operation is like a partial delete where the trailing objects are deleted.

An export operation knows which objects are to be requested from RADOS.

A copy operation knows which objects exist and need to be copied. It does not have to iterate over potentially hundreds and thousands of possible objects.

A flatten operation performs a copy-up for all parent objects to the clone so that the clone can be detached from the parent i.e, the reference from the child clone to the parent snapshot can be removed. So, instead of all potential objects, copy-up is done only for the objects that exist.

A delete operation deletes only the objects that exist in the image.

A read operation skips the read for objects it knows doesn't exist.

So, for operations like resize (shrinking only), exporting, copying, flattening, and deleting, these operations would need to issue an operation for all potentially affected RADOS objects (whether they exist or not). With object map enabled, if the object doesn't exist, the operation need not be issued.

For example, if we have a 1 TB sparse RBD image, it can have hundreds and thousands of backing RADOS objects. A delete operation without object map enabled would need to issue a **`remove object`** operation for each potential object in the image. But if object map is enabled, it only needs to issue **`remove object`** operations for the objects that exist.

Object map is valuable against clones that don't have actual objects but gets object from parent. When there is a cloned image, the clone initially has no objects and all reads are redirected to the parent. So, object map can improve reads as without the object map, first it needs to issue a read operation to the OSD for the clone, when that fails, it issues another read to the parent — with object map enabled. It skips the read for objects it knows doesn't exist.

Object map is not enabled by default. You have to explicitly enable it with **`--image-features`** parameter when creating an image. Also, **`Mandatory Exclusive Locks`** (mentioned in previous section) is a prerequisite for **`object map`**. Without enabling exclusive locking support, object map support cannot be enabled. To enable object map support when creating a image, execute:

```
rbd -p mypool create myimage --size 102400 --image-features 13
```

Here, the numeral **`13`** is a summation of **`1`**, **`4`** and **`8`** where **`1`** enables layering support, **`4`** enables exclusive locking support and **`8`** enables object map support. So, the above command will create a 100 GB rbd image, enable layering, exclusive lock and object map.

3.6. DATA STRIPING

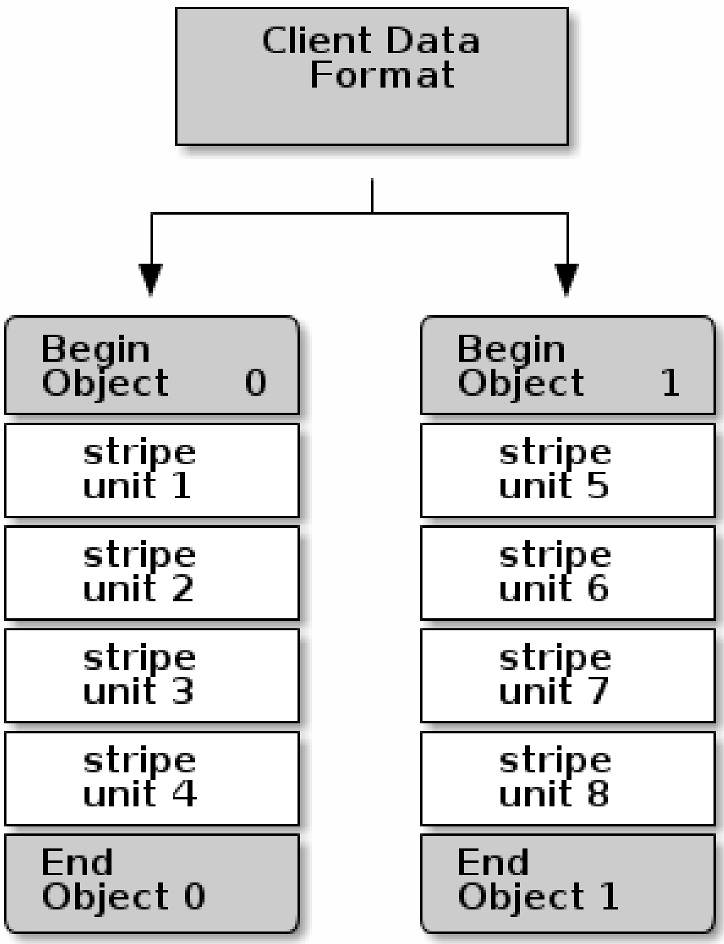
Storage devices have throughput limitations, which impact performance and scalability. So storage systems often support striping—storing sequential pieces of information across across multiple storage devices—to increase throughput and performance. The most common form of data striping comes from RAID. The RAID type most similar to Ceph’s striping is RAID 0, or a 'striped volume.' Ceph’s striping offers the throughput of RAID 0 striping, the reliability of n-way RAID mirroring and faster recovery.

