Red Hat Ceph Storage 3

Architecture Guide

Guide on Red Hat Ceph Storage Architecture

Last Updated: 2018-09-07

Red Hat Ceph Storage 3 Architecture Guide

Guide on Red Hat Ceph Storage Architecture

Legal Notice

Abstract

This document provides architecture information for Ceph Storage Clusters and its clients.

Table of Contents

CHAPTER 1. OVERVIEW

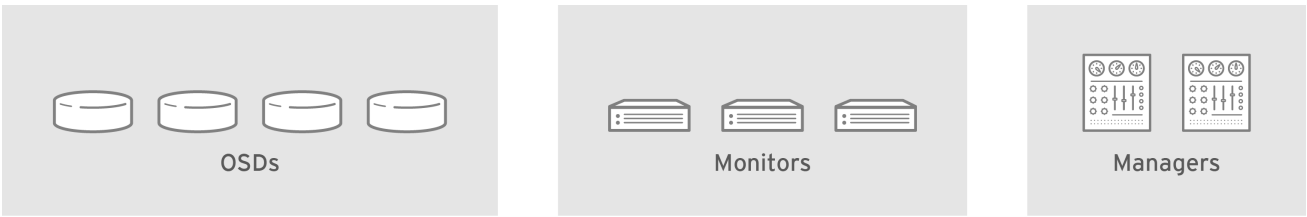
Red Hat Ceph Storage is a distributed data object store designed to provide excellent performance, reliability and scalability. Distributed object stores are the future of storage, because they accommodate unstructured data, and because clients can use modern object interfaces and legacy interfaces simultaneously.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

- APIs in many languages (C/C++, Java, Python)
- RESTful interfaces (S3/Swift)
- Block device interface
- Filesystem interface

The power of Red Hat Ceph Storage 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 Storage 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 three types of daemons:

- **Ceph OSD Daemon:** Ceph OSDs store data on behalf of Ceph clients. Additionally, Ceph OSDs utilize the CPU, memory and networking of Ceph nodes to perform data replication, erasure coding, 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. Monitors require high consistency, and use Paxos to ensure agreement about the state of the Ceph Storage cluster.
- **Ceph Manager:** New in RHCS 3, a Ceph Manager maintains detailed information about placement groups, process metadata and host metadata in lieu of the Ceph Monitor—significantly improving performance at scale. The Ceph Manager handles execution of many of the read-only Ceph CLI queries, such as placement group statistics. The Ceph Manager also provides the RESTful monitoring APIs.



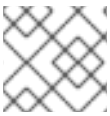
Ceph client interfaces read data from and write data to the Ceph storage cluster. Clients need the following data to communicate with the Ceph storage cluster:

- The Ceph configuration file, or the cluster name (usually **ceph**) and the 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. However, they do not need to maintain an object-to-OSD index or communicate with a centralized object index to look up

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 provide an object name and pool name to **`librados`**, which computes an object's placement group and the primary OSD for storing and retrieving data using the CRUSH (Controlled Replication Under Scalable Hashing) algorithm. 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.

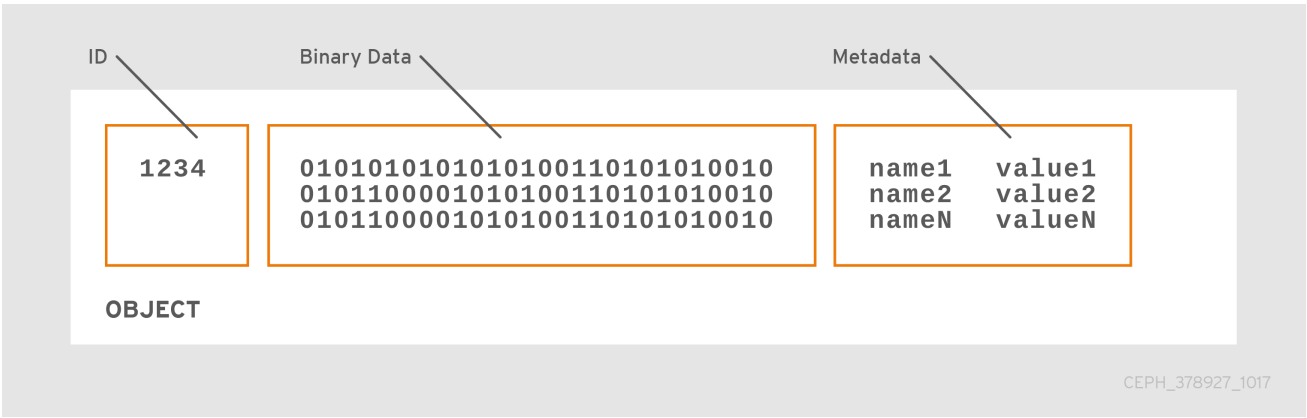
When an OSD stores data, it receives data from a Ceph client—whether the client is a Ceph Block Device, a Ceph Object Gateway, a Ceph Filesystem or another interface—and it stores the data as an object.



NOTE

An object ID is unique across the entire cluster, not just an OSD's storage media.

Ceph OSDs store all data as objects in a flat namespace. There are no hierarchies of directories. An object has a cluster-wide unique identifier, binary data, and metadata consisting of a set of name/value pairs.



Ceph clients define the semantics for the client's data format. 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 can have a large number of Ceph nodes for limitless scalability, high availability and performance. Each node leverages non-proprietary hardware and intelligent Ceph daemons that communicate with each other to:

- Write and read data
- Compress data
- Ensure durability by replicating or erasure coding data
- Monitor and report on cluster health—also called 'heartbeating'
- Redistribute data dynamically—also called 'backfilling'
- Ensure data integrity; and,
- 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, `librados` and the storage cluster perform 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.' Ceph administrators 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 via 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 its data objects.

In actual fact, a Ceph pool is not only a logical partition for storing object data. A pool plays a critical role in how the Ceph storage cluster distributes and stores data. However, 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 to ensure durability. The data durability method is pool-wide, and does not change after creating the pool. The pool type defines the data durability method when creating the pool. Pool types 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. Ceph must handle many types of operations, including data durability via replicas or erasure code chunks, data integrity by scrubbing or CRC checks, replication, rebalancing and recovery. Consequently, managing data on a per-object basis presents a scalability and performance bottleneck. Ceph addresses this bottleneck by sharding a pool into placement groups. The CRUSH algorithm computes the placement group for storing an object and computes the Acting Set of OSDs for the placement group. CRUSH puts each object into a placement group. Then, CRUSH stores each placement group in a set of OSDs. System administrators set the placement group count when creating or modifying a pool.

