Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor thesis

Suitability analysis of Object Storage for HPC workloads

by Lars Thoms

Scientific Computing
Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
Universität Hamburg

Course of studies:	Informatics
Matriculation number:	6415095
First reviewer:	Dr. Michael Kuhn
Second reviewer:	Prof. Dr. Thomas Ludwig
Advisor:	Dr. Michael Kuhn

Hamburg, 2017-03-23

Contents

1 Abstract

This bachelor thesis reviews the possibility of using an *Object Storage* system like **Ceph Object Storage** (RADOS) especially about its performance and functionality of partial rewrite.

Scientific high-performance computing produces large file objects and its metadata has to be fast searchable. That is why *Object Storages* are a good solution because they store data efficiently with simple API calls without the requirement to comply with *POSIX* specification. Unfortunately, these are overloaded and not performant.

Above all, object storing in combination with metadata separation to store them in a search-efficient database will increase the performance of searching.

Furthermore, per definition objects are supposed to be immutable, but if *RADOS* API calls are used, they are mutable and can be rewritten like on other filesystems.

In this thesis, I am going to investigate whether that objects could be segmented rewritten. Accordingly, I am going to program a *FUSE* driver as a proof of concept and prepare a series of measurement to show performance and issues.

Thereby, it is possible to use *Ceph* as normal Filesystem, because of mutable objects. Unfortunately, the write performance of this driver was low (around 3 MiB/s).

At the end, there is a design concept of an *HPC* application using a *Ceph* cluster in combination with a document-oriented database to store metadata.

2 Introduction

In recent years, it has become clear that scientific research increasingly uses more sophisticated computer simulations and models to achieve newer, better and more precise research results. For instance developing a new pharmaceutical is nearly unimaginable without modeling and simulation molecules to understand and proof the dynamics and kinetics of the pharmaceutical. However, most of the pharmacokinetics software[1] runs on a single computer with above-average computer hardware.

Furthermore, other disciplines like weather and climate research even require entire high-performance clusters to calculate their models. During their calculation, they produce a lot of data output, which has to be stored on disk. Moreover, during the last years, clusters get more efficient and powerful, that is why models could be calculated more precisely, which in turn produces more data to store.

At this point, storage systems become more and more necessary. Plugging in a single hard drive disk is no longer enough because the current throughput of an *HPC* application is about 15 GB/s [11, pp. 122] and rising. To manage, store and search this amount of information in real-time it is necessary to distribute files or parts of it to many nodes (computers).

A standard technology for this kind of problem is called **Storage Area Network** (SAN). It consists of many nodes, each of them contains a few drives (typically from 12-24 pieces)[2], which are connected to each other (usually via fiber).

Depending on requirements and configurations, file objects are stored redundantly distributed over several drives and nodes. Each node has to be configured to deal with their disks. So they need a filesystem or object storage to save the file(-parts) to an interpretable format.

In this bachelor thesis I am going to analyze *Ceph Object Storage* (RADOS) as a representative of the *Object Storage Systems*. The central question is whether they are capable of doing partial updates to existing objects, how this works and if there are performance issues.

That is why I describe the functionality and features of this storage system at first. Followed by an implementation of a *FUSE driver* as a proof of concept and a series of measurements while using my implementation as a reference.

2.1 Motivation

Why using an *Object Storage System* instead of a legacy storage system like *Lustre* or other *POSIX* related solutions?

The main goal of a file storage solution in *HPC* is to dump huge measure data files. It is not necessary to use an overloaded *File I/O API* like *POSIX*, which often lead to a lack of performance. Furthermore, many features like directory hierarchy, extended attributes are not needed. Only a naive, simple storage system, where many large files are put in, is required.

[1] List of pharmacokinetics software: `https://www.pharmpk.com/soft.html` (retrieved on 26/10/2016)

[2] On 2U in a 19" rack; 12 pieces correspond to 3.5" drives, while more to 2.5" drives; ex. `https://www.thomas-krenn.com/en/products/storage-systems/jbod-systems.html` (retrieved on 26/10/2016)

Additional problems to use *POSIX* for distributed filesystems are the strict consistency requirements. Most operations have to be done atomically and the results of each write operation have to be visible to all after returning. Moreover, every access to a file is noted with a timestamp in files property `atime` which lead to an extra write operation for a read operation. The possibility of synchronous and asynchronous operations is available, but unfortunately, it has to be declared at mount time and is not decidable by a user, who has different requirements [10, p. 42-43]. These facts are problematic for distributed filesystems because they decrease performance and demand mechanism like locks to ensure *POSIX* requirements.

Another important point is the separation of metadata into a powerful database, instead of filing it into a normal storage. Therefore, files could be sorted, filtered and found more efficiently than crawling a whole filesystem.

These aspects may be met with an *Object Storage System* like *Ceph*.

3 Storage techniques

3.1 Filesystems

Let us begin with the primary question: how should a program save its data on a storage medium, like hard disk drive, solid-state drive or tape drive? Furthermore, how should they read the stored data?

Andrew S. Tanenbaum wrote that storing information has to pass three requirements [1, pp. 263]:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

One possible solution is to organize the data, in which self-contained data are seen as a file, in a classical local filesystem. They even do not need to be sophisticated because data could be written consecutively on a storage device and signal begin and end of the files with a magic flag (see figure 1).

Figure 1: Simple filesystem

Storing information in this way is strongly disadvantageous because to separate files all data has to be iterated through, and there is no additional information.

Despite this, standards are required to interfere with different file systems or storage solutions – the important one is called **POSIX**[3]. It describes which metadata has to been stored and functions, who read this information, have to been implemented at least. As well *POSIX* requires a hierarchical directory-/filestructure. Moreover, *POSIX* has specifications, which lead to performance issues (as described in 2.1).

As an example, the command `ls -la` has to display several properties as seen in figure 2, which are described in [IEEE [5], vol. “Shell & Utilities”; chap. “Utilities”; par. “ls”].

3.1.1 Storage management

Filesystems usually partition their available space into several areas with specific tasks. The first blocks contain information about the partition. In figure 3 the data layout of ext4, a widely-used filesystem, can be seen [25][17, pp. 227].

[3]Portable Operating System Interface, specified by the IEEE Computer Society

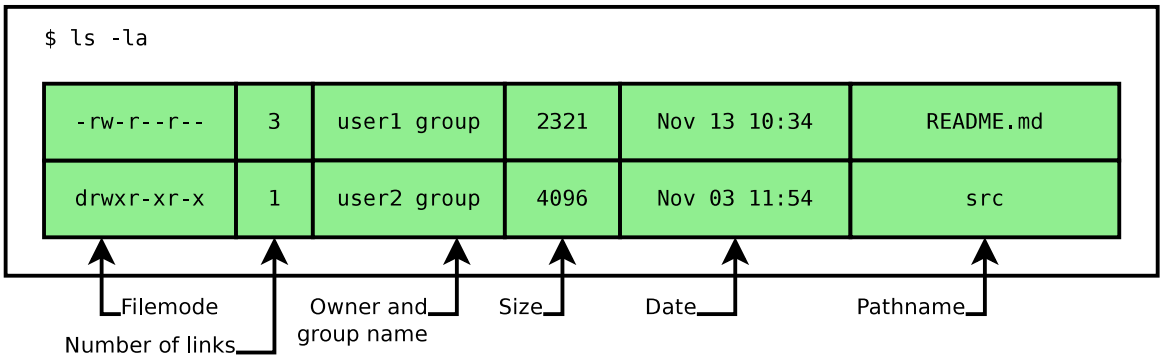


Figure 2: List content of directory (extract)

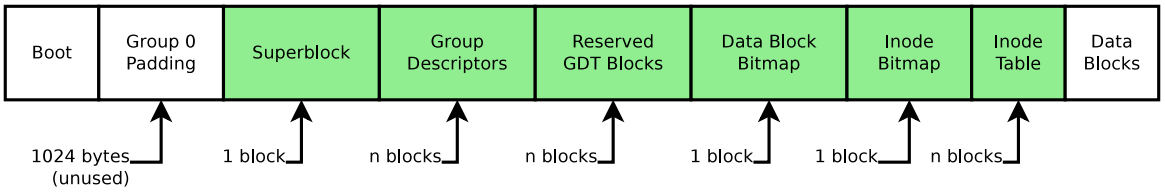


Figure 3: Data layout of ext4

The **Superblock** logs data like the number of available blocks, supported features and amount of inodes. Each group of blocks starts with a **Group Descriptor** with flags about them. Followed by a negligible reserved block and the **Data Block Bitmap**, tracking the usage of data blocks, and **Inode Bitmap**, recording used inode entries.

The last block in the range of metadata blocks is called **inode table**. It contains all **inodes** – each of them describe a file with attributes like permissions, ownership, create- and modified dates and a limited map of pointers, directing to the data block via **LBA**[4].

Thereby two possible address destinations exist: blocks and extents. Addressing blocks is a big drawback caused by limited address space, thus constant file size. By contrast, extents consist of many contiguous blocks, which allow huge files. According to this, fragmentation to some extent is intended.

3.2 Distributed filesystems

At some point, a single storage server is no longer sufficient. Because of reasons like overall storage capacity, read/write throughput or reducing the probability of failure, which is needed in *HPC*. Another reason is the requirement of geographical redundancy if extra data integrity is required.

Thereby many nodes (computers) are connected via *Ethernet* or *Infiniband* to each other directly or indirectly and provide a storage network. From the user perspective, this network is transparent, and they only have to connect with a special client to a gateway server (see figure 4).

There are various methods to store files on different disks or nodes. One simple possibility is to store the whole files on distinct nodes, where the disk configuration is a simple, not redundant logical volume. It is important to note that read/write throughput for a single file is not increased *(sequential access)* –

[4]Logical Block Address

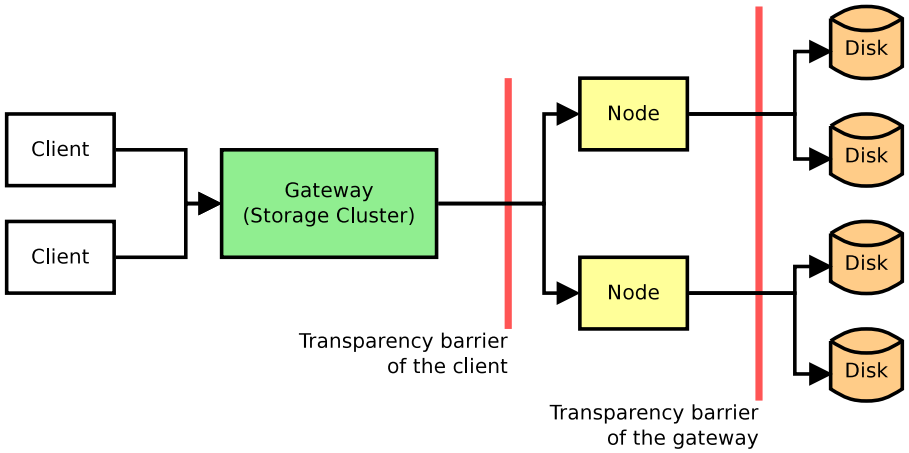


Figure 4: Client connects to transparent storage cluster

disadvantageous for large scientific files. Furthermore, filesize is limited to the greatest free space on a node.

A better solution is to stripe files over many nodes to increase read/write throughput *(parallel access)* and maximum filesize. On top of that, parities reduce the probability of failure.

Underneath the nodes, there are physical disks or similar storage devices, which contain a filesystem to store file(-parts). It is common to use good long-proven solutions like *ZFS*, *EXT4* or *XFS*.

Figure 5: Files distributed over nodes without (left) and with striping (right)

Nowadays there exist many different implementations and products, here is a short overview of the popular ones:

- **GlusterFS**[5]
  An open source software, licensed under *GNU GPL v3*, which works with common hardware configurations. Initially developed by Gluster Inc. and currently maintained by Red Hat Inc. [16] Technically, it works without dedicated *metadata server*, and its modules are loaded via *FUSE*[6]. Therefore it is *POSIX* compatible. [3]
- **BeeGFS**[7]
  A high-performance parallel filesystem developed by *Fraunhofer-Institut für Techno- und Wirtschafts-mathematik*, original named *FhGFS*.
  Its server modules also run in userspace, but unlike GlusterFS metadata of files are stored separately to increase lookup performance. Furthermore, a special client is required. [9]
- **Lustre**[8]
  Similar to *GlusterFS* it is an open-source software, licensed under *GNU GPL v2*, for high-

[5]https://gluster.org
[6]Filesystem in Userspace
[7]https://beegfs.com
[8]http://lustre.org

performance parallel storing. Likewise, it uses separated metadata storages and is compatible with *POSIX*. [12]

3.2.1 Architecture of Lustre (example)

The architecture of distributed filesystems may vary in their implementation, but the fundamental techniques are very similar. Most of them split metadata and object data, use distributed nodes with storage devices, which contains common filesystems like *ZFS*.

Lustre is widely-used, and its essential architecture is a good example to show how these kind of storage systems are created.

The main philosophy of *Lustre* is the disjunction of **servers** and **targets**. Technically, a server provides a service, without storing information on its system. This part is taken by a target. Although a target could be connected to more than one server (failover or balancing, as seen in figure 6).

There are different services, which has to run to form a cluster: **MGS**, **MDS** and **OSS**.

- **Management Server (MGS)**
  This service is responsible for storing and providing all information about its cluster in the **Management Target (MGT)**.
- **Metadata Server (MDS)**
  All metadata of are saved in the **Metadata Target (MDT)**. Names and directories of the Lustre filesystem are also stored.
- **Object Storage Server (OSS)**
  They provide I/O services to the **Object Storage Targets (OST)**, which contains the desired data.

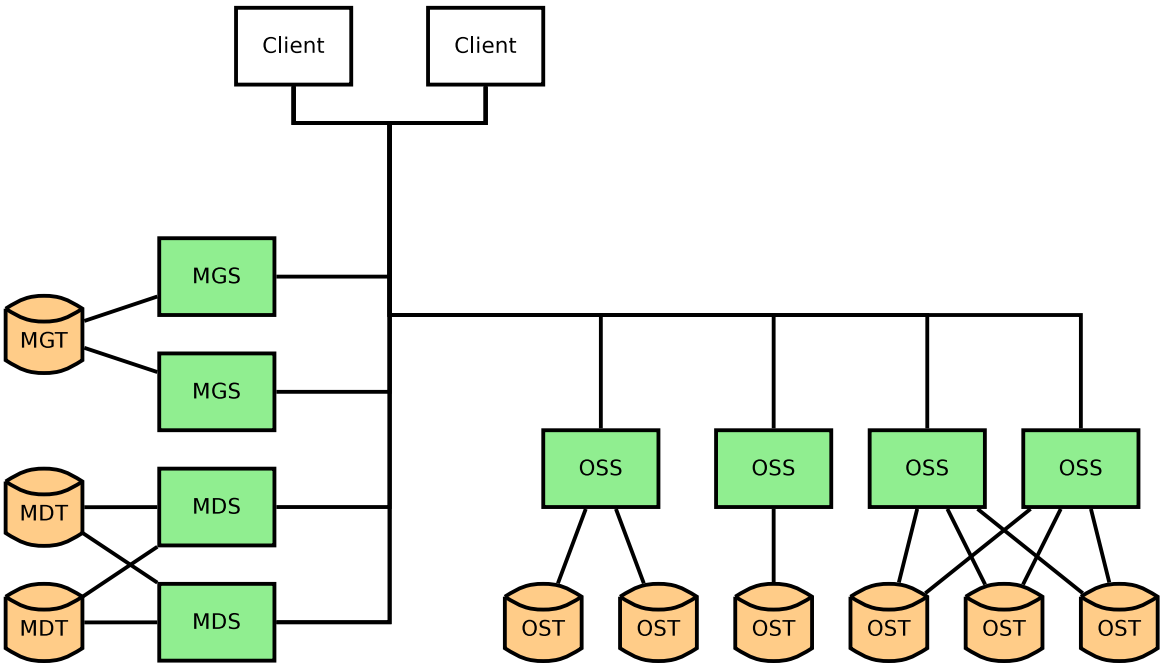


Figure 6: Layout of a typical Lustre cluster (adapted from [12])

3.3 Object storages

The difference between *Object Storages* and *Distributed Filesystems* considered on a technological level is not that huge. It is more like another abstraction of data management. Instead of saving files with a filename, ownership and filetype in a hierarchical directory tree there only exist an object, a collection of metadata and an object identifier. Besides, objects are typically **immutable** which means object storages are intended for archiving or sharing static data.