Ceph provides three types of clients: Ceph Block Device, Ceph Filesystem, and Ceph Object Storage. A Ceph Client converts its data from the representation format it provides to its users (a block device image, RESTful objects, CephFS filesystem directories) into objects for storage in the Ceph Storage Cluster.

Tip

The objects Ceph stores in the Ceph Storage Cluster are not striped. Ceph Object Storage, Ceph Block Device, and the Ceph Filesystem stripe their data over multiple Ceph Storage Cluster objects. Ceph Clients that write directly to the Ceph Storage Cluster via **`librados`** must perform the striping (and parallel I/O) for themselves to obtain these benefits.

The simplest Ceph striping format involves a stripe count of 1 object. Ceph Clients write stripe units to a Ceph Storage Cluster object until the object is at its maximum capacity, and then create another object for additional stripes of data. The simplest form of striping may be sufficient for small block device images, S3 or Swift objects. However, this simple form doesn’t take maximum advantage of Ceph’s ability to distribute data across placement groups, and consequently doesn’t improve performance very much. The following diagram depicts the simplest form of striping:

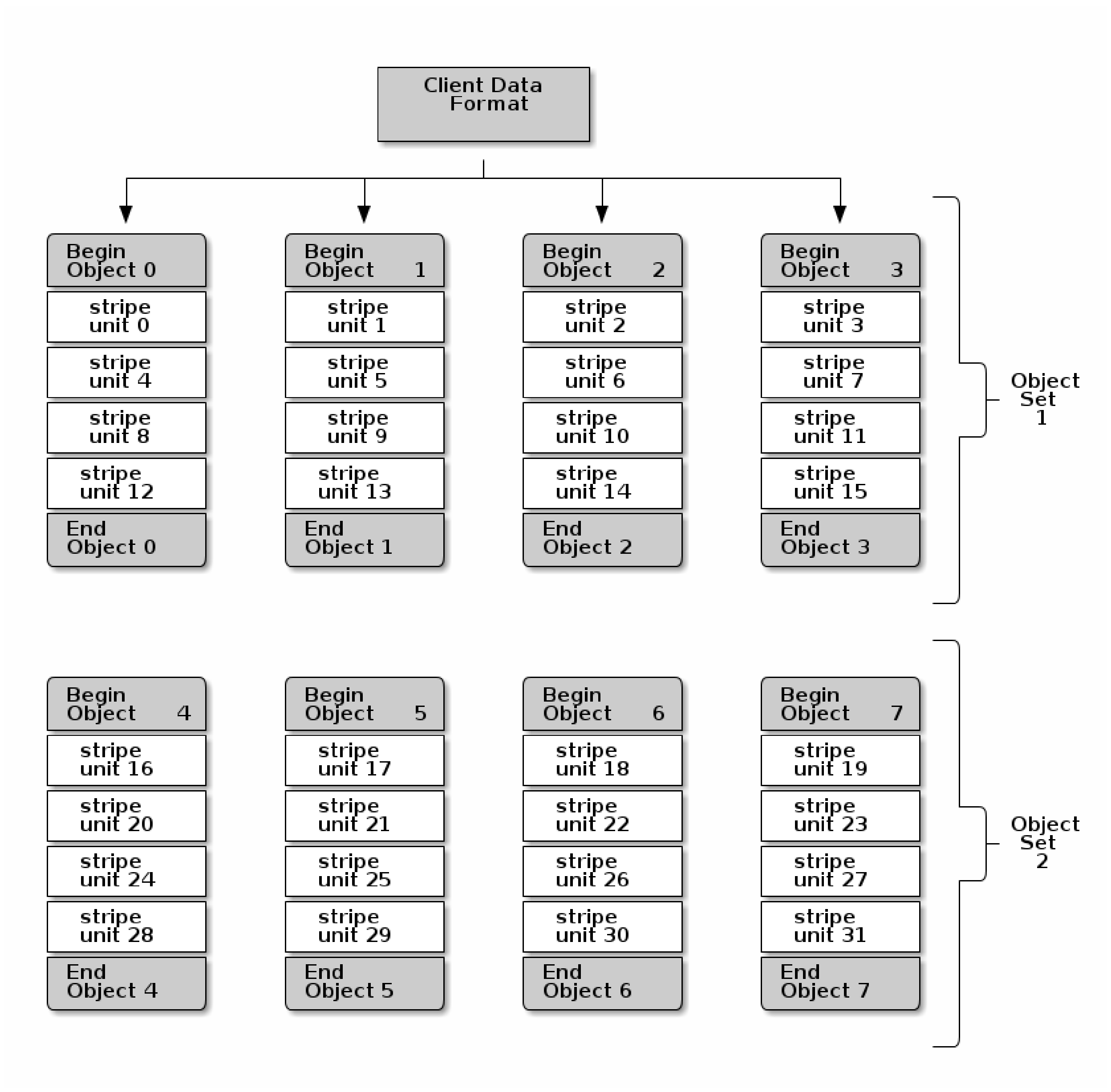


If you anticipate large images sizes, large S3 or Swift objects (e.g., video), you may see considerable read/write performance improvements by striping client data over multiple objects within an object set. Significant write performance occurs when the client writes the stripe units to their corresponding objects in parallel. Since objects get mapped to different placement groups and further mapped to different OSDs, each write occurs in parallel at the maximum write speed. A write to a single disk would be limited by the head movement (e.g. 6ms per seek) and bandwidth of that one device (e.g. 100MB/s). By spreading that write over multiple objects (which map to different placement groups and OSDs) Ceph can reduce the number of seeks per drive and combine the throughput of multiple drives to achieve much faster write (or read) speeds.

Note

Striping is independent of object replicas. Since CRUSH replicates objects across OSDs, stripes get replicated automatically.