- **CRUSH Ruleset:** CRUSH plays another important role: CRUSH can detect failure domains and performance domains. CRUSH can identify OSDs by storage media type and organize OSDs hierarchically into nodes, racks, and rows. CRUSH enables Ceph OSDs to store object copies across failure domains. For example, copies of an object may get stored in different server rooms, aisles, racks and nodes. If a large part of a cluster fails, such as a rack, the cluster can still operate in a degraded state until the cluster recovers.

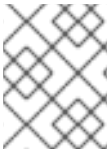
Additionally, CRUSH enables clients to write data to particular types of hardware, such as SSDs, hard drives with SSD journals, or hard drives with journals on the same drive as the data. The CRUSH ruleset determines failure domains and performance domains for the pool. Administrators set the CRUSH ruleset when creating a pool. NOTE: An administrator **CANNOT** change a pool's ruleset after creating 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; that is, copies get distributed to separate physical 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, which authenticates users and daemons.



NOTE

The **cephx** protocol does not address data encryption for data transported over the network or data stored in OSDs.

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 enables both parties to prove to each other that 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)

Storing millions of objects in a cluster and managing them individually is resource intensive. So Ceph uses Placement Groups (PGs) to make managing a huge number of objects more efficient.

A PG is a subset of a pool that serves to contain a collection of objects. Ceph shards a pool into a series of PGs. Then, the CRUSH algorithm takes the cluster map and the status of the cluster into account and distributes the PGs evenly and pseudo-randomly to OSDs in the cluster.

Here is how it works.

When a system administrator creates a pool, CRUSH creates a user-defined number of PGs for the pool. Generally, the number of PGs should be a reasonably fine-grained subset of the data. For example, 100 PGs per OSD per pool would mean that each PG contains approximately 1% of the pool's data.

The number of PGs has a performance impact when Ceph needs to move a PG from one OSD to another OSD. If the pool has too few PGs, Ceph will move a large percentage of the data simultaneously and the network load will adversely impact the cluster's performance. If the pool has too many PGs, Ceph will use too much CPU and RAM when moving tiny percentages of the data and thereby adversely impact the cluster's performance. For details on calculating the number of PGs to achieve optimal performance, see PG Count.

Ceph ensures against data loss by storing replicas of an object or by storing erasure code chunks of an object. Since Ceph stores objects or erasure code chunks of an object within PGs, Ceph replicates each PG in a set of OSDs called the "Acting Set" for each copy of an object or each erasure code chunk of an object. A system administrator can determine the number of PGs in a pool and the number of replicas or erasure code chunks. However, the CRUSH algorithm calculates which OSDs are in the acting set for a particular PG.

The CRUSH algorithm and PGs make Ceph dynamic. Changes in the cluster map or the cluster state may result in Ceph moving PGs from one OSD to another automatically. Here are a few examples:

- **Expanding the Cluster:** When adding a new host and its OSDs to the cluster, the cluster map changes. Since CRUSH evenly and pseudo-randomly distributes PGs to OSDs throughout the cluster, adding a new host and its OSDs means that CRUSH will reassign some of the pool's placement groups to those new OSDs. That means that system administrators do not have to rebalance the cluster manually. Also, it means that the new OSDs contain approximately the same amount of data as the other OSDs. This also means that new OSDs do not contain newly written OSDs, preventing "hot spots" in the cluster.

- **An OSD Fails:** When a OSD fails, the state of the cluster changes. Ceph temporarily loses one of the replicas or erasure code chunks, and needs to make another copy. If the primary OSD in the acting set fails, the next OSD in the acting set becomes the primary and CRUSH calculates a new OSD to store the additional copy or erasure code chunk.

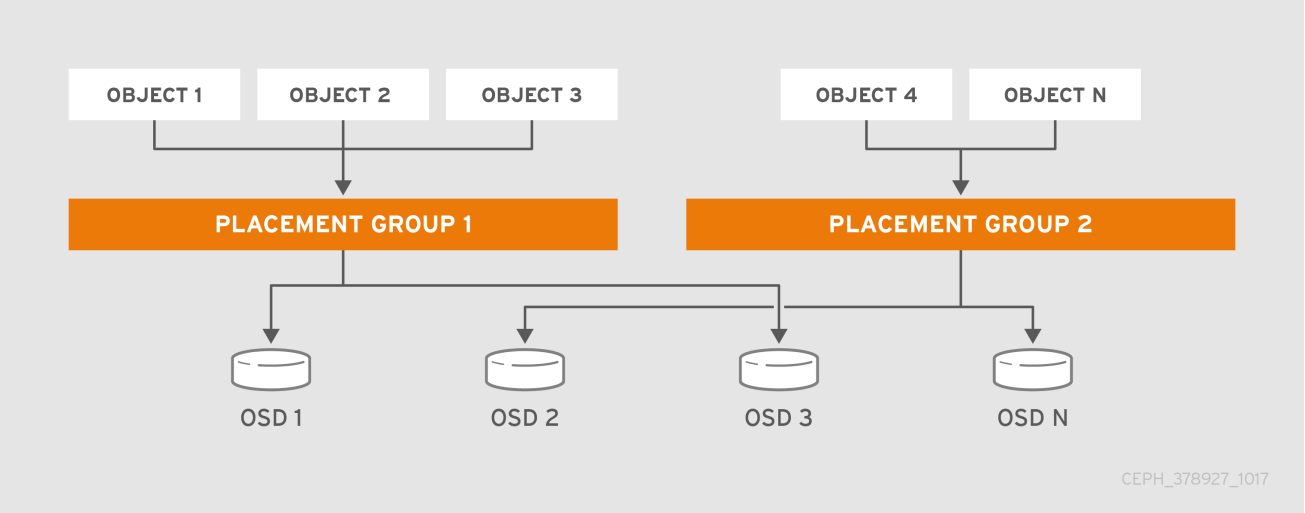
By managing millions of objects within the context of hundreds to thousands of PGs, the Ceph storage cluster can grow, shrink and recover from failure efficiently.

For Ceph clients, the CRUSH algorithm via **`librados`** makes the process of reading and writing objects very simple. A Ceph client simply writes an object to a pool or reads an object from a pool. The primary OSD in the acting set can write replicas of the object or erasure code chunks of the object to the secondary OSDs in the acting set on behalf of the Ceph client.

If the cluster map or cluster state changes, the CRUSH computation for which OSDs store the PG will change too. For example, a Ceph client may write object **`foo`** to the pool **`bar`**. CRUSH will assign the object to PG **`1.a`**, and store it on **`OSD 5`**, which makes replicas on **`OSD 10`** and **`OSD 15`** respectively. If **`OSD 5`** fails, the cluster state changes. When the Ceph client reads object **`foo`** from pool **`bar`**, the client via **`librados`** will automatically retrieve it from **`OSD 10`** as the new primary OSD dynamically.