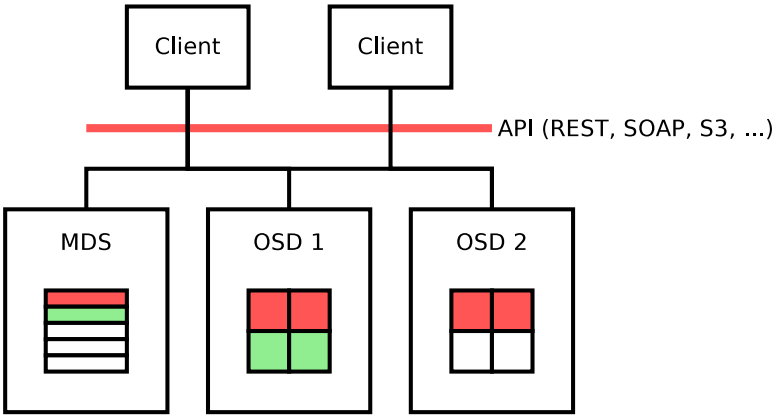


Figure 7: Abstraction of Object Storages

Figure 7 shows a simple scheme. As seen in *Distributed Filesystems* also *Object Storages* divide metadata and binary data. Therefore some *Metadata Server* (MDS) and many *Object Storage Devices* (OSD) exist. Furthermore, an *OSD* can contain many disks or other storage-like devices, configured in a *RAID* or something else and formatted with a normal filesystem (e.g. *EXT4*). In addition, objects are usually striped (redundantly) over many *OSDs* to increase read performance in case of load peaks or failure.

However, the configuration of the *OSDs* or the distribution management is opaque. The users can only push, get or list their objects via an API like *REST*, *SOAP* and *S3*.

At the end of the day, Object Storages write files on a filesystem. Only without hierarchy and possibility of updates, but with a simplified API. [18]

Although an object could contain everything: a database, a filesystem or just a gallery of pictures. That means storing objects is more flexible for storage clusters, because of the fact that they are immutable. No locking features, fragmentation or reallocation is needed. The application behind an object is handling them and implements fitted algorithms to work with them. Besides, numerous applications do not require filenames and file hierarchy.

For example, rich-text documents like `DOCX` or `ODT` are already objects. They contain a filesystem, directories with pictures, graphs, spreadsheets and many `XML` files.

4 Ceph

One of the well-known software packages, providing storage clusters, is *Ceph*. Initially it was developed at the *University of California*, Santa Cruz, by Sage Weil and his working group in 2007 as his dissertation [23]. Later on, he founded Inktank Storage to enhance and support his filesystem. In 2014 Red Hat Inc. acquired Inktank Storage and have been continuing development since, which resulted in a stable version (10.2.0) of *Ceph* in April 2016 called "Jewel". There are also other contributors of *Ceph* like *SUSE LINUX GmbH*, *Canonical Ltd.*, *Fujitsu Ltd.* and *Intel Corporation*.

4.1 Functionality

Ceph is only a package of many modules; the important ones are *RADOS*, *librados*, *RADOSGW*, *RBD* and *CephFS*.

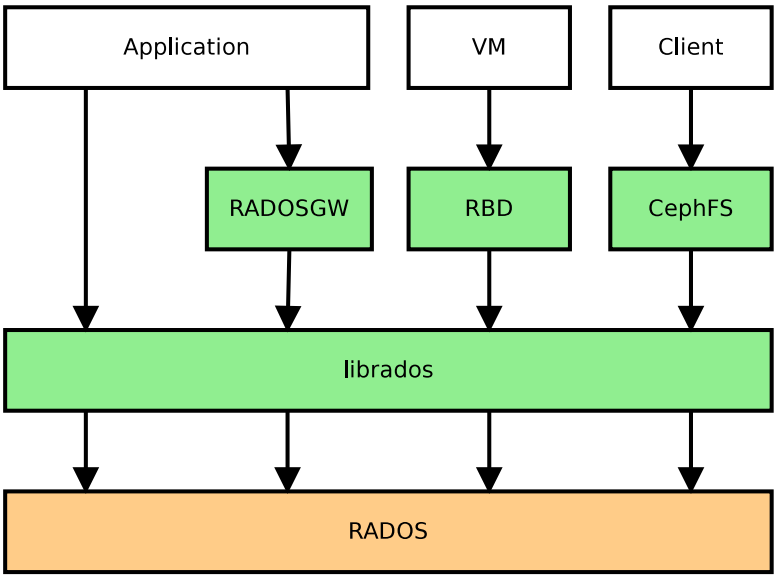


Figure 8: Architecture diagram showing the component relation of the Ceph storage platform (adapted from [22])

RADOS – A reliable autonomic distributed object store. The centerpiece of Ceph because this is where all data belongs to. How exactly this component works is shown in the next paragraph.

librados – A library, written in *C++*, to interact with *RADOS* so that a programmer could implement *Ceph Storage* to an application. Which means interacting with *Object Storage* directly, there are no meta-data servers included. This library is also available in *C*, *Python* and *Java*.

RADOSGW – A gateway to *RADOS*. *Ceph* provides with this module a solution for cloud applications to interfere. This gateway is a *RESTful API*, which is compatible with two popular APIs: *Amazon S3* and *OpenStack Swift*. Although, it arbitrates as a wrapper for *librados*.

RBD – *RADOS Block Device*. If an application wants the possibility to access a "raw" logical block, this module offers a way to store data in blocks (e.g. 512 bytes per block), which are still striped over multiple *OSDs*. This functionality is useful for virtualization *(QEMU/KVM)*.

CephFS – *Ceph's* file system, is a client to enable the possibility to mount a *Ceph Cluster* in a *POSIX*

filesystem. Therefore, a *Metadata Server* (MDS) is mandatory, which stores all *POSIX* relevant information like timestamps, ownership, filename and directory. In turn, these data are saved as an object in *Ceph's Object Storage*. The *MDS* is crucial for commands like `ls` and other metadata specific commands because an *Object Storage* does not handle metadata like these.

In the beginning, *CephFS* was implemented as an *FUSE*[9] driver, which meant to be slow. Therefore, the step to include *CephFS* as a kernel module was not too far. Linus Torvalds packaged kernel 2.6.34 (May 2010) with the *Ceph Client*.

4.2 Cluster

A *Ceph Cluster* is a distributed filesystem, so there have to be many nodes with different tasks, where a node is a dedicated server. The essential ones are *Monitoring Nodes*, *Object Storage Nodes* and *Metadata Nodes*.

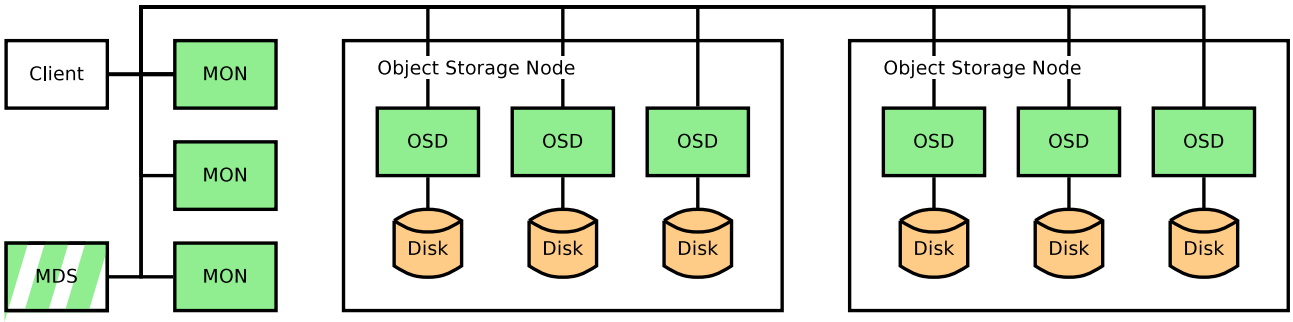


Figure 9: Ceph Object Storage Cluster Setup

Monitoring Node – A highly available daemon, which manages and holds the master copy of the cluster map and configurations. Clients receive their copy from them. Furthermore, there has to be an odd amount of monitoring nodes and if more than 50% are offline/down, the cluster is locking down. Meanwhile, no client can connect to ensure cluster consistency.

An important note is monitors are not storing data to the cluster, instead of that they keep it local on disk.

Metadata Server – They are only relevant in combination with *CephFS* or another *POSIX*-compliant shared filesystem. At least it is a simple key-value store with information about directory hierarchy, ownership, timestamps, permissions and snapshots on any directory.

Certainly, the data itself is saved as an object in *RADOS*.

Object Storage Node – A dedicated server containing *Object Storage Devices* (OSDs).

4.3 Object Storage Device

They are the core of every *Object Storage Cluster*. An *OSD* consists of an active daemon and a single disk or raid containing a filesystem. *Ceph* allows three underlying filesystems: *XFS*, *Ext4* or *btrfs*. It is

[9] Filesystem in USErspace

possible to add more than one disk to an *OSD*, but this is not recommended. Finally, all object(parts) are stored as a file in the root directory / of e.g. *btrfs*.

By default, a primary *OSD* exists for every object; thence replication and striping to other devices is starting. Moreover, it is possible to add a tier level to *OSDs*. This allows promotion and degradation of object importance, which could decide about storing on an *SSD* or *HDD*.

Another strategy to handle different access importances e.g. cold data, which means they were written once, but rarely read, are different hardware configurations (higher *HDD* capacity, lower throughput vs. low *SSD* capacity, higher throughput).

4.4 Storage Management

How are objects saved, striped, replicated, recovered and managed? Foremost, there are different layers to arrange objects. Figure 10 shows three layers: *Pools*, *Placement Groups* (PG) and *Object Storage Devices* (OSD).

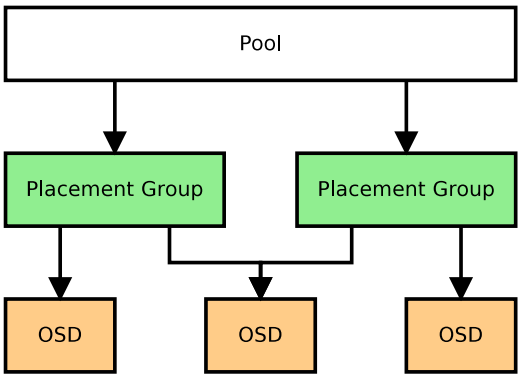


Figure 10: Object management

Pools are logical partitions, containing many *Placement Groups* and *Object Storage Devices* exclusively. Therefore another pool cannot access *OSDs* from another pool. Each pool contains a set of configurations: *Resilience* indicates the maximum number of *OSD* failures. On a fully replicated pool it represents the number of copies. Furthermore, choosing a specific *CRUSH* rule, which best fits the scenario, setting *ownerships* by user ID to limit access, enabling *quota* by restricting amount of bytes or objects and the creation of *snapshots*, to versioning given objects, are part of the settings.

Moreover, each object is assigned to a **Placement Group**. Thereby, it is intended to have a high amount of *PGs*; ca. 100 per *OSD* are recommended, which is configured in the pools' settings. *PGs* do not own an *OSD* exclusively but share them. As seen in figure 11 the pool has a replication level of "2", corresponding to this each object in each *PG* is stored on two different *OSD*. Furthermore, the three *PGs* share four *OSDs* among each other.

The advantage of replication is, of course, load balancing, in the case of a slow, overloaded or offline *OSD*. Additionally, data consistency is enhanced, the higher the redundancy level, the more fail-safe the data. *PGs* make sure that objects are distributed equally to all *OSDs*.

How do *PGs* increase recovery speed? If "OSD 2" in figure 11 lost data and needs a recovery, it could copy objects from "OSD 1" and "OSD 4" simultaneously without causing heavy loads on one node only. Let's assume that there are 20 *OSDs* and 2000 *PGs*. If one *OSD* needs to recover, it is very likely it

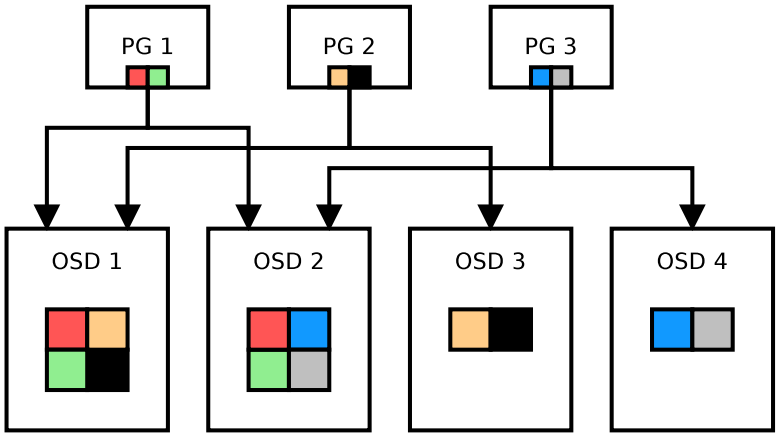


Figure 11: Object distribution in a pool (adapted from [20])

could copy objects from all *OSDs*, because many *PGs* use this *OSD* in combination with one of the other 19.

4.4.1 Read/Write an object

If a user wants to write an object to the cluster, a special algorithm called **CRUSH**[10] decides which *OSD* is the primary one for that object. *CRUSH* maps contain a topological view of the cluster and pseudo-randomly store and retrieve data to ensure an equal distribution. Furthermore, *Ceph* clients are always connected directly to the primary *OSD* without a particular gateway or likewise.

Is is also possible to stripe objects to many *OSDs*, comparable to *RAID 5* or *6*. As seen in figure 12, an object is striped with a variable shard size over six devices. The first four devices are used to stripe original pieces of an object (part 1 to 14) and the last two save parities of the object parts.

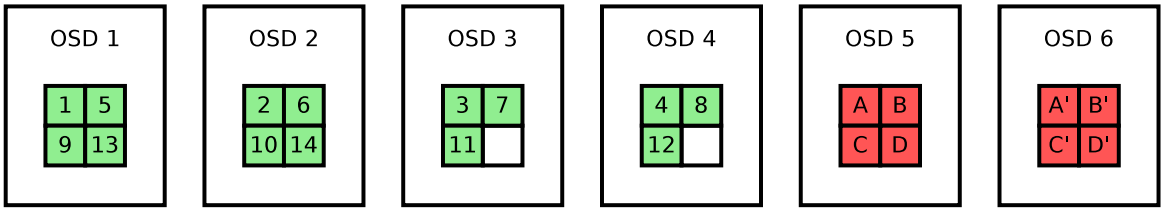


Figure 12: Object striping (adapted from [24, pp. 54])

4.4.2 CRUSH

The abbreviation *CRUSH* stands for *Controlled, Scalable, Decentralized Placement of Replicated Data*. This set of algorithms ensures replication and distribution of objects in a *Ceph* cluster.

CRUSH is a *"pseudo-random placement algorithm"* to construct a deterministic, stable and repeatable mapping path from an object ID and cluster topology. Thereby, it is not possible to rename an object (i.e. change the object ID), because there are no lookup tables, metadata servers or comparable services.

[10] Controlled, Scalable, Decentralized Placement of Replicated Data

This results in the advantage because every location of an object can be calculated from object ID and cluster map.

Compared to other storage solutions, locating an object can be done decentralized and without crawling inodes *(POSIX)* or dedicated servers *(Lustre)*.

The mapping is to ensure that object distribution proceeds even and balanced. It is aided from a *CRUSH* map configuration file, which contains rules, infrastructure topology and weighting of devices (e.g. *OSDs*).

