Marius Oltean
Jose L. Muñoz

UPC Telematics Department

Linux Storage Systems

Contents

Chapter 1

Linux Containers

1.1 Introduction

With several functionalities added to the Linux kernel, it has become very easy to isolate Linux processes into their own little environments. Isolation tools allow to build containers, which are a lightweight virtualization technology. While hardware virtualizations or para-virtualizations provide virtual machines, containers are an operating system-level virtualization method for running multiple isolated Linux systems (containers) on a single host. With containers, **a single Linux kernel is shared between the host and the virtual machines**. Containers can achieve higher densities of isolated environments than when using virtual machines.

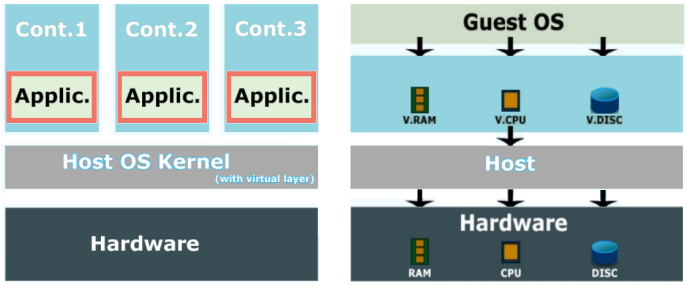


Figure 1.1: Lxc vs VM.

This concept is not new, as it was implemented a few years ago in BSD jails, Solaris Zones and other open-source projects. In Linux, one of the most widely known container technologies has been OpenVZ (OVZ). What OVZ does is, when installed, loads up a specialized kernel that allows us to segment disk space, CPU and memory usage, as well as other features such as bandwidth per container, and locks them in to that specific area of the server. Many engineers from OVZ contributed to the official Linux kernel to build standard isolation tools[1]. These tools are essentially **namespaces**, **cgroups** and **special network interfaces**. These are described below as well as the two main approaches to build containers: LXC (LinuX Containers) and Docker.

1.2 Namespaces

The key tool to build LXC containers is kernel namespaces, which is a specific feature of the Linux kernel to isolate applications from each other. Kernel namespaces provide a way to have varying views of the system for different

[1] As long as we have a kernel version >= 2.6.29 complete functionality exists.

processes. What that means is to have the capability to put applications in isolated environments with separate process lists, network devices, user lists and filesystems. Such a functionality is implemented inside the kernel without the need to run hypervisors or virtualization. The OS needs to keep separate internal structures and make sure they remain isolated. Kernel namespaces are created and manipulated using 3 basic syscalls:

- `clone()` which is used to create a new children processes (this low-level function sits behind the well known `fork()`).
- `unshare()` which allows to modify the execution context of a process without spawning a new child process.
- `setns()` which changes the namespace of the calling process.

The `clone()` system call supports a number of flags, allowing to specify for example, that we want the new process to run within its own namespace. When creating a container, this is exactly what happens: a new process, with new namespace, is created; its network interfaces (including the special pair of interfaces to talk with the outside world) are configured; and it executes an init-like process.

Each namespace is materialized by a special file in /proc/$PID/ns. When the last process within a namespace exits, the associated resources: network interfaces, etc. are automatically reclaimed. It is also possible to "enter" a namespace, by attaching a process to an existing namespace. This is generally used to run an arbitrary command within the namespace. Currently, there are six different namespaces, which are described next.

1.2.1 Process Namespace

Historically, the Linux kernel has maintained a single process tree. The tree contains a reference to every process currently running in a parent-child hierarchy. With the introduction of Linux namespaces, it became possible to have multiple "nested" process trees. Each process tree can have an entirely isolated set of processes. This can ensure that processes belonging to one process tree cannot inspect or kill (in fact cannot even know of the existence of) processes in other sibling or parent process trees.

Every time a computer with Linux boots up, it starts with just one process, with process identifier (PID) 1. This process is the root of the process tree, and it initiates the rest of the system by performing the appropriate maintenance work and starting the correct daemons/services. All the other processes start below this process in the tree. The PID namespace allows one to spin off a new tree, with its own PID 1 process. The process that does this remains in the parent namespace, in the original tree, but makes the child the root of its own process tree.