The Ceph client via **`librados`** connects directly to the primary OSD within an acting set when writing and reading objects. Since I/O operations do not use a centralized broker, network oversubscription is typically NOT an issue with Ceph.

The following diagram depicts how CRUSH assigns objects to PGs, and PGs to OSDs. The CRUSH algorithm assigns the PGs to OSDs such that each OSD in the acting set is in a separate failure domain, which typically means the OSDs will always be on separate server hosts and sometimes in separate racks.



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 within the rule set, 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, access the placement group and read or write object data.

To map placement groups to OSDs, a CRUSH map defines a hierarchical list of bucket types. The list of bucket types are located 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 as 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. Administrators may define it according to their own needs if the default types don't suit their requirements. CRUSH supports a directed acyclic graph that models the Ceph OSD nodes, typically in a hierarchy. So Ceph administrators can support multiple hierarchies with multiple root nodes in a single CRUSH map. For example, an administrator can create a hierarchy representing higher cost SSDs for high performance, and a separate hierarchy of lower cost hard drives with SSD journals for moderate performance.

2.5. I/O OPERATIONS

Ceph clients retrieve a 'Cluster Map' from a Ceph monitor, bind to a pool, and perform i/o on objects within placement groups in the pool. 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 and their current state. **However, the client doesn't know anything about object locations.**

The only inputs required by the client are the object ID and the pool name. It is simple: Ceph stores data in named pools. When a client wants to store a named object in a pool it takes the object name, a hash code, the number of PGs in the pool and the pool name as inputs; then, 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. For example, `pool = liverpool` and `object-id = john`.

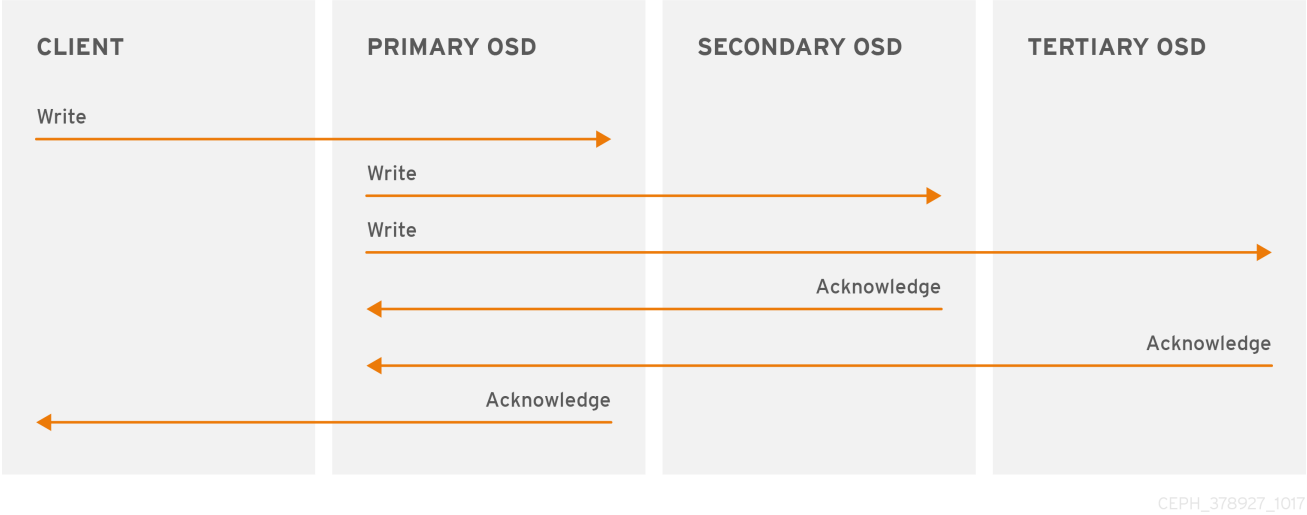
2. CRUSH takes the object ID and hashes it.
3. CRUSH calculates the hash modulo of the number of PGs to get a PG ID. For example, **58**.
4. CRUSH calculates the primary OSD corresponding to the PG ID.
5. The client gets the pool ID given the pool name. For example, the pool "liverpool" is pool number **4**.
6. The client prepends the pool ID to the PG ID. For example, **4.58**.
7. The client performs an object operation such as write, read, or delete 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. Empowering a Ceph client via **librados** to compute object locations is much faster than requiring the client to make a query to the storage cluster 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 data in 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.

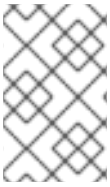
For RHCS 2 and earlier releases, daemons in very large clusters may encounter slower performance when cluster maps grow too large. For example, a cluster with 10k OSDs might have 100 PGs per OSD, leading to ~1M PGs in order to distribute data efficiently—and numerous epochs for the cluster map. Consequently, daemons will use more CPU and RAM in RHCS 2 with very large clusters. For RHCS 3 and later releases, daemons receive the current state of the cluster as in RHCS 2 and earlier releases. However, the Ceph Manager (**ceph-mgr**) daemon now handles queries on PGs, dramatically improving performance at large scales. Red Hat recommends using RHCS 3 and later releases for very large clusters with thousands of OSDs.

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 to store replicas of objects. 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. The value is found in the **osd_pool_default_size** setting. Then, the primary OSD takes the object ID, pool name and the cluster map and uses the CRUSH algorithm to calculate the IDs of secondary OSDs for the acting set. The primary OSD writes the object to the secondary OSDs. When the primary OSD receives an acknowledgment from the secondary OSDs 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. CRUSH computes the IDs of the secondary OSDs with consideration for the failure domains.