A *CRUSH* map includes devices, bucket-types, buckets and rules. As seen in the following code block, devices are *OSDs* and all of them must be listed.

```
device 0 osd-0
device 1 osd-1
device 2 osd-2
device 3 osd-3
```

Furthermore, buckets are nodes in a hierarchy, e.g. *OSDs*, racks or data centers (as seen in figure 13, and not the bucket concept as seen in *RADOS Gateway API*. To describe the hierarchy, an enumerated list with bucket-types has to been defined. The name and the structure can be selected according to the own infrastructure plan. Here is an example:

```
type 0 osd
type 1 host
type 2 rack
type 3 datacenter
type 4 root
```

Every bucket with its configuration has to been defined, too. In this example, two hosts, each with two *OSDs*, and one rack are described.

```
host host-01
{
    id -2
    alg straw
    hash 0
    item osd-0 weight 1.000
    item osd-1 weight 1.000
}
host host-02
{
    id -2
    alg straw
    hash 0
    item osd-2 weight 1.000
    item osd-3 weight 1.000
```

```
}
rack rack-01
{
    id -1
    alg straw
    hash 0
    item host-01 weight 2.000
    item host-02 weight 2.000
}
```

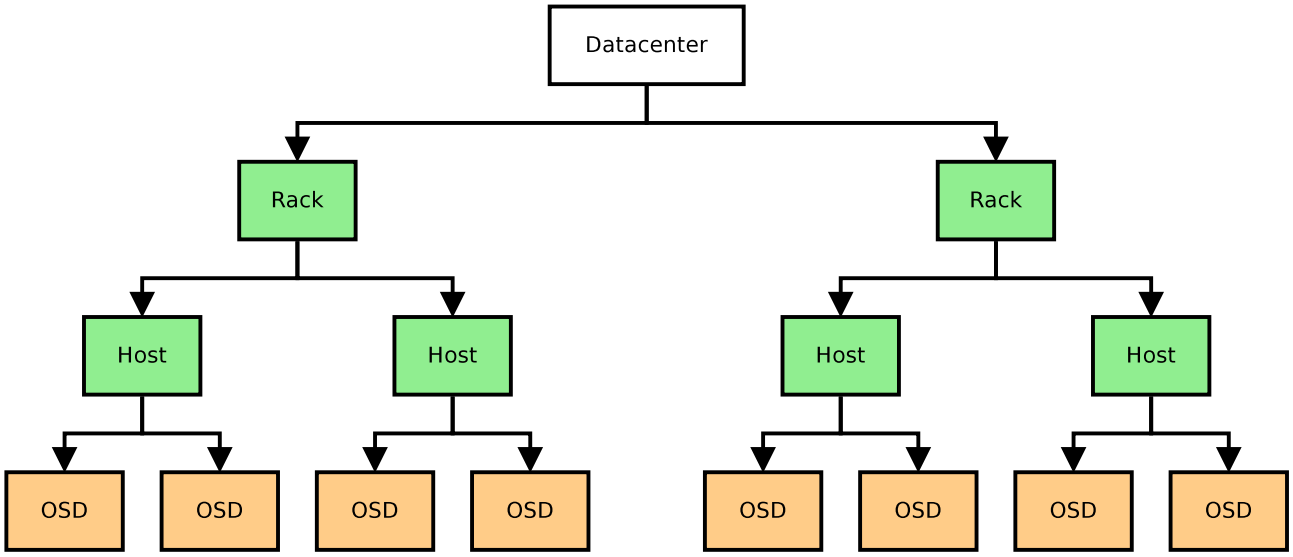


Figure 13: CRUSH map topology

At last, rules to control the data distribution can be deployed, too. The following example defines a rule for OSDs containing SSDs, like `host-02`. Its `ruleset` could set into pool configuration to make use of it. The `type` defines a storage drive (`replicated`) which content has to be replicated between `min_size` and `max_size`. Otherwise, this rule is ignored by *CRUSH*. The last three options (`step`) determine parameters for the named operation.

```
rule ssd
{
    ruleset 1
    type replicated
    min_size 1
    max_size 5
    step take host-02
    step choose firstn 0 type osd
    step emit
}
```

A detailed documentation can be found at the official *Ceph* website [7] and papers from Sage Weil [15], [14], [13]. Parts of the configuration examples are adopted from Sébastien Han [4].

5 Experimental setup

For later use of this bachelor thesis, a testing environment is needed. Therefore, a simple, minimalistic *Ceph Storage Cluster* has to been setup. To conserve hardware resources, all *Ceph* nodes are virtualized on one hardware node. The smallest possible cluster consists of four nodes: *Admin*, *Monitor* and two *OSDs*. An **Admin Node** is needed to configure and deploy *Ceph* on the other nodes.

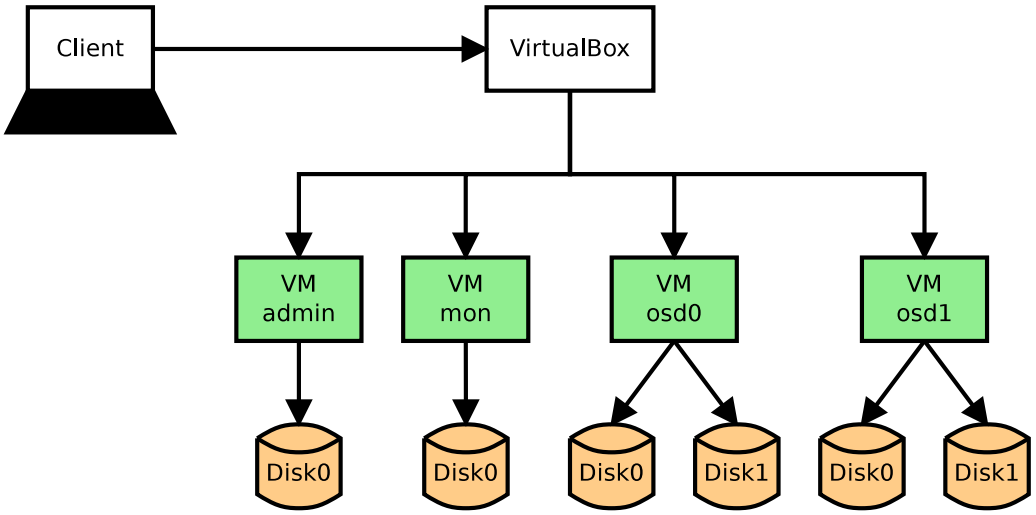


Figure 14: Experimental setup

In this thesis, the hardware node consists of a *Thinkpad X230*:

- **CPU:** Intel® Core™ i7-3520M CPU @ 2.90 GHz (2 cores; hyperthreading and VT-x enabled)
- **Memory:** 8GiB SODIMM DDR3 Synchronous 1600 MHz

As a virtualization program **VirtualBox** (version 5.1.14) is used. The operating system of the host is *Fedora 25* (64-bit) and of the nodes *Debian "jessie"* (64-bit).

5.1 Configuration of VirtualBox

First of all, it is necessary to implement two network interfaces to the nodes. The first one to gain access to the internet via *NAT*[11] and the second one to communicate to each other, called *Host-only*. Figure 15 shows a possible configuration of interface `vboxnet0`, a *Host-only* adapter. *DHCP*[12] has to be turned off because only static *IPs*[13] are used.

In return, it is indispensable to make sure that all *IP* addresses are configured manually and added to a host-list (`/etc/hosts`; see next sections).

Moreover, each node is setup with 1024 MB base memory, one CPU core, a PIIX3-chipset and the following enabled features:

[11] Network Address Translation
[12] Dynamic Host Configuration Protocol
[13] Internet Protocol

- *I/O APIC*: Input/Output Advanced Programmable Interrupt Controller
- *PAE/NX*: Physical Address Extension
- *VT-x*: Intel hardware virtualization extension
- *Nested Paging*: Also called *Second Level Address Translation* to accelerate virtual memory addressing

Beside of that, all nodes have a storage device with a capacity of 8 GB. They are flagged as *Solid State Disk*, to ensure proper handling. To reduce available storage capacity and prevent quadruplication, it is advisable to create a linked clone from a freshly installed/configured VM.

The two *OSDs* contain a second storage device, which is used for *Ceph Object Storage*.

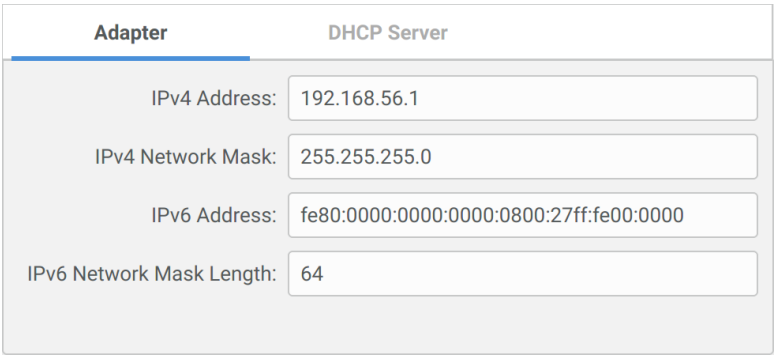


Figure 15: VirtualBox Host-only Network Interface

5.2 Configuration of Debian "jessie"

At first, a clean preconfigured *Debian VM* for cloning is required. For that, it is enough to install a minimal *Debian*, like selecting the *"debian-8.7.1-amd64-netinst"* image. Special configurations during the installation are not required, except for selecting primary network interface, hostname and user creation.

It is necessary to choose the interface, which is configured as NAT in *VirtualBox*, as primary to gain access to the internet. As a hostname terms like *"ceph-admin"* are useful.

Entering a root password is not essential because a user with administrative rights (`sudo`) is setup. Later, *Ceph's* deployment scripts create a user called *"ceph"*. Therefore, another username like *"vm"* should be used as a default user. Furthermore, enabling the *SSH*-server is recommended!

5.2.1 GRUB

To accelerate booting of the *VMs*, lower the timeout of *GRUB*.

```
1 # /etc/default/grub
2
3 GRUB_TIMEOUT=0
```

```
$ grub-mkconfig -o /boot/grub/grub.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.16.0-4-amd64
Found initrd image: /boot/initrd.img-3.16.0-4-amd64
done
```

5.2.2 Network

Before configuring network interfaces, it is required to create a network map. All nodes are connected to each other via *DHCP*-less *Host-only* connections. Therefore, all four VMs should gain an address.

Virtual Machine	IP-Address
ceph-admin	192.168.56.10
ceph-mon	192.168.56.20
ceph-osd0	192.168.56.30
ceph-osd1	192.168.56.40

In the following case, the interface `eth0` is *Host-only* and `eth1` is *NAT* with access to the internet. Accordingly, the first one has to be configured with a static address and the second with a dynamic one.

```
# /etc/network/interfaces

source /etc/network/interfaces.d/*

# localhost
auto lo
iface lo inet loopback

# Internet
allow-hotplug eth1
iface eth1 inet dhcp

# Host-only
auto eth0
iface eth0 inet static
    address 192.168.56.10
    netmask 255.255.255.0
```

Additionally, the `/etc/hosts` file has to been customized, too. For the reason that no *DHCP* and *DNS* exist, the nodes needed to know the translation between hostname to *IP* address.

```
# /etc/hosts

127.0.0.1     localhost
::1           localhost ip6-localhost ip6-loopback
ff02::1       ip6-allnodes
ff02::2       ip6-allrouters

192.168.56.10 ceph-admin
192.168.56.20 ceph-mon
192.168.56.30 ceph-osd0
192.168.56.40 ceph-osd1
```

5.2.3 User permissions

There is a problem with the *Ceph* deployment scripts. They connect via *SSH* to the nodes, to configure, install and deploy. But not with the user *"root"*, they require a user with `sudo` (*"vm"* in this case)!

Additionally, the command `ssh` is called without a *TTY*[14], and that is the reason why `sudo` cannot prompt a password request. According to this, `/etc/sudoers` has to be specially configured. Simply remove password request. **NO, this is really NOT a good idea on a production system!**

```
# /etc/sudoers

vm ALL=(ALL) NOPASSWD: ALL
```

5.2.4 Cloning

To avoid non-essential work and storage capacity, *VirtualBox* can clone a *VM* and link storage devices, like overlaying a snapshot. Thus, *"ceph-mon"*, *"ceph-osd0"* and *"ceph-osd1"* are linked clones as seen in figure 16. Although, it is wise to *Reinitialize the MAC address of all network cards* during cloning.

5.2.5 Post-Cloning

After successful cloning, the new instances have to be reconfigured. The following steps have to be accomplished:

1. Change hostname in `/etc/hostname`
2. Change *IP* address in `/etc/network/interfaces`, corresponding to the network map
3. Add an additional storage device to *"ceph-osd0"* and *"ceph-osd1"*

[14] TeleTYpewriter

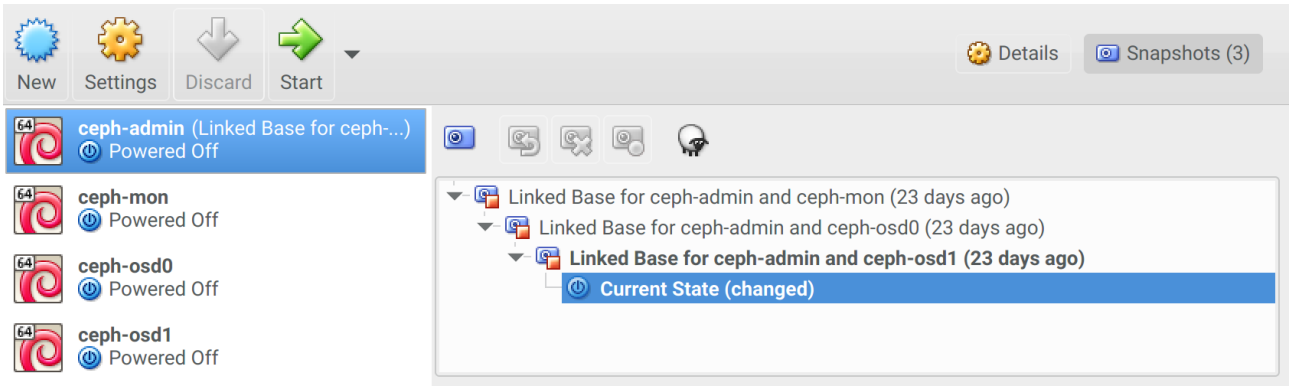


Figure 16: VirtualBox Linked Clones

5.3 Deployment of Ceph

All operations are done on the node *"ceph-admin"*. To increase comfort, a *SSH-Key* should be created and deployed to all nodes.

```
$ ssh-keygen -f ~/.ssh/ceph
Generating public/private rsa key pair.
$ ssh-copy-id -i ~/.ssh/ceph vm@ceph-mon
Number of key(s) added: 1
$ ssh-copy-id -i ~/.ssh/ceph vm@ceph-osd0
Number of key(s) added: 1
$ ssh-copy-id -i ~/.ssh/ceph vm@ceph-osd1
Number of key(s) added: 1
```

Also, configure *SSH* to use this identity file.

```
# ~/.ssh/config

Host ceph-*
    User vm
    IdentityFile ~/.ssh/ceph
```

Now *SSH* is able to connect to other nodes neither asking for a login password nor an administrative one. The next step is to add `ceph-deploy` to *APT* repository configuration and install it.

```
$ echo "deb https://download.ceph.com/debian-jewel/ jessie main" >
↪ /etc/apt/sources.list.d/ceph.list
$ wget -q -O- 'https://download.ceph.com/keys/release.asc' | sudo apt-key add -
$ apt install apt-transport-https
$ apt update
$ apt install ceph-deploy
```

Now, create a directory where the deploy script can store its configuration files.

```
$ mkdir ceph-cluster
$ cd ceph-cluster
```

After that, a monitoring node has to be added. It is necessary to specify an *IP* range to enforce usage of the *Host-only* network interface; simply use the `--public-network` argument. Additionally, to ensure that the initial monitors use the correct *IP* range, it is recommended to append its *IP* address.

```
$ ceph-deploy new --public-network 192.168.56.0/24 ceph-mon:192.168.56.20
[ceph_deploy.new][DEBUG ] Creating new cluster named ceph
[ceph_deploy.new][INFO  ] making sure passwordless SSH succeeds
[ceph-mon][DEBUG ] connected to host: ceph-admin
[ceph-mon][INFO  ] Running command: ssh -CT -o BatchMode=yes ceph-mon
[ceph-mon][DEBUG ] connection detected need for sudo
[ceph-mon][DEBUG ] connected to host: ceph-mon
[ceph-mon][DEBUG ] *IP* addresses found: [u'192.168.56.20', u'10.0.2.15']
[ceph_deploy.new][DEBUG ] Resolving host ceph-mon
[ceph_deploy.new][DEBUG ] Monitor ceph-mon at 192.168.56.20
[ceph_deploy.new][DEBUG ] Monitor initial members are ['ceph-mon']
[ceph_deploy.new][DEBUG ] Monitor addrs are ['192.168.56.20']
[ceph_deploy.new][DEBUG ] Creating a random mon key...
[ceph_deploy.new][DEBUG ] Writing monitor keyring to ceph.mon.keyring...
[ceph_deploy.new][DEBUG ] Writing initial config to ceph.conf...
```