A "parent" namespace can see its children namespaces, and it can affect them (for instance, with signals); but a child namespace cannot do anything to its parent namespace. As a consequence, each pid namespace has its own PID=1 init-like process. Processes living in a namespace cannot affect processes living in parent or sibling namespaces. If a pseudo-filesystem like proc is mounted by a process within a pid namespace, it will only show the processes belonging to the namespace; since the numbering is different in each namespace. This means that a process in a child namespace will have multiple PIDs: one in its own namespace, and a different PID in its parent namespace. This last issue means that from the top-level pid namespace, we will be able to see all processes running in all namespaces, but with different PIDs. Therefore, a process can have more than 2 PIDs if there are more than two levels of hierarchy in the namespaces.

1.2.2 The mnt namespace

The mnt namespace deals with mountpoints. Processes living in different mnt namespaces can see different sets of mounted filesystems and different root directories. If a filesystem is mounted in a mnt namespace, it will be accessible only to those processes within that namespace; it will remain invisible for processes in other namespaces. The mnt namespace allows each container to have its own mountpoints, and see only those mountpoints, with their path correctly translated to the actual root of the namespace. We can test this mnt namespaces with the `unshare` command (which implements the `unshare()` syscall).

For example, let us create a new process in another mnt space and check this.

```
t1# unshare -m /bin/bash
```

The option `-m` is used to unshare the mnt namespace. Now, let's create a new mount point using a filesystem in memory (tmpfs):

```
t1# mount -n -o size=1m -t tmpfs tmpfs  /tmp/mytmpfs
```

Note. The option `-n` is to mount without writing in /etc/mtab
Checking the available mount points.

```
t1# grep /tmp /proc/mounts
tmpfs /tmp/mytmpfs tmpfs rw,relatime,size=1024k 0 0
```

Now, let's create some files:

```
t1# cd /tmp/mytmpfs
t1# touch hello.txt
t1# touch hola.txt
```

Open another terminal now (terminal 2) and execute the following commands:

```
t2# ls -la /tmp/mytmpfs
```

As we will observe, the files hello.txt and hola.txt are not visible because they were written in another mnt namespace inside a tmpfs mounted only in that namespace.

1.2.3 The uts namespace

The uts namespace deals with one little detail: the hostname that will be "seen" by a group of processes. Each uts namespace will hold a different hostname, and changing the hostname will only change it for processes running in the same namespace. We can create uts namespaces with `unshare -u`. We can use the `hostname` command to change the host name (this command implements the `sethostname()` system call).

1.2.4 The user namespace

Using the user namespace, unprivileged users can create a namespace where they can be root and from this namespace they can start other namespaces. The design is based on a 1-1 uid mapping (by ranges) from uids in the container to uids on the host. For instance, uid 0 in the namespace may really be uid 999990 on the host. Users can be pre-allocated their own private ranges to use however they please. For instance each user may get 10,000 uids, with the first user's range starting at 100,000. The uid and gid mappings are exposed and manipulated through /proc/pid/uid_map and /proc/pid/gid_map.

Regarding security, an unprivileged user can only access to his existing privileges in the parent user namespace. For instance, an unprivileged user might create a new filesystem tree and `chroot` into it, but he will not be able to mount over the filesystem on the parent namespace to for example, maliciously set the setuid over /etc/passwd or /bin/bash.

1.2.5 The net namespace & veth interfaces

A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices. Even the loopback interface (`lo`) will be different in each different net namespace. Each net namespace has its own meaning for INADDR_ANY, a.k.a. 0.0.0.0; so when for example our web server process binds to *:80 within its namespace, it will only receive connections directed to the IP addresses and interfaces of its namespace, thus allowing us to run multiple web server instances, with their default configuration listening on port 80.

We can use the `unshare` command to create a network namespace but it is more comfortable to use the `ip` command to work with network namespaces. In the kernel, namespaces do not have names but they are identified by the processes that are running inside these namespaces. However, the `ip` command uses the concept of "named network namespace", which, by convention is an object at /var/run/netns/NAME that can be opened. The file descriptor (fd) resulting from opening /var/run/netns/NAME refers to the specified network namespace and this fd can be used to change the network namespace associated with a process.