2.5.2. Erasure-coded I/O

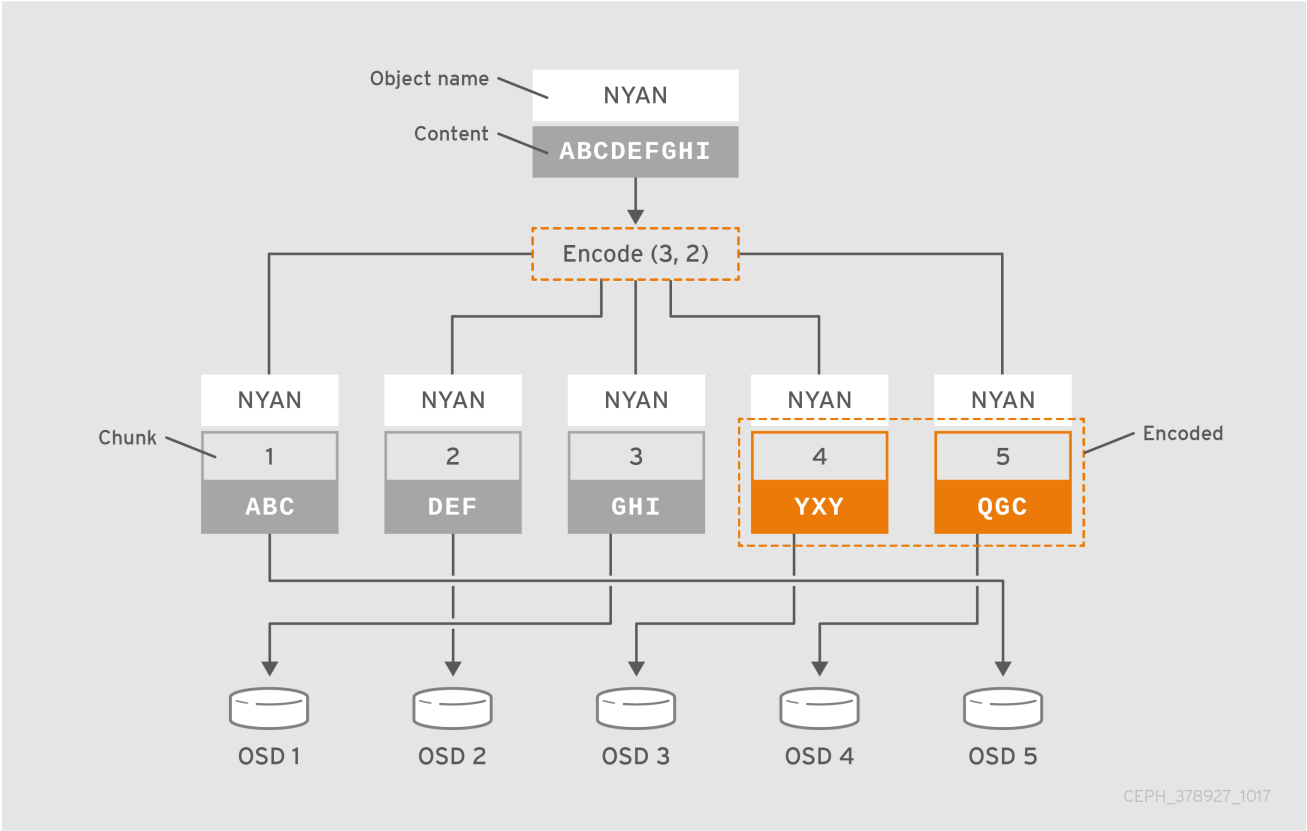
Ceph can load one of many erasure code algorithms. The earliest and most commonly used is the **`Reed-Solomon`** algorithm. An erasure code is actually a forward error correction (FEC) code, which transforms a message of **`K`** chunks into a longer message called a 'code word' of **`N`** chunks, such that Ceph can recover the original message from a subset of the **`N`** chunks.

More specifically, **`N = K+M`** where the variable **`K`** is the original amount of data chunks, the variable **`M`** stands for the extra or redundant chunks that the erasure code algorithm adds 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 the algorithm computes **`N-K`** redundant chunks from **`K`** original data chunks. This approach guarantees that Ceph can access all the original data. The system is resilient to arbitrary **`N-K`** failures. For instance, in a 10 **`K`** of 16 **`N`** configuration, or erasure coding **`10/16`**, the erasure code algorithm adds six extra chunks to the 10 base chunks **`K`**. For example, in a **`M = K-N`** or **`16-10 = 6`** configuration, Ceph will spread the 16 chunks **`N`** across 16 OSDs. The original file could be reconstructed from the 10 verified **`N`** chunks even if 6 OSDs fail—ensuring that the Ceph cluster will not lose data, and thereby 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 OSDs 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 Ceph stores each chunk in an OSD in the acting set. Ceph stores the rank of the chunk 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. The primary OSD is also responsible for maintaining an authoritative version of the placement group logs.

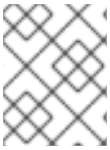
For example, in a typical configuration a system administrator creates an erasure coded pool to use five OSDs and sustain the loss of two of them. That is, (**`K+M = 5`**) such that (**`M = 2`**).

When Ceph writes the object **NYAN** containing **ABCDEFGHI** to the pool, the erasure encoding algorithm 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 algorithm will pad the content 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**. Ceph stores each chunk on an OSD in the acting set, where it stores the chunks in objects that have the same name, **NYAN**, but reside on different OSDs. The algorithm must preserve the order in which it created the chunks as an attribute of the object **shard_t**, in addition to its name. For example, Chunk 1 contains **ABC** and Ceph stores it on **OSD5** while chunk 4 contains **YXY** and Ceph stores it on **OSD3**.



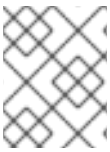
In a recovery scenario, the client attempts to read the object **NYAN** from the erasure-coded pool by reading chunks 1 through 5. The OSD informs the algorithm that chunks 2 and 5 are missing. These missing chunks are called 'erasures'. For example, the primary OSD could not read chunk 5 because the **OSD4** is out, and could not read chunk 2, because **OSD2** was the slowest and its chunk was not taken into account. However, as soon as the algorithm has three chunks, it reads the 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**, and original content of chunk 5, which contained **GQC**.

Splitting data into chunks is independent from object placement. The 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 the erasure code profile creates additional chunks and requires fewer OSDs to recover from. For example, in an **lrc** profile configuration **K=4 M=2 L=3**, the algorithm creates six chunks (**K+M**), just as the **jerasure** plugin would, but the locality value (**L=3**) requiers the algorithm create 2 more chunks locally. The algorithm creates the additional chunks 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, the algorithm only requires 3 chunks for recovery instead of 5. For more information about CRUSH, the erasure-coding profiles, and plugins, see the Storage Strategies Guide for Red Hat Ceph Storage 3.



NOTE

Using erasure-coded pools disables Object Map. For more details on Object Map, see the Object Map section.



NOTE

Red Hat only supports erasure-coded pools with the RADOS Gateway (RGW). Red Hat does not support using erasure-coded pools with a RADOS Block Device (RBD).