To ensure that *Ceph* is healthy with only two *OSDs*, a little config snippet is needed:

```
# ceph.conf

[global]
osd pool default size = 2
```

The next steps are the installation of *Ceph* on all nodes and initializing the monitoring node.

```
$ ceph-deploy install ceph-admin ceph-mon ceph-osd0 ceph-osd1
[ceph-admin][INFO  ] installing *Ceph* on ceph-admin
[ceph-mon][INFO  ] installing *Ceph* on ceph-mon
[ceph-osd0][INFO  ] installing *Ceph* on ceph-osd0
[ceph-osd1][INFO  ] installing *Ceph* on ceph-osd1
$ ceph-deploy mon create-initial
[ceph_deploy.mon][DEBUG ] Deploying mon, cluster *Ceph* hosts ceph-mon
[ceph_deploy.gatherkeys][INFO  ] Storing ceph.client.admin.keyring
[ceph_deploy.gatherkeys][INFO  ] Storing ceph.bootstrap-mds.keyring
[ceph_deploy.gatherkeys][INFO  ] keyring 'ceph.mon.keyring' already exists
```

```
[ceph_deploy.gatherkeys][INFO  ] Storing ceph.bootstrap-osd.keyring
[ceph_deploy.gatherkeys][INFO  ] Storing ceph.bootstrap-rgw.keyring
```

As seen in the last few lines of debug output, the monitoring node sent a collection of keyrings, which have to be deployed to all nodes later. At first, each *OSD* has to be configured to use the second disk as cluster storage. In this case, the second disk is completely available.

```
$ ceph-deploy *OSD* create ceph-osd0:/dev/sdb
[ceph_deploy.osd][DEBUG ] Deploying *OSD* to ceph-osd0
$ ceph-deploy *OSD* create ceph-osd1:/dev/sdb
[ceph_deploy.osd][DEBUG ] Deploying *OSD* to ceph-osd1
$ ceph-deploy admin ceph-admin ceph-mon ceph-osd0 ceph-osd1
[ceph_deploy.admin][DEBUG ] Pushing admin keys and conf to ceph-admin
[ceph_deploy.admin][DEBUG ] Pushing admin keys and conf to ceph-mon
[ceph_deploy.admin][DEBUG ] Pushing admin keys and conf to ceph-osd0
[ceph_deploy.admin][DEBUG ] Pushing admin keys and conf to ceph-osd1
```

In the end, *Ceph* status and health report can be seen via `ceph status`. However, these commands have to be submitted via `root`, because *ceph-admin* is now a node and contains the keyrings in `/etc/ceph/` with restricted permissions.

```
$ *Ceph* status
    cluster f753696a-978b-45bd-aeb1-60e5bef21c2c
     health HEALTH_OK
     monmap e1: 1 mons at {ceph-mon=192.168.56.20:6789/0}
            election epoch 3, quorum 0 ceph-mon
     osdmap e10: 2 osds: 2 up, 2 in
            flags sortbitwise,require_jewel_osds
      pgmap v20: 64 pgs, 1 pools, 0 bytes data, 0 objects
            68960 kB used, 6054 MB / 6121 MB avail
                  64 active+clean
```

If the cluster is setup correctly, the status message is `health HEALTH_OK`. Now, it is the time to add pools, to store objects inside.

A *Ceph* cluster already contains pools, depending on deploymemt. However, in this example, another pool is created, a list of available pools and the configuration of the new one is printed out.

```
# Name: data
# Number of PGs: 8
$ *Ceph* *OSD* pool create data 8
pool 'data' created
$ *Ceph* *OSD* lspools
0 rbd, 1 data
$ *Ceph* *OSD* pool get data all
size: 2
min_size: 1
```

```
crash_replay_interval: 0
pg_num: 64
pgp_num: 64
crush_ruleset: 0
hashpspool: true
nodelete: false
nopgchange: false
nosizechange: false
write_fadvise_dontneed: false
noscrub: false
nodeep-scrub: false
use_gmt_hitset: 1
auid: 0
min_write_recency_for_promote: 0
fast_read: 0
```

One of the last steps is testing the new storage cluster. A simple one is to create and copy a file to *Ceph*, check its existence and locate its position.

```
$ echo "Sample Data" > testfile.txt
$ rados put testobject testfile.txt --pool=data
$ rados -p data ls
testobject
$ *Ceph* *OSD* map data testobject
osdmap e27 pool 'data' (1) object 'testobject' -> pg 1.98824931 (1.31) -> up ([0,1],
↪  p0) acting ([0,1], p0)
```

An overview of a quick *Ceph* installation can be found on the online documentation [6].

6 Ceph-FUSE driver

6.1 FUSE

FUSE is an abbreviation and stands for *"Filesystem in Userspace"*. It is itself a kernel module allowing users to deploy their filesystem driver without root privileges. Instead, they run in user space and communicate via the kernel module *libfuse*.

Thereby, system (kernel) stability is not deteriorated by additional modules, which can crash and cause a kernel panic. Furthermore, it is easier to program a filesystem driver against *libfuse* than writing a whole kernel module. Above all, these programs could run by a user without loading into the kernel or sign it, if secure boot is used.

One of the big disadvantages of *FUSE* in comparison to a native filesystem driver is the IO performance. Thus, *FUSE* driver runs in user mode; a handful context switches are required to handle user-to-kernel operations.

6.2 How does FUSE work

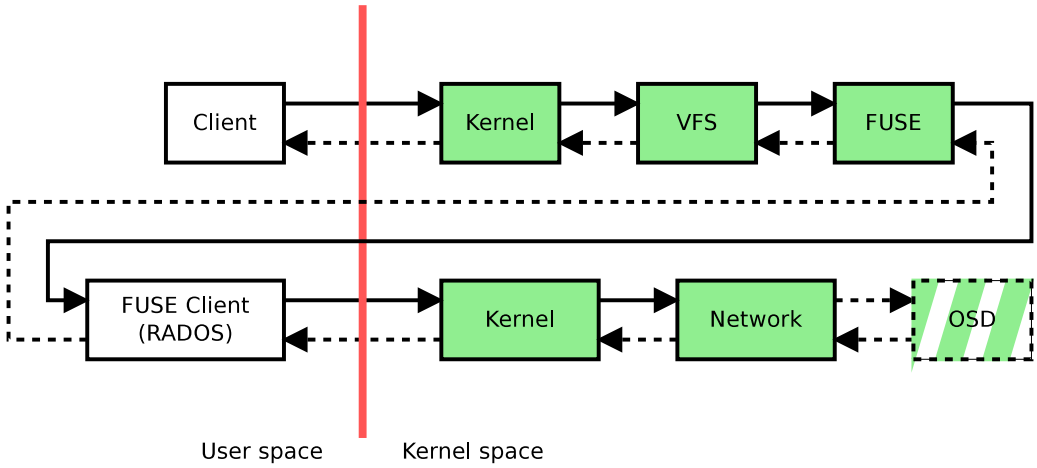


Figure 17: Context switches of a Ceph-FUSE driver (adapted from [21])

To use a *FUSE* resource, it has to been mounted in the *Virtual Filesystem* like a normal block device. Therefore, a normal user cannot distinguish this mountpoint from a directory, a mounted block device or a network-attached *FUSE* resource.

For every operation the client requests there is a long chain of modules to walk through (as seen in figure 17). For example: If a client makes an operation request on the filesystem, e.g. `stat` to receive file or directory information, this is passed through the kernel and *Virtual Filesystem* to the *FUSE*. This kernel module interacts with the *FUSE* Client, which was started by a user in *user space*.

In this case, a network connection is required to receive data. Therefore, the request is transmitted to the network stack, which operates in *kernel space*.

6.3 A simple FUSE driver for Ceph

A *FUSE* driver is usually written in *C* and consists of a list of supported operations, the implementations of these and a `main()` entry point. This proof-of-concept uses a flat directory hierarchy to keep it simple. Besides, it is not necessary for functionality or benchmarking.

The needed information can be looked up in the documentation of *librados* [8] and *libfuse* [2].

Needed headers

To compile this driver, a few developer packages are needed. Both *libfuse* and *librados* have to be installed. It is also recommended to set the used *FUSE* version to ensure correct API usage.

```
1 #define FUSE_USE_VERSION 26
2
3 #include <fuse.h>
4 #include <rados/librados.h>
5 #include <errno.h>
6 #include <stdio.h>
7 #include <stdlib.h>
```

Configuration

For this experiment, pool name and the paths to keyring and ceph.conf are compiled in to minimize complexity.

```
1 #define POOL_NAME    "data"
2 #define CONF_PATH    "/etc/ceph/ceph.conf"
3 #define KEYRING_PATH "/etc/ceph/ceph.client.admin.keyring"
```

Operation list

At first, a list of implemented operations should be determined. It contains, among other things, *POSIX* operations like `open`, `read` or `write`. Technically, this list is a `struct` and maps operations to their callback functions, using a function pointer.

Some *POSIX File IO* commands like `getattr` and `readdir` are recommended to ensure interoperability. Otherwise, the mountpoint has no permissions or ownership set and cannot list any entries, which leads to a curious state. This driver uses the following *POSIX* operations:

- **create** – it is called, if a new file should be created
- **getattr** – which is responsible for information presented in `stat`. Therefore, it has to return permissions, ownership, modification/create/access times, block type, etc.
- **read** – map (parts) of the file to memory for processing in other programs
- **write** – save (parts) of a file to disk. As usual in normal *POSIX* filesystems, parts of the file can be overwritten and writing above file size is interpreted as an addition
- **readdir** – list all entries of the given path
- **truncate** – trim a file to given size parameter
- **unlink** – remove a file or directory and free space

Additionally, two commands, which are part of *FUSE*, are added to the operations list: `init` and `destroy`. They are responsible for connecting and disconnecting to the cluster.

```
static struct fuse_operations fuse_ceph_operations =
{
    .init = init_callback,
    .destroy = destroy_callback,
    .read = read_callback,
    .write = write_callback,
    .create = create_callback,
    .truncate = truncate_callback,
    .unlink = unlink_callback
    .getattr = getattr_callback,
    .readdir = readdir_callback,
};
```

Main entry point
The `main()`-function only calls the main-function of *FUSE*. In this process, all command line arguments and the list of operations are passed on.

```
int main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &fuse_ceph_operations, NULL);
}
```

6.3.1 Implementation of *FUSE* operations

Two variables have to be declared. One of them contains the cluster handler to interact with. The other one is an IO handler for the *POSIX*-like operations such as `read()` and `write()`. Besides, a helper function to raise an error is defined, too.

```
rados_t cluster; rados_ioctx_t io;

static int is_error(int err, char *message)
{
    if(err < 0)
    {
        fprintf(stderr, "%s: %s\n", message, strerror(-err));
        return 1;
    }
    else
    {
        return 0;
    }
}
```

The **initialization** is only called at startup to read config files, connect to the cluster and select a specific pool. At line 3 a cluster handle is created, which is used to read the specified config file (line 8). Moreover, a keyring with the permissions to connect is loaded (line 13), too. Thereafter, the handle opens a connections to the *Ceph* cluster (line 18). In the last step an IO handle is initialized while selecting a pool (line 23). This IO handle is needed to interact with objects of the selected pool.

```
1  static void* init_callback(struct fuse_conn_info *conn)
2  {
3      if(is_error(rados_create(&cluster, NULL), "Cannot create a cluster handle"))
4      {
5          exit(1);
6      }
7
8      if(is_error(rados_conf_read_file(cluster, CONF_PATH), "Cannot read config file"))
9      {
10         exit(1);
11     }
12
13     if(is_error(rados_conf_set(cluster, "keyring", KEYRING_PATH), "Cannot read
       ↪ keyring"))
14     {
15         exit(1);
16     }
17
18     if(is_error(rados_connect(cluster), "Cannot connect to cluster"))
19     {
20         exit(1);
21     }
22
23     if(is_error(rados_ioctx_create(cluster, POOL_NAME, &io), "Cannot open RADOS
       ↪ pool"))
24     {
25         rados_shutdown(cluster);
26         exit(1);
27     }
28
29     return NULL;
30 }
```

Shortly before termination of the driver, the deconstructor-like function `destroy()` is called to close existing connections to leave the cluster clean and fresh.

```
1  static void destroy_callback(void* private_data)
2  {
3      rados_ioctx_destroy(io);
4      rados_shutdown(cluster);
5  }
```

To **read** an object, it is possible to pass `buffer`, `size` and `offset` directly to *librados*. Only the *path* has to be modified. It contains the full path of the selected object, e.g. `/object_name`. To get only the object name, a simple pointer arithmetic is sufficient to eliminate the leading slash. If `ret` on line 3 is negative, an error is raised (line 7), otherwise the size of read bytes is returned (line 11).

```
1  static int read_callback(const char *path, char *buf, size_t size, off_t offset,
   ↪  struct fuse_file_info *fi)
2  {
3      int ret = rados_read(io, (path + 1), buf, size, offset);
4
5      if(is_error(ret, "Cannot read part of the object"))
6      {
7          return 0;
8      }
9      else
10     {
11         return ret;
12     }
13 }
```

The **write** operation is almost equal to the previous command. In this case, *RADOS* will overwrite objects with given `buffer`, `size` and `offset`, similar to other *POSIX* filesystems. If `ret` on line 3 is not zero, an error is raised (lines 7,15), otherwise the size of written bytes is returned (line 11).

```
1  static int write_callback(const char *path, const char *buf, size_t size, off_t
   ↪  offset, struct fuse_file_info *fi)
2  {
3      int ret = rados_write(io, (path + 1), buf, size, offset);
4
5      if(is_error(ret, "Cannot write to object"))
6      {
7          return -errno;
8      }
9      else if(ret == 0)
10     {
11         return size;
12     }
13     else
14     {
15         return -errno;
16     }
17 }
```

If a new file is created, the function `create()` is called. It is necessary to store at least an empty object (i.e. with a size of zero bytes) because if this new file is saved, there have to be `stat()` information, otherwise *FUSE* is raising errors.

```
1  static int create_callback(const char *path, mode_t mode, struct fuse_file_info *fi)
2  {
3      if(is_error(rados_write_full(io, (path + 1), "", 0), "Cannot create new object"))
4      {
5          return -errno;
6      }
7      else
8      {
9          return 0;
10     }
11 }
```

Truncating is not necessary, but since it is supported by *librados*, it should be implemented.

```
1  static int truncate_callback(const char *path, off_t size)
2  {
3      if(is_error(rados_trunc(io, (path + 1), size), "Cannot truncate object"))
4      {
5          return -errno;
6      }
7      else
8      {
9          return 0;
10     }
11 }
```

Moreover, **deleting** an object is implemented with a single API call (line 3).

```
1  static int unlink_callback(const char* path)
2  {
3      if(is_error(rados_remove(io, (path + 1)), "Cannot delete object"))
4      {
5          return -errno;
6      }
7      else
8      {
9          return 0;
10     }
11 }
```

Attributes are an important component of an entity. *Ceph* stores the object name as well the modification time and size. However, neither permissions nor ownership are filed.

To reduce problems, the easiest way is to set permissions to default (`0755` for directories (line 9), `0644` for files (line 17)) and ownership to the executing user (lines 11,12,19,20). Size and modification time were retrieved while calling `rados_stat()` (line 15) and are committed to *FUSE* (lines 21,22)

```
1  static int getattr_callback(const char *path, struct stat *stbuf)
2  {
3      memset(stbuf, 0, sizeof(struct stat));
4      uint64_t size;
5      time_t mtime;
6
7      if(strcmp(path, "/") == 0)
8      {
9          stbuf->st_mode = S_IFDIR | 0755;
10         stbuf->st_nlink = 2;
11         stbuf->st_uid = getuid();
12         stbuf->st_gid = getgid();
13         return 0;
14     }
15     else if(!is_error(rados_stat(io, (path + 1), &size, &mtime), "Object not found"))
16     {
17         stbuf->st_mode = S_IFREG | 0644;
18         stbuf->st_nlink = 1;
19         stbuf->st_uid = getuid();
20         stbuf->st_gid = getgid();
21         stbuf->st_size = size;
22         stbuf->st_mtime = mtime;
23         return 0;
24     }
25
26     return -ENOENT;
27 }
```

Readdir is implemented, to return always the root directory with all objects. Like a *POSIX* filesystem, the own and previous directory are set (lines 11,12) and since there are no subdirectories, all other requests are invalid (line 8). The API call `rados_objects_list_open` creates a list of all objects, which is iterated through (lines 14-19).

```
1  static int readdir_callback(const char *path, void *buf, fuse_fill_dir_t filler,
   ↪  off_t offset, struct fuse_file_info *fi)
2  {
3      rados_list_ctx_t object_list;
4      const char *object_name;
5
6      if(strcmp(path, "/") != 0)
7      {
8          return -ENOENT;
9      }
10
11     filler(buf, ".", NULL, 0);
12     filler(buf, "..", NULL, 0);
13
14     rados_objects_list_open(io, &object_list);
15     while(rados_objects_list_next(object_list, &object_name, NULL) != -ENOENT)
```

```
16      {
17          filler(buf, object_name, NULL, 0);
18      }
19      rados_objects_list_close(object_list);
20
21      return 0;
22  }
```

6.3.2 Renaming

Since *Ceph* uses object names to calculate the position of the data on the cluster, it is not possible to change an objects' name. Solutions are to create a new object and copy all data, manage object names in a separate database or use *extended file attributes* to store an alternative name to show.