In the following diagram, client data gets striped across an object set (**`object set 1`** in the following diagram) consisting of 4 objects, where the first stripe unit is **`stripe unit 0`** in **`object 0`**, and the fourth stripe unit is **`stripe unit 3`** in **`object 3`**. After writing the fourth stripe, the client determines if the object set is full. If the object set is not full, the client begins writing a stripe to the first object again (**`object 0`** in the following diagram). If the object set is full, the client creates a new object set (**`object set 2`** in the following diagram), and begins writing to the first stripe (**`stripe unit 16`**) in the first object in the new object set (**`object 4`** in the diagram below).



Three important variables determine how Ceph stripes data:

- **Object Size:** Objects in the Ceph Storage Cluster have a maximum configurable size (e.g., 2MB, 4MB, etc.). The object size should be large enough to accommodate many stripe units, and should be a multiple of the stripe unit.

- **Stripe Width:** Stripes have a configurable unit size (e.g., 64kb). The Ceph Client divides the data it will write to objects into equally sized stripe units, except for the last stripe unit. A stripe width, should be a fraction of the Object Size so that an object may contain many stripe units.

- **Stripe Count:** The Ceph Client writes a sequence of stripe units over a series of objects determined by the stripe count. The series of objects is called an object set. After the Ceph Client writes to the last object in the object set, it returns to the first object in the object set.

Important

Test the performance of your striping configuration before putting your cluster into production. You CANNOT change these striping parameters after you stripe the data and write it to objects.

Once the Ceph Client has striped data to stripe units and mapped the stripe units to objects, Ceph’s CRUSH algorithm maps the objects to placement groups, and the placement groups to Ceph OSD Daemons before the objects are stored as files on a storage disk.

Note

Since a client writes to a single pool, all data striped into objects get mapped to placement groups in the same pool. So they use the same CRUSH map and the same access controls.