2.6. THE OBJECTSTORE INTERFACE

`ObjectStore` provides a low-level interface to an OSD's raw block device. When a client reads or writes data, it interacts with the **`ObjectStore`** interface. Ceph write operations are essentially ACID transactions: that is, they provide **Atomicity**, **Consistency**, **Isolation** and **Durability**. **`ObjectStore`** ensures that a **`Transaction`** is all-or-nothing to provide **Atomicity**. The **`ObjectStore`** also handles object semantics. As noted in the Overview section, an object stored in the storage cluster has a unique identifier, object data and metadata. So **`ObjectStore`** provides **`Consistency`** by ensuring that Ceph object semantics are correct. **`ObjectStore`** also provides the **Isolation** portion of an ACID transaction by invoking a **`Sequencer`** on write operations to ensure that Ceph write operations occur sequentially. By contrast, an OSDs replication or erasure coding functionality provides the **Durability** component of the ACID transaction. Since **`ObjectStore`** is a low-level interface to storage media, it also provides performance statistics.

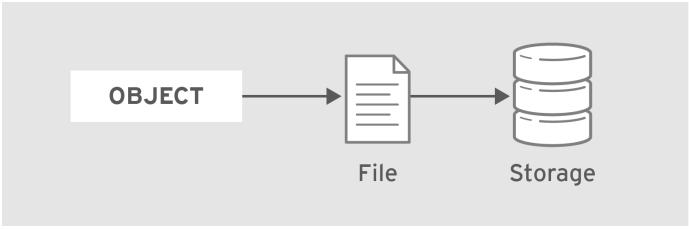
Ceph implements several concrete methods for storing data:

- **FileStore:** A production grade implementation using a filesystem to store object data.
- **BlueStore:** A production grade implementation using a raw block device to store object data.
- **Memstore:** A developer implementation for testing read/write operations directly in RAM.
- **K/V Store:** An internal implementation for Ceph's use of key/value databases.

Since administrators will generally only address **`FileStore`** and **`BlueStore`**, the following sections will only describe those implementations in greater detail.

2.6.1. FileStore

`FileStore` is one of the original storage implementations for Ceph, and is the most widely used implementation. When the Ceph project began in 2004, Ceph relied exclusively on hard disk drives for storage, as there was no market or even economic feasibility for solid state drives or non-volatile memory over PCI express. Rather than interacting directly with a raw block device, **`FileStore`** interacts with a filesystem, usually **`xfs`**. When **`ObjectStore`** handles an object's semantics and passes them to **`FileStore`**, **`FileStore`** treats placement groups as directories, objects as files, and metadata as XATTRs or **`omap`** entries.



FileStore provides the advantages of leveraging open source file system semantics and the ability to journal transactions on separate drives. **Filestore** also has some disadvantages. Among the disadvantages of **FileStore** is that Ceph write operations are essentially ACID transactions. To achieve **Atomicity**, Ceph **FileStore** journals all **Transactions** before writing data. When using the same drive for journaling and writing data, this introduces significant write latency—a double write penalty. Developers assumed that the **btrfs** file system would eventually be the default filesystem format, because it had transactions, copy-on-write semantics and could journal and write data simultaneously. However, **btrfs** never met the Ceph project's reliability requirements for production systems. Consequently, **FileStore** typically uses the **ext4** and **xfs** file systems in lieu of **btrfs**.

One of the short comings of **ext4** is that it has very limited storage for XATTRs—about 4k. Hence, **xfs** became the preferred filesystem for **FileStore**, because it has greater storage for XATTRs among other reasons—about 64k. However, 64k is a limitation for objects that require larger metadata, such as movies that might have large thumbnail pictures. **ObjectStore** and **FileStore** were extended with an **ObjectMap** or **omap**, which uses essentially the same semantics as XATTRs but have unbounded storage to overcome the limitations of XATTRs.

When commercial support for Ceph was launched in 2012, SSDs were still very expensive; however, the ability to journal write operations on an SSD drive and store object data on a fast SAS hard disk drive provided performance characteristics suitable for high throughput and high i/o workloads, such as storing volumes and images for OpenStack. While journaling to SSDs provided substantial performance improvements, **xfs** still journals transactions before writing them. So the double write penalty persists in **FileStore**, even though SSDs make the penalty less onerous.

FileStore treats placement groups as directories. For an initial cluster, it presents few if any problems. However, Ceph clusters tend to grow. As administrators add new nodes and OSDs, placement group counts must increase. With **FileStore**, object data resides as files within directories, the directories representing placement groups. So when increasing the number of placement groups, object data within files has to move to different directories. Ceph distributes objects pseudo-randomly using a 32-bit hash algorithm, a portion of which is incorporated into an object's file name. This is an inefficient approach to addressing and redistributing data.

2.6.2. BlueStore

NOTE

The BlueStore feature is a Technology Preview and as such it is not fully supported yet.

BlueStore is the next generation storage implementation for Ceph. As the market for storage devices now includes solid state drives or SSDs and non-volatile memory over PCI Express or NVMe, their use in Ceph reveals some of the limitations of the **FileStore** storage implementation. While **FileStore** has many improvements to facilitate SSD and NVMe storage, other limitations remain; among them, increasing placement groups remains computationally expensive, and the double write penalty remains. Whereas, **FileStore** interacts with a file system on a block device, **BlueStore** eliminates that layer of indirection and directly consumes a raw block device for object storage. **BlueStore** uses the very light weight **BlueFS** file system on a small partition for its k/v databases. **BlueStore** eliminates the paradigm of a directory representing a placement group, a file representing an object and file XATTRs representing metadata. **BlueStore** also eliminates the double write penalty of **FileStore**, so write operations are nearly twice as fast with **BlueStore** under most workloads.

BlueStore stores data as:

- **Object Data:** In **BlueStore**, Ceph stores objects as blocks directly on a raw block device. The

portion of the raw block device that stores object data does NOT contain a filesystem. The omission of the filesystem eliminates a layer of indirection and thereby improves performance. However, much of the **BlueStore** performance improvement comes from the block database and write-ahead log.

- **Block Database:** In **BlueStore**, the block database handles the object semantics to guarantee **Consistency**. An object's unique identifier is a key in the block database. The values in the block database consist of a series of block addresses that refer to the stored object data, the object's placement group, and object metadata. The block database may reside on a **BlueFS** partition on the same raw block device that stores the object data, or it may reside on a separate block device, usually when the primary block device is a hard disk drive and an SSD or NVMe will improve performance. The block database provides a number of improvements over **FileStore**; namely, the key/value semantics of **BlueStore** do not suffer from the limitations of filesystem XATTRs; and, **BlueStore** may assign objects to other placement groups quickly within the block database without the overhead of moving files from one directory to another, as is the case in **FileStore**. **BlueStore** also introduces new features. The block database can store the checksum of the stored object data and its metadata, allowing full data checksum operations for each read, which is more efficient than periodic scrubbing to detect bit rot. **BlueStore** can compress an object and the block database can store the algorithm used to compress an object—ensuring that read operations select the appropriate algorithm for decompression.