6.3.3 Compile and run

In this experiment, *GCC 6.3.1* is used as a compiler. Furthermore, it is always recommended to show all warnings.

```
1  $ gcc -Wall -pedantic -O3 ceph-fuse.c -lrados $(pkg-config *FUSE* --cflags --libs) -o
   ↪  ceph-fuse
```

To mount *Ceph* to a selected directory, simply execute the binary and pass targeted mountpoint as an argument.

```
1  $ ./ceph-fuse /mnt
```

7 Benchmarks

Last but not least, it is interesting to look at the performance of *Ceph*. Therefore, two testing processes to compare to each other are presented. The first one benchmarks read/write performance of previously mentioned *FUSE* driver with *IOR*. At the second one, the native `rados` shell command is used to measure the maximum possible throughput in this test constellation.

7.1 Setup

The benchmarks were done on a single host, namely a monitor and two *OSDs* running on the same dedicated hardware; due to available testing resources.

The host consists of two *Intel® Xeon® CPU X5560 @ 2.80G Hz* CPUs (hyperthreading was deactivated) with 12 GB RAM. The two *OSDs* share an *HDD (Hitachi HUA72202; 512 physical and logical block size)* with two partitions à 500 GB, each formatted with *XFS* and used by one *OSD*. *Ubuntu 16.04.2 LTS 64bit* with kernel *4.4.0-63* was installed on the node.

Thus, a single host is used for the entire storage cluster with two *OSDs* and a monitor, the preferences in the following example have to been added in `ceph.conf` to ensure the ability to run. The option in line 4 set the minimal amount of pools to two and line 5 allows *Ceph* to replicate objects on the same node.

```
1 # /etc/ceph/ceph.conf
2
3 [global]
4 osd pool default size = 2
5 osd crush chooseleaf type = 0
```

7.2 Measurement methodology

The first series of benchmarks is done by *IOR*[15], which is used for benchmarking filesystems via interfaces like *POSIX*, *MPI-IO* and *HDF5*.

IOR in version *3.0.1*, which was published by the *Lawrence Livermore National Laboratory*, was compiled with *GCC*[16] *6.3.0* and *OpenMPI 2.0.1*.

```
1 // Compile IOR
2 $ ./configure --prefix=${HOME} --with-posix --without-mpiio --without-lustre
  ↪ --without-hdf5 --without-ncmpi
3 $ make
4 $ make install
```

[15] Interleaved-Or-Random
[16] GNU Compiler Collection

IOR was programmed to measure the time between beginning and ending of a transfer. The amount of data to transmitted and the size of every chunk (block size) have to been set. In the end, the transfer volume is divided by elapsed time to determine average throughput.

Ceph was mounted via the *FUSE* driver with two additional options: `direct_io` and `big_writes`. *Direct IO* is enabled to bypass read and write caches, managed by the kernel. *Big Writes* allows the possibility to write larger blocks than e.g. 4K per time. In this scenario the biggest block size was 128K.

```
// Mount FUSE
$ ${HOME}/ceph-fuse -o direct_io -o big_writes ${HOME}/mount/
```

Each benchmark configuration was repeated 20 times to get a significant dataset. In *IOR* the amount of data is called *block size* and was varied between 1 M, 10 M, 100 M and 1 G. The size per transaction is called *transfer size* and was set to 4 K, 8 K, 16 K, 32 K, 64 K and 128 K, but only the last three options are submitted for 100 M and 1 G *block size* because the duration of one iteration would exceed processing time limits on the cluster.

Also, *Direct IO* was enabled to bypass kernel caches by setting `-B`.

```
// Run IOR
$ ${HOME}/bin/ior -a POSIX -b ${BLOCKSIZE} -t ${TRANSFERSIZE} -i 20 -B -o
↪  ${HOME}/mount/ior_test
```

To compare throughput via *FUSE* with another I/O technique, which connects to *Ceph* in a native way, a simple setup with `rados` and `time` commands was built.

Specific data packages were prepared with `dd` and `/dev/urandom`, which are transferred later.

```
$ dd if=/dev/urandom of=/dev/shm/1m.data bs=1M count=1
$ dd if=/dev/urandom of=/dev/shm/10m.data bs=1M count=10
$ dd if=/dev/urandom of=/dev/shm/100m.data bs=1M count=100
$ dd if=/dev/urandom of=/dev/shm/1g.data bs=1M count=1000
```

The write operation is done via `rados put` and read operation via `rados get`, in which the file is stored to `/dev/shm/`. This path is mounted via *TMPFS*, thus located in RAM.

Before downloading an object from *Ceph*, kernel's *PageCache* has to been freed. So that the real download bandwidth could be measured. This is done by calling `sync` and setting a variable in `/proc`.

One important fact is, that the original `time` command is used, not the built-in from *BASH*.

```
// Write
$ /usr/bin/time rados -p data put 10m.object /dev/shm/10m.data

// Read
$ for n in {1..100}; do \
$     sync; \
$     echo 3 > /proc/sys/vm/drop_caches \
$ done
$ /usr/bin/time rados -p data get 10m.object /dev/shm/dummy
```

7.3 Results from IOR

1M Filesize, writing

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	0.0685	0.1429	0.2690	0.5116	0.9574	1.7690
Median (MiB/s)	0.0675	0.1440	0.2758	0.5189	0.9720	1.8400
Standard Deviation (MiB/s)	0.0044	0.0100	0.0176	0.0416	0.0850	0.1805

10M Filesize, writing

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	0.0683	0.1384	0.2761	0.5376	1.1086	2.0920
Median (MiB/s)	0.0682	0.1381	0.2784	0.5343	1.1300	2.1300
Standard Deviation (MiB/s)	0.0019	0.0042	0.0102	0.0402	0.0689	0.1371

100M Filesize, writing

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	0.7827	1.8035	3.2035
Median (MiB/s)	n/a	n/a	n/a	0.7903	1.8200	3.1900
Standard Deviation (MiB/s)	n/a	n/a	n/a	0.2387	0.1064	0.1119

1G Filesize, writing

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	0.9483	1.6930	3.2650
Median (MiB/s)	n/a	n/a	n/a	0.9734	1.6900	3.1700
Standard Deviation (MiB/s)	n/a	n/a	n/a	0.0554	0.0142	0.1767

1G Filesize, writing, without *Direct IO*

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	0.9889	1.6790	3.1095
Median (MiB/s)	n/a	n/a	n/a	1.0050	1.6800	3.1100
Standard Deviation (MiB/s)	n/a	n/a	n/a	0.0427	0.0152	0.0296

1M Filesize, reading

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	8.9710	15.3330	25.7895	42.8885	73.0165	126.7385
Median (MiB/s)	9.1950	15.5800	25.8500	43.3400	72.4200	124.2550
Standard Deviation (MiB/s)	1.6733	3.6199	3.6195	4.2888	4.7250	8.3922

10M Filesize, reading

	4K	8K	16K	32K	64K	128K
Average (MiB/s)	9.7975	17.2015	35.2115	62.3080	90.0935	142.3180
Median (MiB/s)	10.0000	18.0250	34.8450	55.3700	86.0500	143.5800
Standard Deviation (MiB/s)	0.9691	2.7249	7.9593	17.2743	12.1261	10.5217

100M Filesize, reading

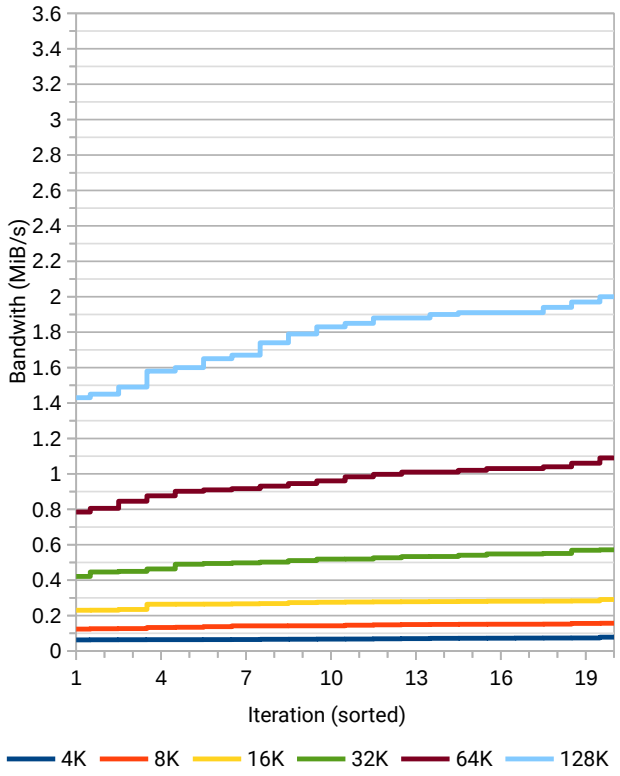
	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	60.1045	85.8680	145.0535
Median (MiB/s)	n/a	n/a	n/a	62.2500	84.7200	145.3650
Standard Deviation (MiB/s)	n/a	n/a	n/a	8.3921	5.8874	5.9266

1G Filesize, reading

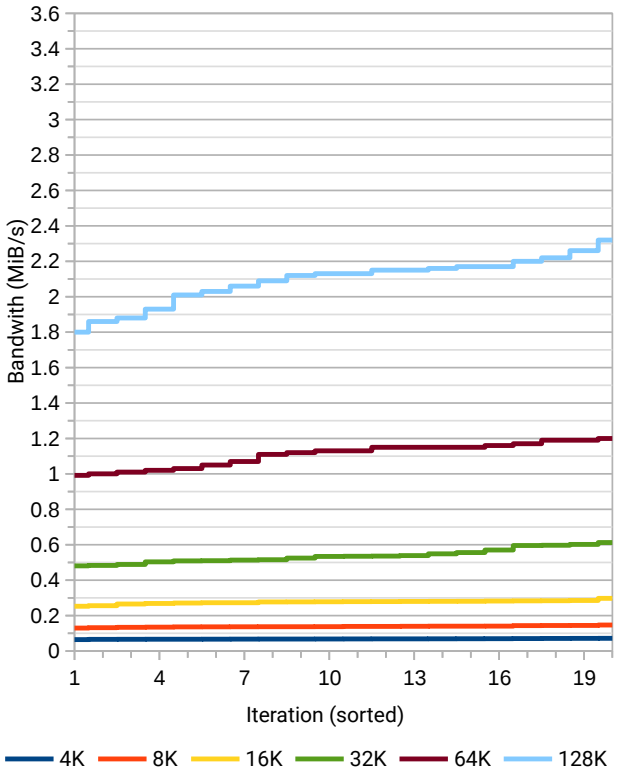
	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	52.3410	89.1605	153.6635
Median (MiB/s)	n/a	n/a	n/a	52.7550	89.3200	154.1600
Standard Deviation (MiB/s)	n/a	n/a	n/a	3.8361	3.2917	5.3540

1G Filesize, reading, without *Direct IO*

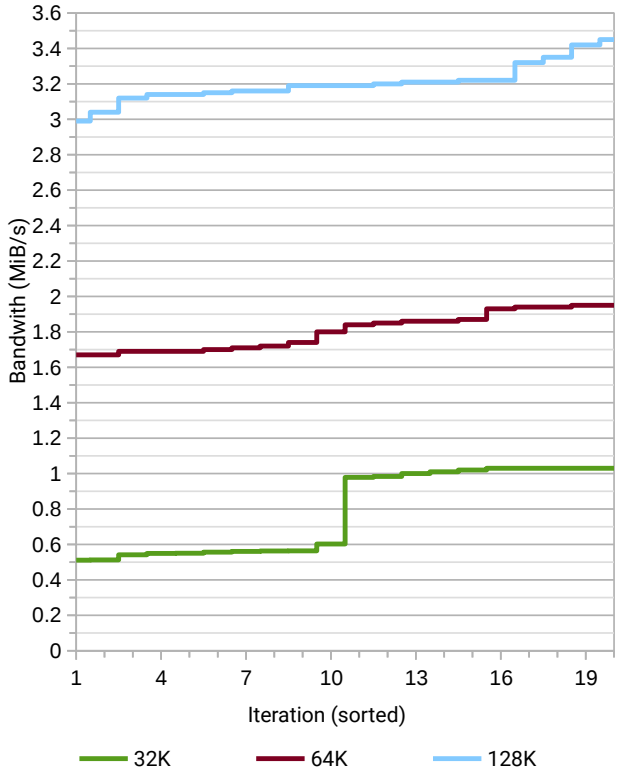
	4K	8K	16K	32K	64K	128K
Average (MiB/s)	n/a	n/a	n/a	312.3930	316.5310	325.9650
Median (MiB/s)	n/a	n/a	n/a	310.7350	313.9150	317.8400
Standard Deviation (MiB/s)	n/a	n/a	n/a	5.7526	9.4835	44.7208