- **Write-ahead Log:** In **BlueStore**, the write-ahead log ensures **Atomicity**, similar to the journaling functionality of **FileStore**. Like **FileStore**, **BlueStore** logs all aspects of each transaction. However, the **BlueStore** write-ahead log or WAL can perform this function simultaneously, which eliminates the double write penalty of **FileStore**. Consequently, **BlueStore** is nearly twice as fast as **FileStore** on write operations for most workloads. BlueStore can deploy the WAL on the same device for storing object data, or it may deploy the WAL on another device, usually when the primary block device is a hard disk drive and an SSD or NVMe will improve performance.

NOTE

It is only helpful to store a block database or a write-ahead log on a separate block device if the separate device is faster than the primary storage device. For example, SSD and NVMe devices are generally faster than HDDs. Placing the block database and the WAL on separate devices may also have performance benefits due to differences in their workloads.

2.7. 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. By using CRUSH to assign 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.

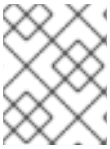
2.7.1. Heartbeating

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—for example, 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, heartbeating also empowers Ceph OSDs to determine if a neighboring OSD is **down**, to update the cluster map and to report it to the Ceph monitors. This means that Ceph monitors can remain light weight processes.

2.7.2. Peering

Ceph stores copies of placement groups on multiple OSDs. Each copy of a placement group has a status. These OSDs "peer" or check each other to ensure that they agree on the status of each copy of the 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*. The *Primary* that stores the first copy of a placement group is responsible for coordinating the peering process for that placement group. The *Primary* is the **ONLY** OSD that that will accept client-initiated writes to objects for a given placement group where it acts as the *Primary*.

An *Acting Set* is a series of OSDs that are responsible for storing a placement group. 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.

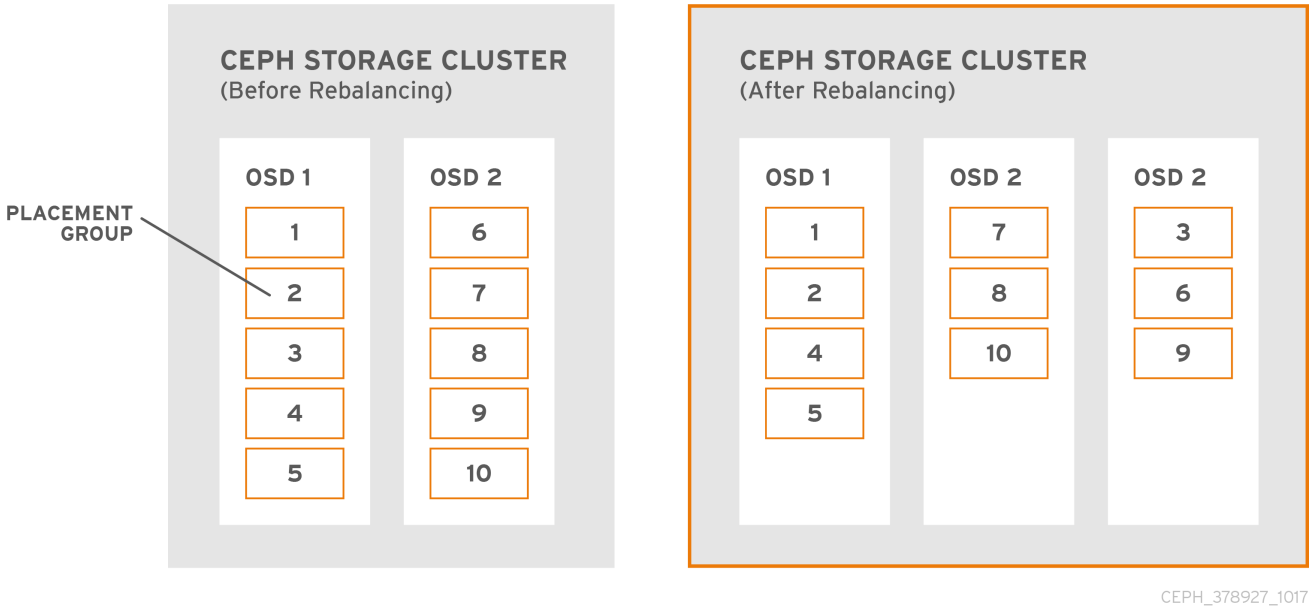
NOTE

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 Ceph will remove **`osd.25`** from the *Up Set*.

2.7.3. Rebalancing and Recovery

When an administrator adds a Ceph OSD to a Ceph storage cluster, Ceph updates the cluster map. This change to the cluster map also changes object placement, because the modified cluster map changes an input for the CRUSH calculations. CRUSH places data evenly, but pseudo randomly. So only a small amount of data moves when an administrator adds a new OSD. The amount of data is usually the the number of new OSDs divided by the total amount of data in the cluster. For example, in a cluster with 50 OSDs, 1/50th or 2% of the data might move when adding an OSD.

The following diagram depicts the rebalancing process 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 the cluster rebalances.



2.7.4. Ensuring Data Integrity

As part of maintaining data integrity, Ceph provides numerous mechanisms to guard against bad disk sectors and bit rot.

- **Scrubbing:** 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 storage 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.

- **CRC Checks:** In RHCS 3 when using `BlueStore`, Ceph can ensure data integrity by conducting a cyclical redundancy check (CRC) on write operations; then, store the CRC value in the block database. On read operations, Ceph can retrieve the CRC value from the block database and compare it with the generated CRC of the retrieved data to ensure data integrity instantly.

2.8. 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.

2.8.1. 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, Red Hat recommends `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.

2.8.2. 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. That is, 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 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.

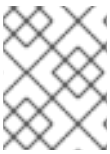
Administrators usually deploy Ceph with an odd number of monitors so that determining a majority is efficient. For example, a majority may be 1, 2:3, 3:5, 4:6, and so forth.

2.8.3. CephX

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

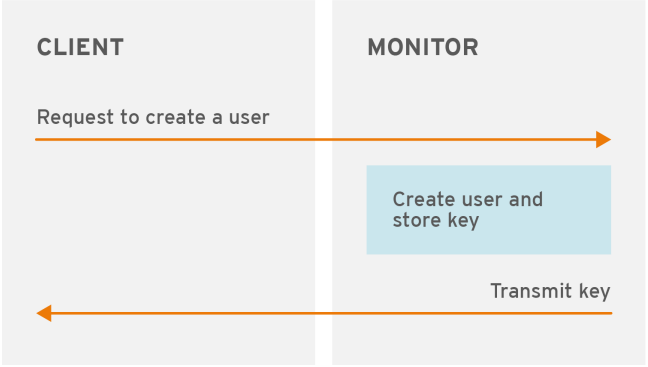
A user/actor invokes a Ceph client to contact a monitor. Unlike Kerberos, each monitor can authenticate users and distribute keys, so there is no single point of failure or bottleneck when using **cephx**. The monitor returns an authentication data structure similar to a Kerberos ticket that contains 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 monitors. 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
CEPH_378927_0917

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.