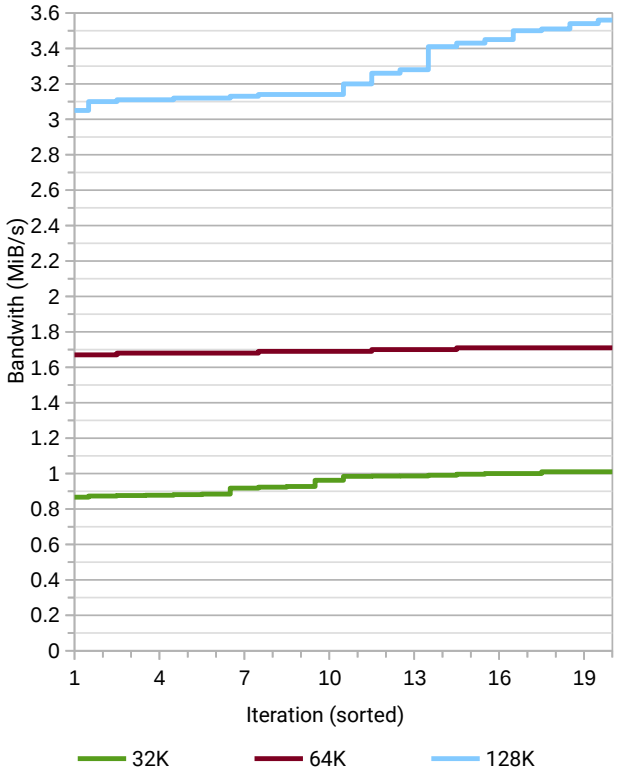
(a) 1M filesize



(b) 10M filesize

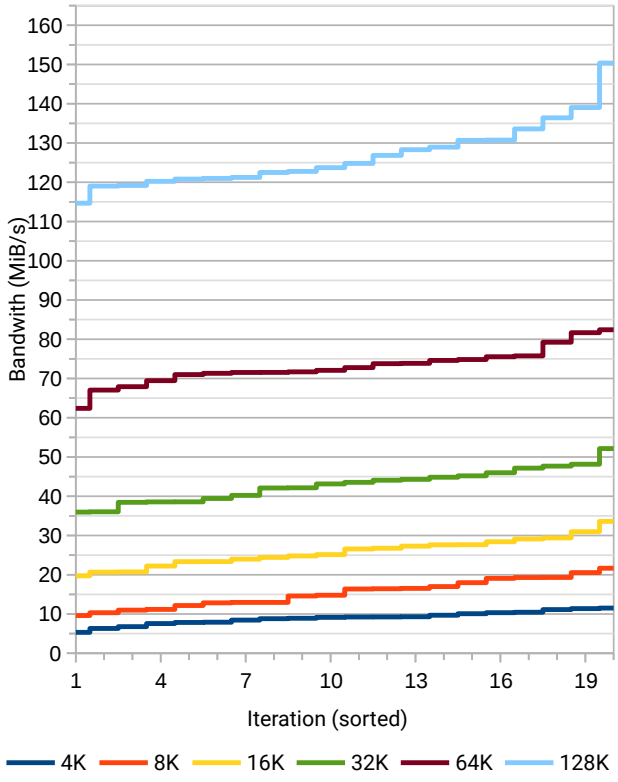


(c) 100M filesize

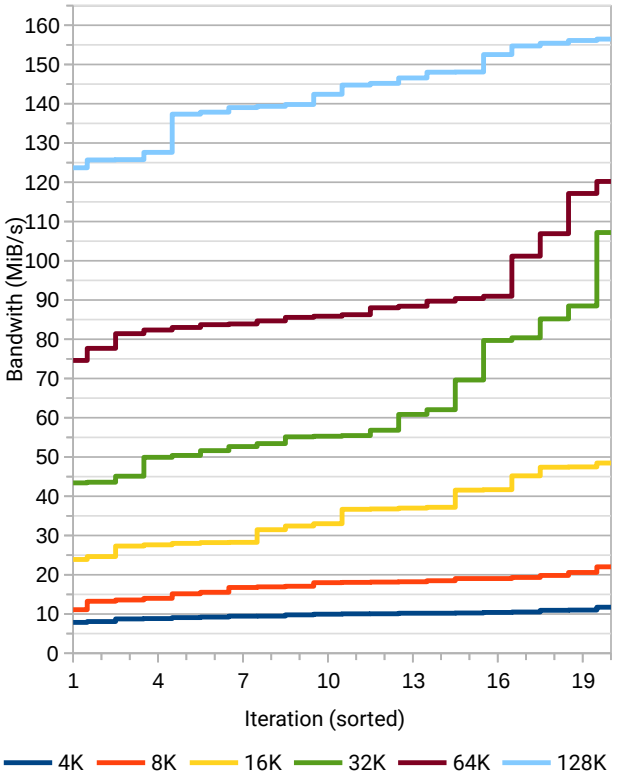


(d) 1G filesize

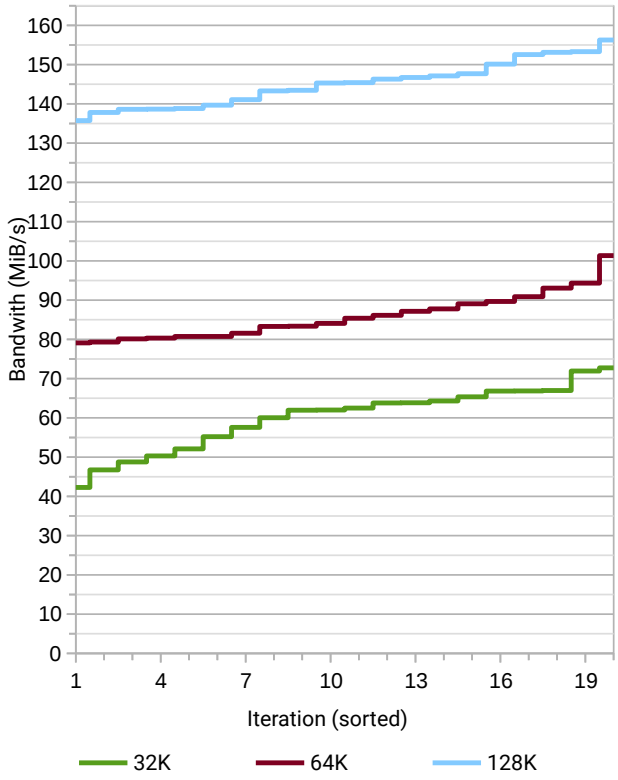
Figure 18: IOR benchmark with write operations at different block sizes (iterations are sorted by throughput)



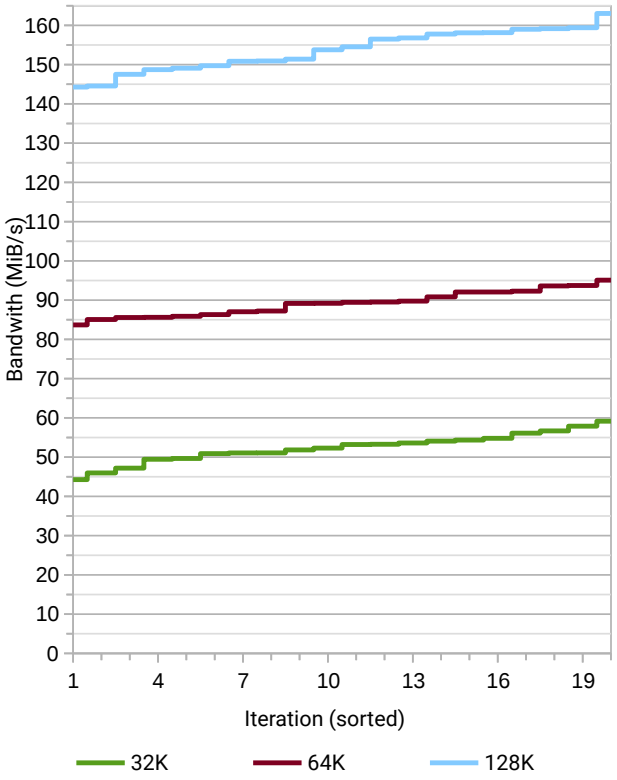
(a) 1M filesize



(b) 10M filesize

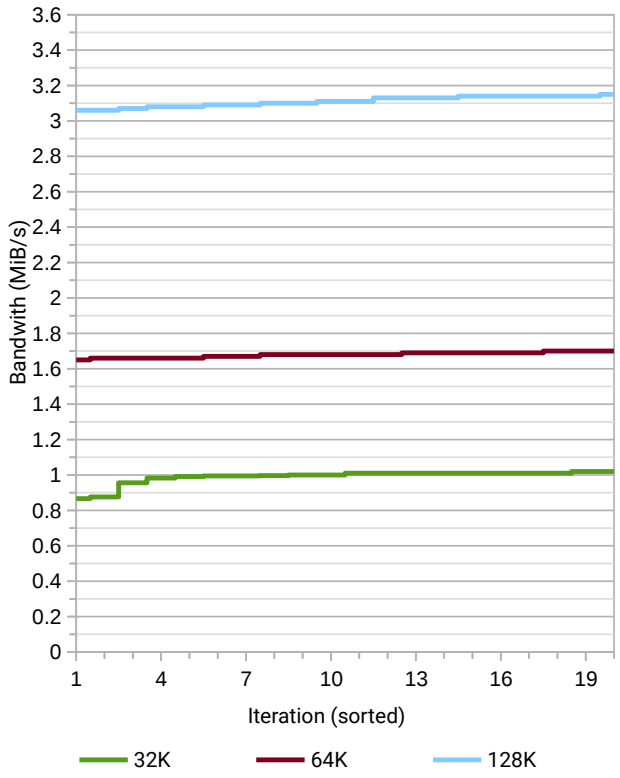


(c) 100M filesize

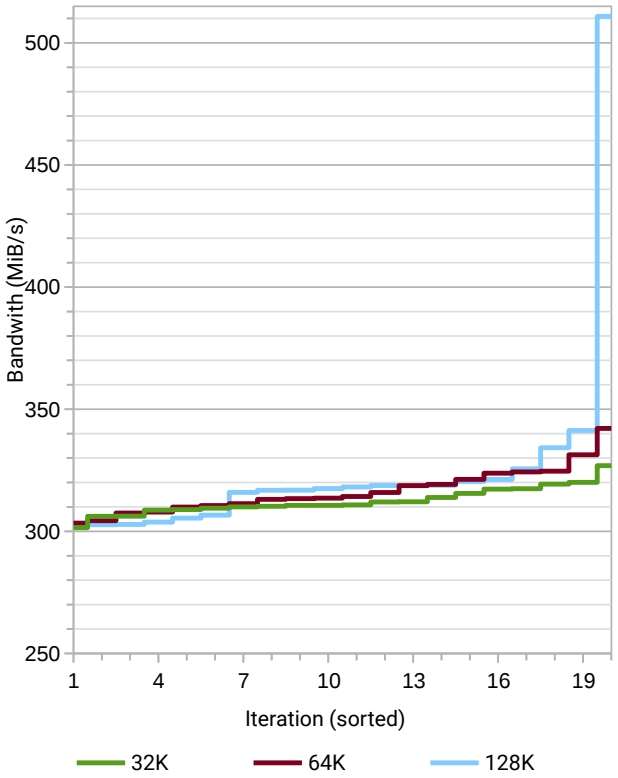


(d) 1G filesize

Figure 19: IOR benchmark with read operations at different block sizes (iterations are sorted by throughput)



(a) Write operations



(b) Read operations

Figure 20: IOR benchmark with 1G filesizes and without Direct IO (iterations are sorted by throughput)

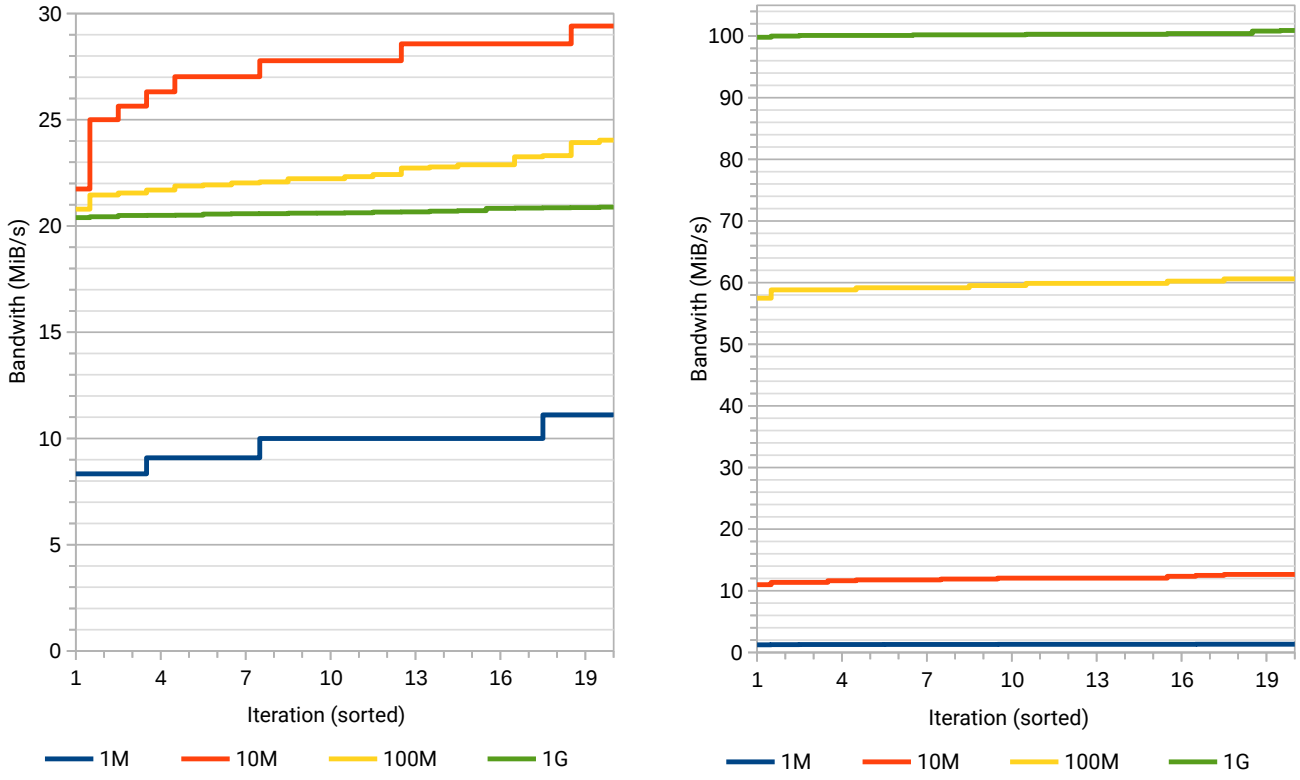
7.4 Results from RADOS

Writing

	1M	10M	100M	1G
Average (MiB/s)	9.7348	27.4459	22.4197	20.6449
Median (MiB/s)	10.0000	27.7778	22.2718	20.6122
Standard Deviation (MiB/s)	0.8565	1.7695	0.8201	0.1499

Reading

	1M	10M	100M	1G
Average (MiB/s)	1.3043	11.9779	59.5689	100.2713
Median (MiB/s)	1.3158	12.0482	59.7020	100.2507
Standard Deviation (MiB/s)	0.0254	0.4489	0.7718	0.2492



(a) Write operations

(b) Read operations

Figure 21: Benchmark with different filesizes while using RADOS

7.5 Evaluation

The measuring results of the *FUSE* driver are quite unsurprising. As seen in figures 18a, 18b, 18c and 18d, the writing performance via *FUSE* is not that fast as it could be (c.f. to 21a). The average throughput of *FUSE* started at 0.07MiB/s and ended at 3.27MiB/s. Compared to this, native writing reached 9.73 MiB/s to 27.78 MiB/s. The differences between small and large block sizes are enormous. Each of the datasets proves that a smaller block size is responsible for a massive performance issue.

The problem of *FUSE* is, the more callbacks are requested – caused by small block sizes – the more context switches are needed (as described in section 6.2). Because of that, *FUSE* can be very slow.

Also, for each write callback *Ceph* has to allocate space, open a file descriptor, change or append data and close the handler. This caused a very low throughput.

Another interesting fact is that a larger filesize (c.f. 1 M + 10 M to 100 M + 1 G) increased the average throughput. It is likely that initializing write operations took some time till it reached an acceptable transmission rate. Accordingly, 1 M and 10 M are an inadequate measuring size to determine average throughput.

A curious discrepancy among half of the measure data appeared in figure 18c. The difference between these two groups amounts to 0.4 MiB/s, but this is, unfortunately, inexplicable so far.

The next four figures (19a, 20b, 19c and 20a) present the read throughput at different file and block sizes using *FUSE*. Thereby, the file size is not a significant factor while reading. The differences between 1 M, 10 M, 100 M and 1 G are small. However, larger block sizes increased read performance, as seen before.

Again, there is an anomaly in the data. The standard deviation of the dataset *10 M file size, 32 K block size* is abnormal high: 17.2743 MiB/s.

The last two IOR benchmarks (figure {**fig:1g_write_dio**} and {**fig:1g_write_dio**}) evaluate the performance without the option *"Direct IO"*. But the differences between writing with or without *Direct IO* are not significant. In contrast, reading is highly significant faster than without this option. The reasons for this are memory caches because there are bare HDD requests.

The other benchmark method shows the limits of *Ceph Storage Cluster* in this scenario.

Figure 21a clearly shows how slow a *FUSE* implementation is. The throughput in writing is around ten times higher. Interestingly, writing 10M to disk is around 5 MiB/s faster than 100 M or 1 G with a difference of 7 MiB/s. Perhaps, allocating larger disk space takes more time than expected. On the other hand, the standard deviation was smaller at larger file sizes – almost a constant throughput.

Reading objects from *Ceph* is another important point. As seen in graph 21b, throughput varied with respect to file size. Presumably, the initialization of `rados` and requesting an object over network took more time than transferring it to the client. Therefore, *Ceph* is not suitable for many small files, but good in archiving and reading large files. But, the difference between *FUSE* and native reading came from the fact that *IOR* did not eliminate kernel's *PageCache*. From this point of view, `rados` is much faster.

8 Conclusion

Is *Ceph* suitable as a storage solution for *HPC*? In my estimation, it is an eligible distributed file system with reservations.

At first, *Ceph* is scalable and extendable, which is an important characteristic. It is possible to add or remove *OSDs* and monitors at any time to change the topology of an existing cluster. Besides, *OSDs* do not have to contain similar hardware configurations, which is useful if a new rack with state-of-the-art hardware is put into operation.

Furthermore, thanks to *CRUSH* maps monitors do not have to store object location and therefore no big tables like *inodes* are needed. On the other hand, objects can not be renamed. If setting a new name is indispensable, the object has to be rewritten.

Another point is the simple API of `librados`, which can be used to manage and transfer objects in the cluster. It is well documented and bindings are available in *C*, *C++*, *Python* and *Java*; other languages like *Rust* are unofficially supported, too

As seen in the chapter 7, which is about benchmarking, the proof-of-concept *FUSE* driver is not very suitable if it comes to fast I/O throughput. It is a better solution to implement `librados` into a high-performance application to take the advantages of *Ceph's* performance. That means further work because there are no interfaces for *MPI-IO* or *HDF5*.

Also, as described in section 8.2.2, a combination of *Ceph* as an object storage cluster and a database as a metadata server could be a fast and user-optimized way to improve current data storage solution.

8.1 Future work

My estimation was with reservations. One reason for this is that a benchmark on a real cluster with more than a handful nodes has to be done. This is the only way to estimate the performance in comparison to other storage solutions like *Lustre*.

There are some characteristics of *Ceph*, which are not mentioned yet. This includes caching of object and tiering of hosts, which should also be evaluated in the test as mentioned above.

Furthermore, implementing an interface to create a transparent layer while splitting data and metadata into different storages endpoints has to be examined. Because it is important to know its performance compared to the current solution.

Another important point needs an evaluation: the backup/restore algorithms. Backups are important and great until the restore function fails. Therefore, a test with random failures like offline hosts or faulty storage devices has to be done.

8.2 Design of a fictional HPC Ceph Cluster

This section is about a fictional *HPC* application whose results has to be stored in a *Ceph* cluster. Thereby, this application generates the very common filetype *HDF5*.

The proposal contains a cluster structure, separation of metadata and an I/O interface.

8.2.1 HDF5

The Hierarchical Data Format (5th version) is a self-describing data format, stored in a single file, developed by *The HDF Group* and released under a *BSD*-like license.

It contains a hierarchical *POSIX*-like directory structure with grouped datasets. These datasets consist of multidimensional, homogeneous, numerical arrays, as seen in figure 22. Also, *HDF5* supports lossless compression of these.

Furthermore, it is possible to add comprehensive metadata to each group of datasets to describe presented data. [19]

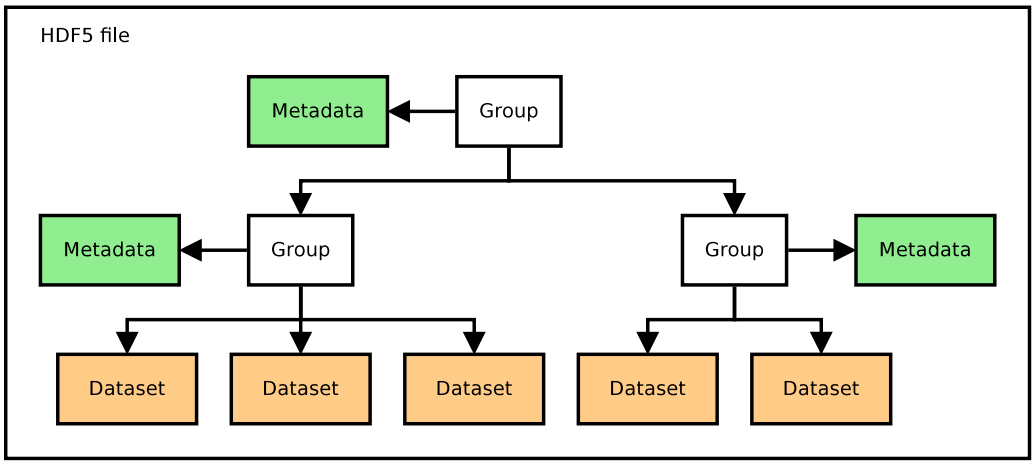


Figure 22: Exemplary structure of a simple HDF5 file

8.2.2 Metadata Server

The fact that each group of an *HDF5* file could contain searchable metadata it is eligible to separate metadata and data. The detachment makes it possible to store the metadata in a document-oriented database, for example *MongoDB* or *Elasticsearch*. Therefore, clients and their users can find a specific dataset much faster than crawling through all *HDF5* files. Moreover, these databases have the opportunity to replicate and distribute data over a database cluster.

For the reason that metadata in *HDF5* files are assigned to tree like groups and contains different key-value pairs, it makes sense to organize and map this structure into a document-oriented database. Thus, the original structure is preserved and visible to the user (figure 23.

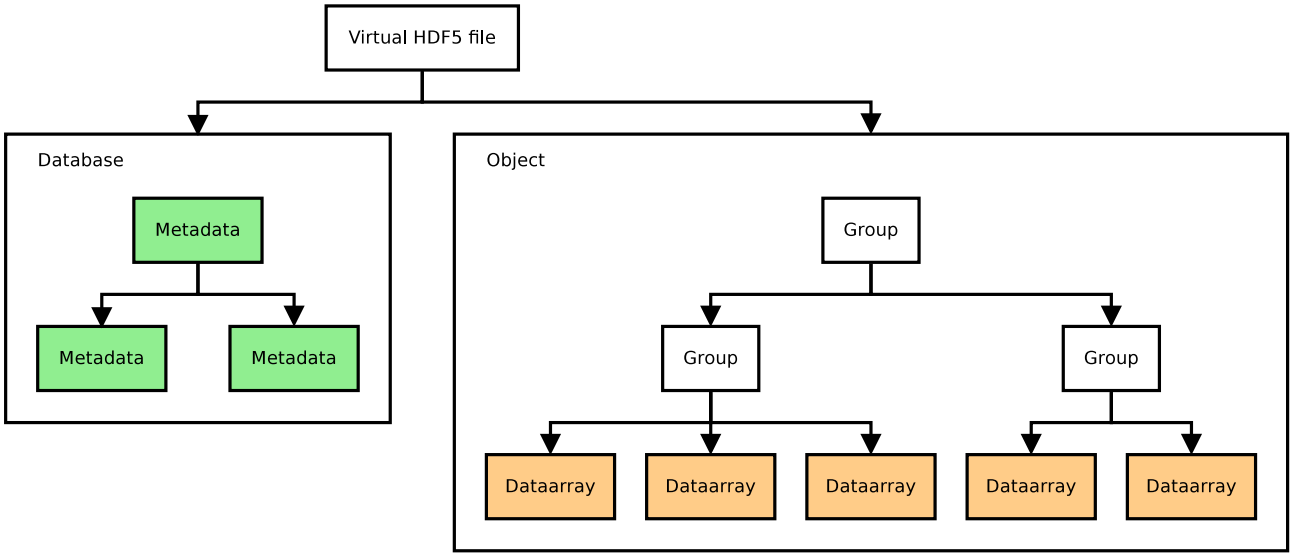


Figure 23: Visualisation of a virtual HDF5 file

8.2.3 Ceph Cluster

The size and amount of nodes in a cluster is coupled to required throughput and quantity of stored data. Accordingly, this proposal offers a structural overview of a Ceph cluster.

There must be an odd number of monitors. Additionally, it is wise to use *SSDs* and a high memory capacity, comparable to a database server, so that initial client requests are processed fast.

Also, a combination of different *HDD* sizes can be placed on the storage nodes *(OSDs)*. Thereby, *OSDs* with lower capacity must be weighted lower in the global *CRUSH* map to ensure that data is distributed uniformly.

Figure 24 shows a possible cluster map hierarchy, which is deposited in the monitors for a *CRUSH* map. In this scenario, there are two server rooms with rows of racks, containing *OSD* hosts.

8.2.4 I/O Interface

As seen in figure 25 the best solution for efficient read/write operations is to implement an I/O interface directly into *HPC* application. A great adventage over *MPI-IO* or a *POSIX/FUSE* solution is the reduction of complexity by removing unnecessary layers and in the case of *FUSE* decreasing context switches.

Of course, it is possible to program an adapter for applications using *MPI-IO* to simplify integration with many application, but this will decrease performance. More layers would be added to the application and the storage cluster, which caused more callbacks and context switches.

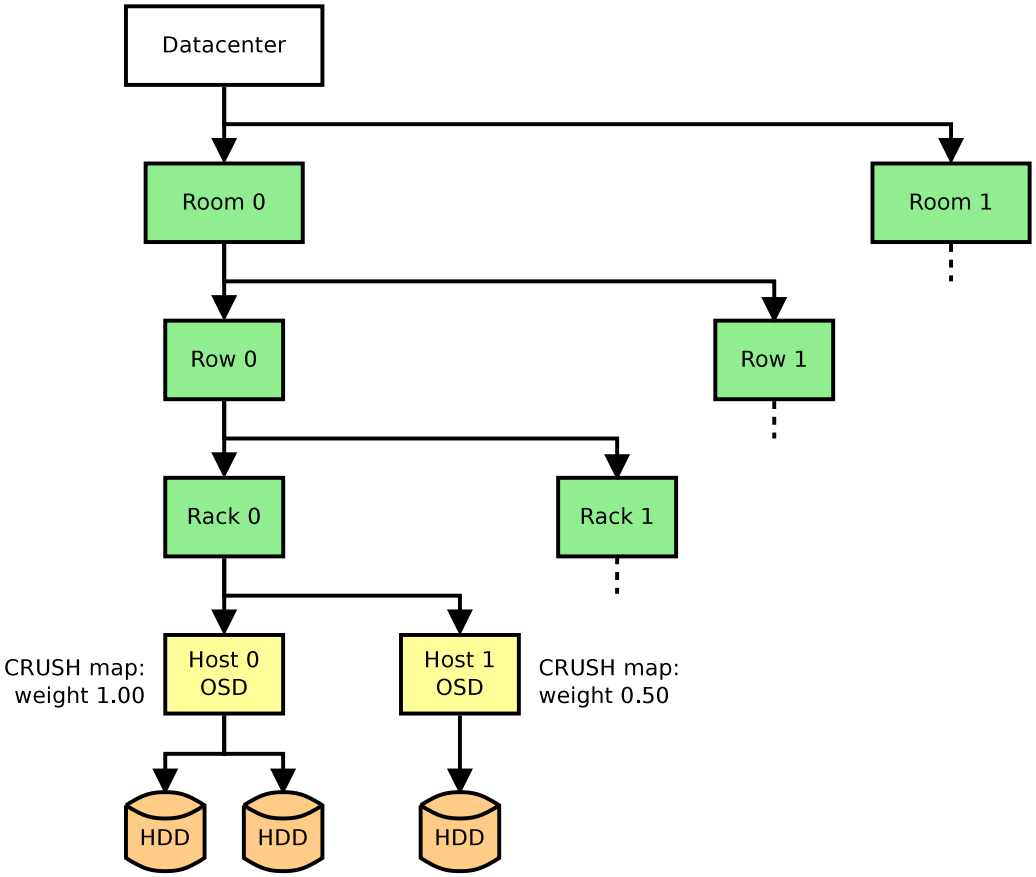


Figure 24: Structural cluster design

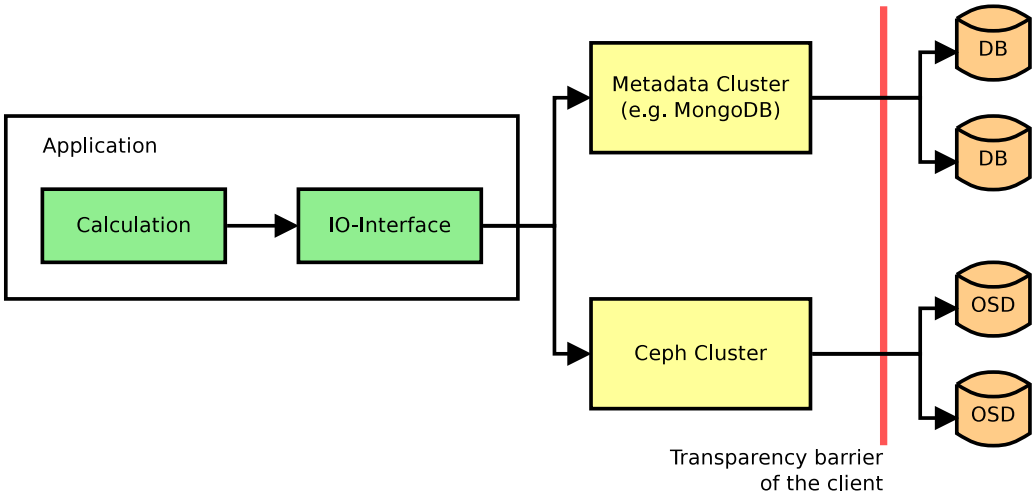


Figure 25: Application design with an I/O interface

9 Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Hamburg, den 23. März 2017

Lars Thoms

List of Figures

References

[1] Andrew S. Tanenbaum; Herbert Bos. *Modern operating systems*. 4th ed. Pearson, 2015. ISBN: 1-292-06142-1.

[2] Doxygen. *libfuse API documentation*. URL: https://libfuse.github.io/doxygen/index.html (visited on 02/18/2017).

[3] Gluster. *GlusterFS Documentation*. URL: http://gluster.readthedocs.io/en/latest/Quick-Start-Guide/Terminologies (visited on 11/25/2016).

[4] Sébastien Han. *Ceph: manage storage zones with CRUSH*. URL: https://www.sebastien-han.fr/blog/2012/12/07/ceph-2-speed-storage-with-crush/ (visited on 03/01/2017).

[5] IEEE. *IEEE Std 1003.1-2008*. 2016. URL: http://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 11/13/2016).

[6] Inktank Storage Inc. *Storage Cluster Quick Start*. URL: http://docs.ceph.com/docs/giant/start/quick-ceph-deploy/ (visited on 02/07/2017).

[7] Red Hat Inc. *CRUSH Maps*. URL: http://docs.ceph.com/docs/master/rados/operations/crush-map/ (visited on 03/01/2017).

[8] Red Hat Inc. *Librados (C) – API call*. URL: http://docs.ceph.com/docs/master/rados/api/librados/#api-calls (visited on 02/18/2017).

[9] Yves Kemp. *BeeGFS at DESY*. URL: https://indico.cern.ch/event/346931/contributions/817830/attachments/684674/940478/beegfs-at-desy.pdf (visited on 11/25/2016).

[10] Michael Kuhn. "Dynamically Adaptable I/O Semantics for High Performance Computing". PhD Thesis. Universität Hamburg, Apr. 2015. URL: http://ediss.sub.uni-hamburg.de/volltexte/2015/7302/.

[11] Julian Martin Kunkel; Michael Kuhn; Thomas Ludwig. "Exascale Storage Systems – An Analytical Study of Expenses". In: (2014). DOI: 10.14529/jsfi140106.

[12] Lustre. *Lustre Operations Manual*. URL: http://doc.lustre.org/lustre_manual.xhtml (visited on 11/25/2016).

[13] Sage A. Weil; Andrew W. Leung; Scott A. Brandt; Carlos Maltzahn. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters". In: *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*. PDSW '07. Reno, Nevada: ACM, 2007, pp. 35–44. ISBN: 978-1-59593-899-2. DOI: 10.1145/1374596.1374606. URL: http://doi.acm.org/10.1145/1374596.1374606.

[14] Sage A. Weil; Scott A. Brandt; Ethan L. Miller; Carlos Maltzahn. "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188582. URL: http://doi.acm.org/10.1145/1188455.1188582.

[15] Sage A. Weil; Scott A. Brandt; Ethan L. Miller; Darrell D. E. Long; Carlos Maltzahn. "Ceph: A Scalable, High-performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298485.

[16] Timothy Prickett Morgan. *Red Hat snatches storage Gluster file system for $136m*. URL: http://www.theregister.co.uk/2011/10/04/redhat_buys_gluster (visited on 11/25/2016).

[17] Steve D. Pate. *UNIX Filesystems*. Wiley, 2003. ISBN: 0-471-16483-6.

[18] Mike Mesnier; Greg Ganger; Erik Riedel. "Object-based storage". In: (2003). DOI: 10.1109/MCOM.2003.1222722.

[19] PhD Cyrille Rossant. *Moving away from HDF5*. URL: http://cyrille.rossant.net/moving-away-hdf5/ (visited on 03/08/2017).

[20] Karan Singh. *How Data Is Stored In CEPH Cluster*. URL: http://karan-mj.blogspot.de/2014/01/how-data-is-stored-in-ceph-cluster.html (visited on 12/14/2016).

[21] Sumit Singh. *Develop your own filesystem with FUSE*. URL: https://www.ibm.com/developerworks/library/l-fuse/ (visited on 02/22/2017).

[22] Ross Turk. *Architecture diagram showing the component relation of the Ceph storage platform*. URL: https://raw.githubusercontent.com/ceph/ceph/master/doc/images/stack.png (visited on 01/18/2017).