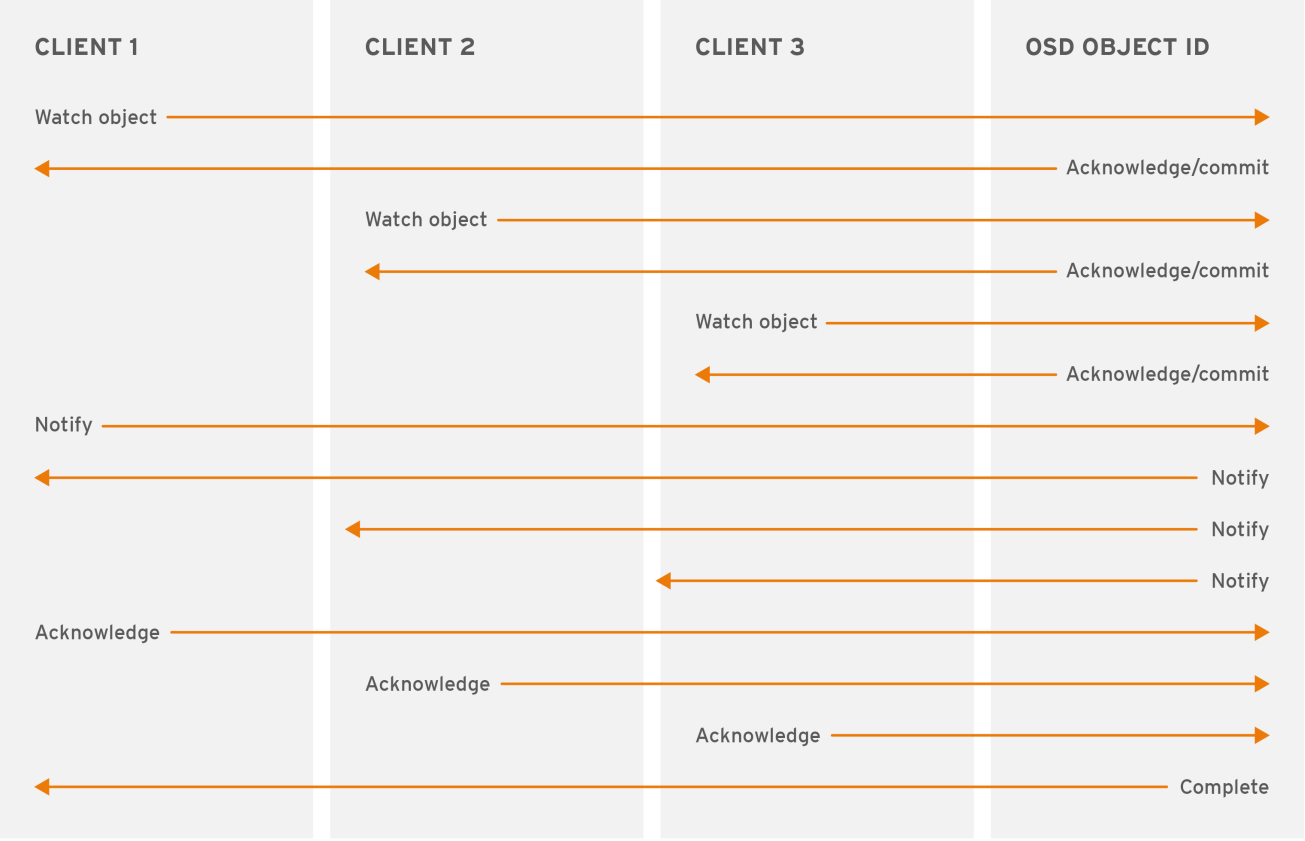
3.1. NATIVE PROTOCOL AND LIBRADOS

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

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.



3.3. 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.

3.4. 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.

3.5. DATA STRIPING

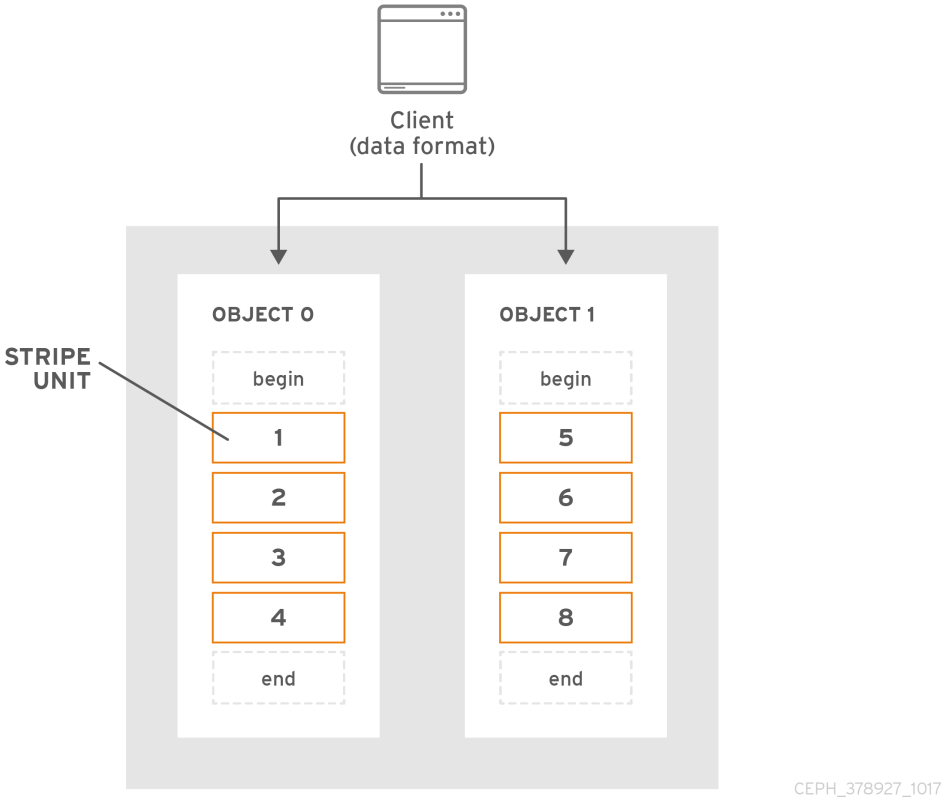
Storage devices have throughput limitations, which impact performance and scalability. So storage systems often support striping—storing sequential pieces of information across across multiple storage devices—to increase throughput and performance. The most common form of data striping comes from RAID. 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.