[23] Sage A. Weil. "Ceph: Reliable, Scalable, and High-performance Distributed Storage". PhD thesis. University of California Santa Cruz, 2007.

[24] Sage A. Weil. *Erasure coding and cache tiering*. URL: http://www.slideshare.net/sageweil1/20150222-scale-sdc-tiering-and-ec (visited on 01/19/2017).

[25] Ext4 Wiki. *Ext4 Disk Layout*. URL: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Layout (visited on 11/13/2016).

Attachment

IOR / 1M filesize / variable blocksize / write operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	0.062748	0.123691	0.230038	0.421136	0.784773	1.43
2	0.063273	0.125761	0.230310	0.446140	0.805621	1.45
3	0.063646	0.126278	0.234361	0.449433	0.845283	1.49
4	0.063981	0.132703	0.264042	0.463360	0.876203	1.58
5	0.064295	0.134210	0.264305	0.489973	0.901641	1.60
6	0.064672	0.137405	0.264803	0.493949	0.909593	1.65
7	0.065098	0.141820	0.266547	0.497531	0.916524	1.67
8	0.066279	0.141822	0.267825	0.502191	0.930787	1.74
9	0.066832	0.142150	0.273309	0.510712	0.945284	1.79
10	0.067250	0.142158	0.275202	0.518814	0.960403	1.83
11	0.067738	0.145787	0.276482	0.519052	0.983613	1.85
12	0.069182	0.147757	0.277743	0.526358	0.997362	1.88
13	0.070181	0.149484	0.278418	0.533086	1.01	1.88
14	0.071582	0.149978	0.279055	0.533423	1.01	1.90
15	0.072223	0.150729	0.279722	0.540600	1.02	1.91
16	0.072240	0.150887	0.280992	0.547961	1.03	1.91
17	0.073022	0.151110	0.281023	0.548027	1.03	1.91
18	0.073505	0.152064	0.281644	0.550582	1.04	1.94
19	0.073642	0.155815	0.282914	0.568851	1.06	1.97
20	0.078200	0.156628	0.290869	0.571472	1.09	2.00

IOR / 10M filesize / variable blocksize / write operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	0.064674	0.129916	0.252512	0.480870	0.991532	1.80
2	0.065916	0.131772	0.255577	0.483191	1.00	1.86
3	0.066295	0.133696	0.265414	0.488678	1.01	1.88
4	0.066707	0.134745	0.268381	0.503429	1.02	1.93
5	0.066822	0.136152	0.270923	0.508777	1.03	2.01
6	0.067130	0.136538	0.272454	0.509886	1.05	2.03
7	0.067261	0.136872	0.272559	0.513081	1.07	2.06
8	0.067541	0.137053	0.276995	0.515339	1.11	2.09
9	0.067827	0.137259	0.277301	0.524786	1.12	2.12
10	0.068158	0.137414	0.278084	0.533685	1.13	2.13
11	0.068321	0.138777	0.278663	0.534880	1.13	2.13
12	0.068553	0.138832	0.278923	0.535795	1.15	2.15
13	0.068701	0.139620	0.279823	0.538446	1.15	2.15
14	0.069177	0.140238	0.280408	0.548566	1.15	2.16
15	0.069381	0.140309	0.280683	0.555658	1.15	2.17
16	0.069625	0.140555	0.282482	0.570164	1.16	2.17
17	0.070298	0.143244	0.283334	0.595625	1.17	2.20
18	0.071044	0.143855	0.283939	0.596810	1.19	2.22
19	0.071330	0.144218	0.285358	0.601598	1.19	2.26
20	0.071565	0.146632	0.297380	0.612359	1.20	2.32

IOR / 100M filesize / variable blocksize / write operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	0.511391	1.67	2.99
2	n/a	n/a	n/a	0.512702	1.67	3.04
3	n/a	n/a	n/a	0.541388	1.69	3.12

Iteration	4K	8K	16K	32K	64K	128K
4	n/a	n/a	n/a	0.549397	1.69	3.14
5	n/a	n/a	n/a	0.550298	1.69	3.14
6	n/a	n/a	n/a	0.556560	1.70	3.15
7	n/a	n/a	n/a	0.560595	1.71	3.16
8	n/a	n/a	n/a	0.562824	1.72	3.16
9	n/a	n/a	n/a	0.563428	1.74	3.19
10	n/a	n/a	n/a	0.602301	1.80	3.19
11	n/a	n/a	n/a	0.978276	1.84	3.19
12	n/a	n/a	n/a	0.983901	1.85	3.20
13	n/a	n/a	n/a	1.00	1.86	3.21
14	n/a	n/a	n/a	1.01	1.86	3.21
15	n/a	n/a	n/a	1.02	1.87	3.22
16	n/a	n/a	n/a	1.03	1.93	3.22
17	n/a	n/a	n/a	1.03	1.94	3.32
18	n/a	n/a	n/a	1.03	1.94	3.35
19	n/a	n/a	n/a	1.03	1.95	3.42
20	n/a	n/a	n/a	1.03	1.95	3.45

IOR / 1G filesize / variable blocksize / write operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	0.866656	1.67	3.05
2	n/a	n/a	n/a	0.873445	1.67	3.10
3	n/a	n/a	n/a	0.876178	1.68	3.11
4	n/a	n/a	n/a	0.877902	1.68	3.11
5	n/a	n/a	n/a	0.881121	1.68	3.12
6	n/a	n/a	n/a	0.884385	1.68	3.12
7	n/a	n/a	n/a	0.918111	1.68	3.13
8	n/a	n/a	n/a	0.923424	1.69	3.14
9	n/a	n/a	n/a	0.927250	1.69	3.14
10	n/a	n/a	n/a	0.962354	1.69	3.14
11	n/a	n/a	n/a	0.984536	1.69	3.20
12	n/a	n/a	n/a	0.986219	1.70	3.26
13	n/a	n/a	n/a	0.986631	1.70	3.28
14	n/a	n/a	n/a	0.990702	1.70	3.41
15	n/a	n/a	n/a	0.996332	1.71	3.43
16	n/a	n/a	n/a	1.00	1.71	3.45
17	n/a	n/a	n/a	1.00	1.71	3.50
18	n/a	n/a	n/a	1.01	1.71	3.51
19	n/a	n/a	n/a	1.01	1.71	3.54
20	n/a	n/a	n/a	1.01	1.71	3.56

IOR / 1G filesize / variable blocksize / write operation / without *Direct IO* / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	0.866442	1.65	3.06
2	n/a	n/a	n/a	0.876175	1.66	3.06
3	n/a	n/a	n/a	0.955648	1.66	3.07
4	n/a	n/a	n/a	0.982942	1.66	3.08
5	n/a	n/a	n/a	0.991272	1.66	3.08
6	n/a	n/a	n/a	0.994531	1.67	3.09
7	n/a	n/a	n/a	0.994597	1.67	3.09
8	n/a	n/a	n/a	0.996249	1.68	3.10
9	n/a	n/a	n/a	1.00	1.68	3.10
10	n/a	n/a	n/a	1.00	1.68	3.11
11	n/a	n/a	n/a	1.01	1.68	3.11
12	n/a	n/a	n/a	1.01	1.68	3.13

Iteration	4K	8K	16K	32K	64K	128K
13	n/a	n/a	n/a	1.01	1.69	3.13
14	n/a	n/a	n/a	1.01	1.69	3.13
15	n/a	n/a	n/a	1.01	1.69	3.14
16	n/a	n/a	n/a	1.01	1.69	3.14
17	n/a	n/a	n/a	1.01	1.69	3.14
18	n/a	n/a	n/a	1.01	1.70	3.14
19	n/a	n/a	n/a	1.02	1.70	3.14
20	n/a	n/a	n/a	1.02	1.70	3.15

IOR / 1M filesize / variable blocksize / read operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	5.31	9.61	19.73	35.97	62.39	114.67
2	6.30	10.32	20.68	36.04	67.04	119.03
3	6.77	10.99	20.73	38.45	67.91	119.18
4	7.56	11.18	22.22	38.56	69.44	120.23
5	7.83	12.17	23.34	38.58	70.99	120.80
6	7.91	12.85	23.36	39.43	71.32	120.98
7	8.45	12.95	23.99	40.22	71.53	121.23
8	8.81	12.96	24.42	42.11	71.54	122.50
9	8.92	14.60	24.80	42.15	71.68	122.75
10	9.15	14.78	25.13	43.14	72.07	123.71
11	9.24	16.38	26.57	43.54	72.77	124.80
12	9.26	16.44	26.76	44.09	73.77	126.84
13	9.31	16.53	27.30	44.30	73.85	128.30
14	9.69	16.99	27.62	44.85	74.60	128.95
15	10.08	17.98	27.65	45.20	74.83	130.68
16	10.35	19.08	28.41	45.99	75.54	130.74
17	10.45	19.30	29.13	47.17	75.75	133.58
18	11.13	19.32	29.39	47.68	79.24	136.42
19	11.38	20.56	30.97	48.15	81.67	139.03
20	11.52	21.67	33.59	52.15	82.40	150.35

IOR / 10M filesize / variable blocksize / read operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	7.87	11.10	23.90	43.40	74.60	123.69
2	8.09	13.25	24.64	43.58	77.68	125.68
3	8.75	13.57	27.32	45.10	81.42	125.75
4	8.84	13.99	27.63	49.92	82.34	127.61
5	9.09	15.16	27.99	50.40	82.99	137.35
6	9.23	15.54	28.20	51.62	83.70	137.86
7	9.46	16.77	28.26	52.65	83.89	139.01
8	9.47	16.91	31.48	53.42	84.66	139.34
9	9.78	17.07	32.42	55.13	85.57	139.80
10	9.95	17.99	33.01	55.29	85.85	142.41
11	10.05	18.06	36.68	55.45	86.25	144.75
12	10.07	18.14	36.76	56.82	88.02	145.19
13	10.21	18.21	36.98	60.84	88.43	146.56
14	10.21	18.48	37.19	62.04	89.71	148.03
15	10.27	19.03	41.56	69.61	90.38	148.09
16	10.41	19.03	41.67	79.69	90.96	152.53
17	10.51	19.31	45.20	80.36	101.17	154.72
18	10.94	19.82	47.40	85.17	106.90	155.42
19	11.01	20.58	47.47	88.48	117.14	156.11
20	11.74	22.02	48.47	107.19	120.21	156.46

IOR / 100M filesize / variable blocksize / read operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	42.28	79.09	135.74
2	n/a	n/a	n/a	46.75	79.32	137.81
3	n/a	n/a	n/a	48.77	80.11	138.61
4	n/a	n/a	n/a	50.30	80.32	138.66
5	n/a	n/a	n/a	52.10	80.73	138.82
6	n/a	n/a	n/a	55.21	80.74	139.67
7	n/a	n/a	n/a	57.58	81.55	141.08
8	n/a	n/a	n/a	60.03	83.29	143.30
9	n/a	n/a	n/a	61.95	83.35	143.45
10	n/a	n/a	n/a	62.00	84.07	145.32
11	n/a	n/a	n/a	62.50	85.37	145.41
12	n/a	n/a	n/a	63.79	86.12	146.31
13	n/a	n/a	n/a	63.84	87.16	146.72
14	n/a	n/a	n/a	64.31	87.78	147.12
15	n/a	n/a	n/a	65.36	89.08	147.68
16	n/a	n/a	n/a	66.82	89.66	150.13
17	n/a	n/a	n/a	66.86	90.89	152.56
18	n/a	n/a	n/a	66.96	93.06	153.13
19	n/a	n/a	n/a	71.94	94.33	153.29
20	n/a	n/a	n/a	72.74	101.34	156.26

IOR / 1G filesize / variable blocksize / read operation / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	44.27	83.67	144.27
2	n/a	n/a	n/a	45.97	85.06	144.56
3	n/a	n/a	n/a	47.19	85.56	147.52
4	n/a	n/a	n/a	49.46	85.61	148.72
5	n/a	n/a	n/a	49.66	85.88	149.09
6	n/a	n/a	n/a	50.89	86.30	149.72
7	n/a	n/a	n/a	51.06	87.05	150.85
8	n/a	n/a	n/a	51.08	87.24	150.93
9	n/a	n/a	n/a	51.83	89.17	151.40
10	n/a	n/a	n/a	52.30	89.20	153.78
11	n/a	n/a	n/a	53.21	89.44	154.54
12	n/a	n/a	n/a	53.28	89.53	156.50
13	n/a	n/a	n/a	53.60	89.76	156.80
14	n/a	n/a	n/a	54.08	90.84	157.78
15	n/a	n/a	n/a	54.34	92.07	158.09
16	n/a	n/a	n/a	54.79	92.08	158.14
17	n/a	n/a	n/a	56.10	92.30	159.01
18	n/a	n/a	n/a	56.68	93.63	159.16
19	n/a	n/a	n/a	57.88	93.73	159.39
20	n/a	n/a	n/a	59.15	95.09	163.02

IOR / 1G filesize / variable blocksize / read operation / without *Direct IO* / bandwith in MiB

Iteration	4K	8K	16K	32K	64K	128K
1	n/a	n/a	n/a	301.48	303.38	302.57
2	n/a	n/a	n/a	306.16	304.36	302.70
3	n/a	n/a	n/a	306.20	307.51	302.83
4	n/a	n/a	n/a	308.69	307.90	303.77
5	n/a	n/a	n/a	308.89	309.93	305.42
6	n/a	n/a	n/a	309.48	310.56	306.63
7	n/a	n/a	n/a	310.00	311.33	315.95

Iteration	4K	8K	16K	32K	64K	128K
8	n/a	n/a	n/a	310.27	313.08	316.83
9	n/a	n/a	n/a	310.64	313.38	316.88
10	n/a	n/a	n/a	310.64	313.58	317.51
11	n/a	n/a	n/a	310.83	314.25	318.17
12	n/a	n/a	n/a	312.03	315.90	318.77
13	n/a	n/a	n/a	312.10	318.71	318.79
14	n/a	n/a	n/a	313.87	319.20	318.87
15	n/a	n/a	n/a	315.54	321.28	320.42
16	n/a	n/a	n/a	317.30	323.80	321.26
17	n/a	n/a	n/a	317.45	324.34	325.59
18	n/a	n/a	n/a	319.34	324.64	334.22
19	n/a	n/a	n/a	320.06	331.35	341.28
20	n/a	n/a	n/a	326.89	342.14	510.84

RADOS / variable filesize / write operation / duration in seconds

Iteration	1M	10M	100M	1G
1	0.12	0.46	4.81	49.03
2	0.12	0.40	4.66	48.94
3	0.12	0.39	4.64	48.79
4	0.11	0.38	4.61	48.78
5	0.11	0.37	4.57	48.76
6	0.11	0.37	4.56	48.64
7	0.11	0.37	4.54	48.60
8	0.10	0.36	4.53	48.59
9	0.10	0.36	4.50	48.54
10	0.10	0.36	4.50	48.53
11	0.10	0.36	4.48	48.50
12	0.10	0.36	4.46	48.42
13	0.10	0.35	4.40	48.40
14	0.10	0.35	4.39	48.31
15	0.10	0.35	4.37	48.26
16	0.10	0.35	4.37	48.01
17	0.10	0.35	4.30	47.98
18	0.09	0.35	4.29	47.94
19	0.09	0.34	4.18	47.92
20	0.09	0.34	4.16	47.87

RADOS / variable filesize / read operation / duration in seconds

Iteration	1M	10M	100M	1G
1	0.81	0.91	1.74	10.02
2	0.79	0.88	1.70	10.00
3	0.78	0.88	1.70	9.99
4	0.78	0.86	1.70	9.99
5	0.78	0.85	1.69	9.99
6	0.77	0.85	1.69	9.99
7	0.77	0.85	1.69	9.98
8	0.77	0.84	1.69	9.98
9	0.77	0.84	1.68	9.98
10	0.76	0.83	1.68	9.98
11	0.76	0.83	1.67	9.97
12	0.76	0.83	1.67	9.97
13	0.76	0.83	1.67	9.97
14	0.76	0.83	1.67	9.97
15	0.76	0.83	1.67	9.97
16	0.76	0.81	1.66	9.96

Iteration	1M	10M	100M	1G
17	0.75	0.80	1.66	9.96
18	0.75	0.79	1.65	9.96
19	0.75	0.79	1.65	9.92
20	0.75	0.79	1.65	9.91