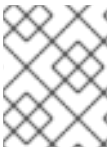
TIP

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:



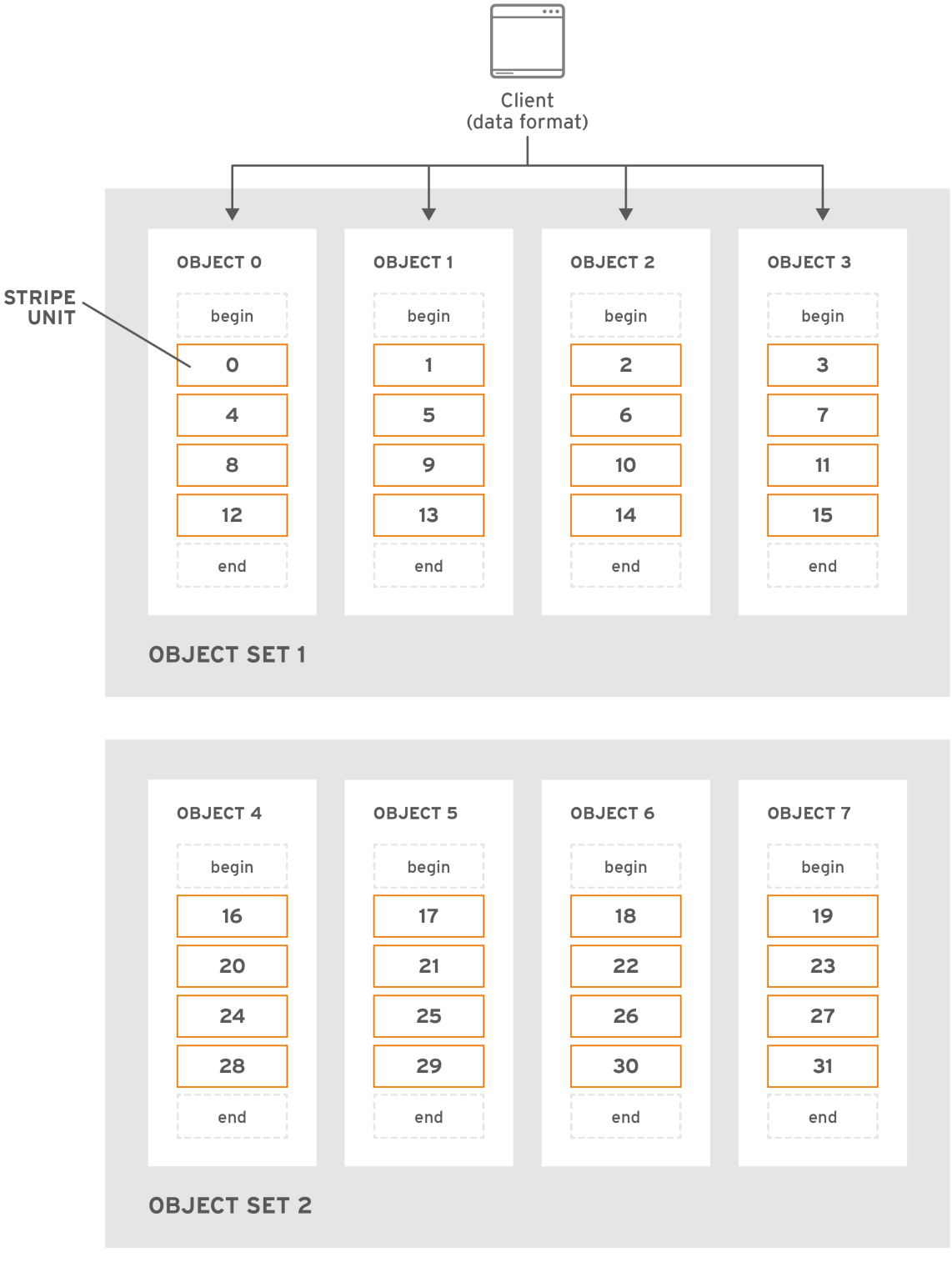
If you anticipate large images sizes, large S3 or Swift objects (e.g., video), you may see considerable read/write performance improvements by striping client data over multiple objects within an object set. Significant write performance occurs when the client writes the stripe units to their corresponding 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).

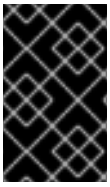


Three important variables determine how Ceph stripes data:

- **Object Size:** Objects in the Ceph Storage Cluster have a maximum configurable size (2MB, 4MB, etc.). The object size should be large enough to accommodate many stripe units, and should be a multiple of the stripe unit. Red Hat recommends a safe maximum value of 16MB.

- **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.

CHAPTER 4. ENCRYPTION

About LUKS Disk Encryption and its Benefits

You can use the Linux Unified Key Setup-on-disk-format (LUKS) method to encrypt partitions on the Linux system. LUKS encrypts the entire block devices and is therefore well-suited for protecting the contents of mobile devices such as removable storage media or laptop disk drives.

Use the **ceph-ansible** utility to create encrypted OSD nodes to protect data stored on them. For details, see the Installing a Red hat Ceph Storage Cluster section in the Red Hat Ceph Storage 3 Installation Guide for Red Hat Enterprise Linux.

For details on LUKS, see the Overview of LUKS section in the Security Guide for Red Hat Enterprise Linux 7.

How ceph-ansible Creates Encrypted Partitions

During the OSD installation, **ceph-ansible** calls the **ceph-disk** utility that is responsible for creating encrypted partitions.

The **ceph-disk** utility creates a small **ceph lockbox** partition in addition to the data (**ceph data**) and journal (**ceph journal**) partitions. Also, **ceph-disk** creates the **cephx client.osd-lockbox** user. The **ceph lockbox** partition contains a key file that **client.osd-lockbox** uses to retrieve the LUKS private key needed to decrypt encrypted **ceph data** and **ceph journal** partitions.

Then, **ceph-disk** calls the **cryptsetup** utility that creates two **dm-crypt** devices for the **ceph data** and **ceph journal** partitions. The **dm-crypt** devices use the **ceph data** and **ceph journal** GUID as an identifier.

How ceph-ansible Handles the LUKS Keys

The **ceph-ansible** utility stores the LUKS private keys in the Ceph Monitor key-value store. Each OSD has its own key for decrypting the **dm-crypt** devices containing the OSD data and the journal. The encrypted partitions are decrypted on boot automatically.