For example, to create a network namespace we can use these commands:

```
# ip netns add myns
# ip netns list
myns
# ls /var/run/netns/
myns
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1e:4f:f9:76:ac
          ...
lo        Link encap:Local Loopback
          ...
```

Then, we can start a bash in that namespace and type some commands:

```
# ip netns exec myns bash
# ifconfig
# exit
exit
```

As shown, the previous network namespace has not active network devices. Now, if we want to connect the network namespace to a network, we can send a physical interface to a network namespace and this is a way to enable network communication.

```
# ip link set eth0 netns myns
```

However, if we execute the previous command, the interface is no longer available in the parent namespace. A more typical approach is to use veth interfaces. These interfaces are virtual ethernet interfaces that always exist in pairs. Whatever enters on one interface, exits from the other one, and viceversa. Next, we create and test a pair of veth interfaces:

```
# ip link add veth0 type veth peer name veth1
# ip link show
...
23: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether ee:c0:0e:d6:ae:09 brd ff:ff:ff:ff:ff:ff
24: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 4e:e8:84:bd:01:f0 brd ff:ff:ff:ff:ff:ff
```

In the previous command we created a pair of veth interfaces called `veth0` and `veth1` but these names can be chosen by the user. Now, let's send some traffic to show the interfaces in action:

```
# ip link set veth0 up
# ip link set veth1 up
# ip addr add 10.0.0.1/24 dev veth0
# ping -c 3 10.0.0.2
```

Note. For an interface to be up, the other one must be up too. While the ping is running, we will be able to observe traffic on `veth1` (most likely ARP). Now, we can send one leg of the veth interface to a network namespace:

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns myns
```

The first command sets up a veth interface and the second command assigns `veth1` to myns.

Non-root processes that are assigned to a namespace via `clone()`, `unshare()`, or `setns()` only have access to the networking devices and configuration that have been set up in that namespace. Only root can add new devices and configure them, of course. Using the `ip netns` sub-command, there are two ways to address a network namespace: by its name or by the process ID of a process in that namespace. For example:

```
# ip netns exec myns bash
# ip link add vethMYTEST type veth peer name eth0
# ip link set vethMYTEST netns 1
# exit
```

1.3 cgroups

Control groups, or "cgroups", facilitate the management and administration of resources. They consist of a set of mechanisms to measure and limit resource usage for groups of processes. LXC relies on the Linux kernel cgroups, which are a feature to limit, control and isolate resource usage of process groups (CPU, memory, disk I/O,...).

Conceptually, it works a bit like the `ulimit` shell command or the `setrlimit` system call; but instead of manipulating the resource limits for a single process, they allow to set them for groups of processes. These groups are hierarchical, beginning with the top group which all processes are located in unless set otherwise. We can then define groups, subgroups, sub-subgroups, etc. to facilitate our limitation needs. Note bandwidth cannot be limited by now with cgroups. Cgroups are not dependent upon namespaces. This means that is possible to build cgroups without namespaces kernel support.

1.3.1 Pseudo-FS Interface

The easiest way to manipulate control groups is through the cgroup filesystem. All cgroups actions are performed via filesystem actions. This means creating, removing, reading or writing into directories. And as cgroups uses a Virtual File System, all entries created are deleted after rebooting (are not persistent).

Depending on our Linux distribution we may or may not have cgroups already set up. The easiest way to check is to run the mount and look for lines starting with cgroup. In the event that we do not have cgroup mount points, we need to mount them by adding the following line to our /etc/fstab file:

```
cgroup  /sys/fs/cgroup  cgroup  defaults  0   0
```

Some distributions (like Ubuntu) mount the separate subsystems (cpu, memory, blkio, etc) into separate directories, some just use a flat structure. It does not matter much, we just have to go into subdirectories for the different subdirectories if our system works like that. To manage cgroups the recommended way is to use `libcgroup` as it contains a bunch of utilities for managing cgroups in an easier manner. We can install these utilities with the following command:

```
sudo apt-get install cgroup-bin
```

If we are configuring cgroups manually we need to save values into files within the /sys/fs/cgroups directory. These files are not regular files though, just like /proc they are actually configuration interfaces for the Linux kernel's internal workings. Use the `echo` and `cat` commands to set the desired values.

1.3.2 Creating a cgroup

To create a cgroup simply create a directory in /sys/fs/cgroup or if we have a per-subsystem setup, in the appropriate directory for the subsystem. The kernel automatically fills the cgroup's directory with the settings file nodes. If we want to use the toolkit-way, use `cgcreate` and provide the subsystems we wish to add as a parameter:

```
# cgcreate -g cpu,cpuacct,...:/my_group
```

Example:

```
# cgcreate -g cpuset:/my_group
# cd /sys/fs/cgroup/cpuset/my_group
# ls
cgroup.clone_children  cpuset.memory_pressure
cgroup.event_control   cpuset.memory_spread_page
cgroup.procs           cpuset.memory_spread_slab
cpuset.cpu_exclusive   cpuset.mems
cpuset.cpus            cpuset.sched_load_balance
cpuset.mem_exclusive   cpuset.sched_relax_domain_level
cpuset.memory_migrate  tasks
```

1.3.3 Attaching processes

To attach a process just echo the processes ID into the tasks file of the group. Note, that we can only inject one task at a time:

```
# echo 1234 >/sys/fs/cgroup/my_group/tasks
```

Alternatively we can of course use the cgclassify tool to classify multiple processes:

```
# cgclassify -g cpu,cpuset,...:/my_group 1234 1235 ...
```

1.3.4 Deleting a cgroup

Deleting a cgroup is a little more tricky as we ca not directly use rm -rf since we cannot really remove the files in that directory. Instead just use a little trick, which removes the directory with a depth of one:

```
# find my_group -depth -type d -print -exec rmdir {} \;
```

Again, there is a utility for that:

```
# cgdelete cpu,cpuset,...:/my_group
```

1.3.5 Limiting CPU usage

CPU limits come in two flavors: binding cgroups to certain CPU cores and limiting the actual usage. To set the CPU affinity of a group we have to use the cpuset subsystem. As discussed before, we will use command line tools to adjust the settings. Let's look at our process group's CPU affinity before setting it:

```
# cat cpuset.cpus
0-7
```

As we can see the CPU affinity for this group is set to all 8 cores. (If we have more cores, the affinity will be set accordingly.) To adjust the affinity, we simply echo the desired core list into the file:

```
# echo "0-2,4" >cpuset.cpus
```

The limit is then applied immediately as we can observe using the top command. Be careful though, setting the CPU affinity of the root cgroup will affect all processes. If CPU affinity is not enough, we can also adjust CPU bandwidth. This means, that we set the weight of a group with the process scheduler. This will still give the process all free CPU, but will give other processes a higher priority when considering CPU allowance. Setting this is done via the cpu.shares option, which defaults to 1024.

```
# echo 2048 >cpu.shares
```

The final and most strict setting is the realtime CPU quota a process gets. This only works on realtime scheduling groups and we should not use it unless we know what we are doing.
There are two configuration options, cpu.rt_runtime_us and cpu.rt_period_us. The former limits how long the process can keep the CPU continuously at most, whereas the latter sets the period length for the former setting. In other words if we want a process to access the CPU 4 seconds out of 5, we need to set cpu.rt_runtime_us to 4000000 and cpu.rt_period_us to 5000000. There are a few non-trivial issues in the way however. Setting very small values in either option can result in an unstable system.

1.3.6 Limiting memory usage

Limiting memory usage is easy compared to the CPU. We will basically want to limit how much memory a certain process can use, and that is what memory.limit_in_bytes and memory.memsw.limit_in_bytes are for. memory.limit_in_bytes limits the total memory usage of a group including file cache, whereas memory.memsw.limit_in_bytes limits the amount of memory and swap a group can use. Pay attention however, memory.limit_in_bytes should be set first, otherwise we will receive an error. Also note that we do not need to specify the amount in bytes, we can use the shorthand multipliers k or K for kilobytes, m or M for Megabytes, and g or G for Gigabytes.

1.3.7 Limiting (disk) IO

Last but not least among the major limiting features there is the IO. Disk IO was a long time pain in the neck, since no really reliable methods existed for limiting it. With cgroups however, we have a couple of switches available. Again, we have blkio.weight, which behaves just like the CPU shares and deserves no further explanation.

When weights are not enough, fixed limits come into play. We can either limit by bytes-per-second or by IOPS (IO Operations Per Second). The configuration options are aptly named:

```
blkio.throttle.read_bps_device for read limits in BPS
blkio.throttle.read_iops_device for read limits in IOPS.
blkio.throttle.write_bps_device for write limits in BPS.
blkio.throttle.write_iops_device for write limites in IOPS.
```

To adjust them we need to figure out the minor and major number of the device we wish to put a limit on. Easily done though, just use ls -la /dev and look at the line with our device, the numbers in just before the date will be the two numbers we are looking for. To place a limit run the following with our major, minor and byte limits replaced:

```
# echo "252:2 10485760" > blkio.throttle.write_bps_device
```

1.3.8 Permanent Configuration of cgroups

Now that all limits are configured, we might want to make sure they are applied upon restart as all contents of /sys, cgroup configurations are volatile and a reboot deletes them. To persist the configuration one can use the cgconfigparser utility, which is available in the libcgroup toolkit. The config parser takes a configuration file and builds the whole cgroup structure from scratch and also allows for resource and process assignment.

1.4 LXC Containers

1.4.1 Install

The standard tools for complete Linux containers is LXC. Complete containers are like virtual machines with all its relevant environment isolated and being able to execute multiple processes inside. LXC can be installed as:

```
hypervisor# apt-get install lxc
```

1.4.2 Create

To create a container with LXC we just need to run the create command `lxc-create`.

```
hypervisor# lxc-create -n mycontainer -t ubuntu
```

The `-n` or `--name` option specifies the name of the container (virtual machine) and the `-t` option specifies the template to create the filesystem of the container. A template file is really nothing more than a shell script that runs when create is called. By default, templates reside in /usr/lib/lxc/templates on Ubuntu.

An LXC template is nothing more than a script which builds a container for a particular Linux environment. When we create an LXC container, we need to use one of these templates. The templates download the data to create a system (for Debian-based systems the template uses a utility called `debootstrap`[2]).

By default, it will create a minimal Ubuntu install of the same release version and architecture as the local host. If we want, we can create Ubuntu containers of any arbitrary version by passing the release parameter.

For example, to create a Ubuntu 14.10 container:

```
hypervisor# lxc-create -n <container-name> -t ubuntu -- --release utopic
```

It will download and validate all the packages needed by a target container environment. The whole process can take a couple of minutes or more depending on the type of container. So be patient. After a series of package downloads and validation, an LXC container image are finally created, and we will see a default login credential to use. The container is stored in /var/lib/lxc/<container-name>. Its root filesystem is found in /var/lib/lxc/<container-name>/rootfs.

All the packages downloaded during LXC creation get cached in /var/cache/lxc, so that creating additional containers with the same LXC template will take no time.

1.4.3 SELinux

New linux distros apply some security mechanisms that support control security policies for applications. These are called Security-Enhanced Linux (SELinux). The solution to SELinux in Ubuntu is called App Armor. This causes problems to LXC containers. To overcome this issue we have to add the following line to the container's config file (/var/lib/lxc/mycontainer/config):

```
lxc.aa_profile = unconfined
```

1.4.4 Start/Stop

To start the container type the following command:

```
hypervisor# lxc-start -n mycontainer
```

Once we do this we will be prompted with a login and asking for our username and password. This, by default, is ubuntu for both. In the console of the container, we can shut down it as any other Linux box:

```
mycontainer# halt
```

[2] If we get an error about `lxc-create` or `debootstrap` make sure we installed the lxc package.

We can also shut down the container from the **hypervisor**:

```
hypervisor# lxc-stop -n mycontainer
```

We can ran the start command with the `-d` or `--daemon` option to "daemonize" the container:

```
hypervisor# lxc-start -n mycontainer -d
```

To connect to a container that has been started in the background, we can use the following command:

```
hypervisor# lxc-console -n mycontainer
```

We can exit the console of a container without shutting down it by typing `ctrl+a` then `q`.

Finally, the command `lxc-wait` waits for a specific container state before exiting. This command is very useful in scripting. In this example the command will terminate when the specified container reaches the state RUNNING:

```
hypervisor# lxc-wait -n mycontainer -s 'RUNNING'
```

1.4.5 Destroy

The command `lxc-destroy` completely destroys the container and removes it from our system. The required argument is the container name but we can also pass along -f to tell LXC to force a deletion if the container is currently running (default is to abort/error out). Example:

```
hypervisor# lxc-destroy -n mycontainer
```

1.4.6 Freeze

We can also freeze or pause a container. The command `lxc-freeze` and `lxc-unfreeze` lets us pause the container (stop all processes from running) and lxc-unfreeze will thaw all the processes previously frozen.

1.4.7 Get information

There are several tools that come with the LxC package to get information about containers. For example, with `lxc-ls` we can see the containers that are running, stopped, frozen or active. With the option `--fancy` basic information is displayed:

```
hypervisor# lxc-ls --fancy
hypervisor# lxc-info -n mycontainer
```

The command `lxc-info` is similar to `lxc-ls` but requires the -n <container name> argument. It tells us the state of the container as well as its process ID on the `hypervisor` and information about resources used. This command will also tell ys if the container is frozen.

On the other hand, the command `lxc-checkconfig` checks the current kernel for LxC support. The command `lxc-monitor` monitors the state of containers. By default the output is not saved in a log, but with `--logfile=FILE` parameter we can choose where to save the log.

1.4.8 Run a Command

The command `lxc-attach` let us run a command in a running container. This command is mainly used when we want to quickly launch an application in an isolated environment or create some scripts. Example:

```
hypervisor# lxc-attach -n webserver -- ifconfig eth1 192.168.1.2/24
```

The previous command configures the network of a container called **webserver**.

1.4.9 Limiting Resources

The command `lxc-cgroup` allows us to set inner workings of a specific container (these can also be set in the container's "config" file as well). The one that most people will probably find helpful is setting memory limits, which can be done like this (300M RAM limit):

```
hypervisor# lxc-cgroup -n mycontainer memory.limit_in_bytes 300000000
```

1.4.10 Devices

The command `lxc-device` let us manage devices in running containers. At this point, only the `add` device action is supported. For example, with the following command we can create a device into a container based on the maching device on the **hypervisor**. Like this, an USB device will be available on the container:

```
hypervisor# lxc-device -n mycontainer add /dev/sdb1
```

Or we can move the interface `eth0` from the **hypervisor** to a container as `eth3`:

```
hypervisor# lxc-device -n mycontainer add eth0 eth3
```

1.4.11 Auto-start

By default a new container will not start if the host system is rebooted. If we want a container to start at system boot, we can configure that behavior in each container's configuration file. To do so, add the following lines to config file:

```
lxc.start.auto = 1
lxc.start.delay = 5
```

With these parameters, the container will start when the host server boots, then the host system will wait 5 seconds before starting any other containers. We can also set a value for `lxc.group` parameter to group containers. A container can belong to several groups by using this syntax:

```
lxc.group = group1,group2,group3
```

Then, with groups, we can easily shutdown, reboot, kill or list all the containers that belong to a specific group:

```
hypervisor# lxc-autostart -s -g <group>
```

1.4.12 Container Storage

By default, LXC simply stores the rootfs of containers under the directory /var/lib/lxc/<container>/rootfs (the `--dir` option can be used to override the path). However, apart from a simple directory with a traditional filesystem, LXC supports additional storage backends (also referred to as backingstores). These are overlayfs, btrfs, zfs and lvm. We can specify them in the `lxc-create` command with the `-B` option. For the btrfs and zfs, LXC can autodetect the backingstore and automatically use the underlying features of it.

The overlayfs is mostly used when cloning containers to create a container based on another one and storing any changes in an overlay. Overlay-filesystem (or Unionfs) is a filesystem service that uses union mount to mount different filesystems hierarchies to appear as one unified filesystem. The overlay-filesystem overlays one filesystem above the other into a layered representation.

When a directory appears in both layers, overlayfs forms a merged directory for both of them. In case of two files have the same name in both layers, only one is served from the upper or the lower layer, but if a file only exists in the lower layer and an edit needs to be done on that file, a copy of this file is created on the upper layer to be edited. In

most cases the lower layer is normally a read-only filesystem, while the upper layer is read-write one, to allow what is called copy on write, which allows only the writes to the upper filesystem leaving the base lower unchanged.

When used with lxc-create it will create a container where any change done after its initial creation will be stored in a "delta0" directory next to the container's rootfs. With btrfs, LXC will setup a new subvolume for the container which makes snapshotting much easier.

The zfs filesystem is similar to btrfs.

With lvm, LXC will use a new logical volume for the container:

- The LV can be set with `--lvname` (the default is the container name).
- The VG can be set with `--vgname` (the default is "lxc").
- The filesystem can be set with `--fstype` (the default is "ext4").
- The size can be set with `--fssize` (the default is "1G").
- We can also use LVM thinpools with `--thinpool`.

Clones are either snapshots or copies of another container. A copy is a new container copied from the original, and takes as much space on the host as the original. A snapshot exploits the underlying backing store's snapshotting ability to make a copy-on-write container referencing the first. Snapshots can be created from different technologies such as btrfs, LVM and directory backed containers.

To create copies and snapshots we can use the command `lxc-clone`. The syntax is pretty simple:

```
hypervisor# lxc-clone -o <original container name> -n <new container name>
```

If we want to create a snapshot instead of a copy we have to include the `-s` option in the command:

```
hypervisor# lxc-clone -s -o <original container name> -n <new container name>
```

Snapshots of directory-packed containers are created using the overlay filesystem. For instance, a privileged directory-backed container C1 will have its root filesystem under /var/lib/lxc/C1/rootfs. A snapshot clone of C1 called C2 will be started with C1's rootfs mounted readonly under /var/lib/lxc/C2/delta0. Importantly, in this case C1 should not be allowed to run or be removed while C2 is running. It is advised instead to consider C1 a canonical base container, and to only use its snapshots.

To create snapshots we can also use the command `lxc-snapshot`:

```
hypervisor$ sudo lxc-snapshot -n C1
```

The previous command creates a snapshot-clone called 'snap0' under /var/lib/lxcsnaps or `$HOME/.local/share/lxcsnaps`. The next snapshot will be called 'snap1', etc. Existing snapshots can be listed using `lxc-snapshot -L -n C1`.

A snapshot can be restored (erasing the current C1 container) using:

```
hypervisor$ lxc-snapshot -r snap1 -n C1
```

After the restore command, the snap1 snapshot continues to exist, and the previous C1 is erased and replaced with the snap1 snapshot.

Snapshots are supported for btrfs, lvm, zfs, and overlayfs containers. If lxc-snapshot is called on a directory-backed container, an error will be logged and the snapshot will be created as a copy-clone. The reason for this is that if

the user creates an overlayfs snapshot of a directory-backed container and then makes changes to the directory-backed container, then the original container changes will be partially reflected in the snapshot. If snapshots of a directory backed container C1 are desired, then an overlayfs clone of C1 should be created, C1 should not be touched again, and the overlayfs clone can be edited and snapshotted at will, as such:

```
lxc-clone -s -o C1 -n C2
lxc-start -n C2 -d # make some changes
lxc-stop -n C2
lxc-snapshot -n C2
lxc-start -n C2 # etc
```

While snapshots are useful for longer-term incremental development of images, **ephemeral containers** utilize snapshots for quick, single-use throwaway containers. Given a base container C1, we can start an ephemeral container using:

```
lxc-start-ephemeral -o C1
```

The container begins as a snapshot of C1. Instructions for logging into the container will be printed to the console. After shutdown, the ephemeral container will be destroyed.

1.4.13 Networking

1.4.13.1 Default Upstart Configuration

The default upstart configuration in Ubuntu is in the file: **/etc/init/lxc-net.conf**:

- By default, LxC creates a private network namespace for each container, which includes a layer 2 networking stack.
- LxC also automatically creates a network bridge called `lxcbr0` that is started when the host is started.
- The bridge interface is NATed to enable the containers to access the outside world with a veth endpoint passed into the container. The NAT is created with `iptables` with a rule to basically masquerade all the traffic leaving the containers which are bridged with `lxcbr0`.
- By default, the LXC installation also launches a `dnsmasq` process that acts as DHCP server for the default linux bridge `lxcbr0`. This DHCP server uses IPv4 addresses from 10.0.3.2 to 10.0.3.254.

1.4.13.2 Basic Data Link Layer

Let's consider that we have created a container called mycontainer. Then, the configuration of the container is in the file **/var/lib/lxc/mycontainer/config**. In particular, the main L2 parameters that can be configured are the following:

```
lxc.utsname = mycontainer
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.name = lxcnet0
lxc.network.veth.pair = vethmycontainer
```

The previous configuration is for a container called mycontainer. Also the configuration file specifies that we are going to use a veth interface, connected to a bridge in the host called `lxcbr0`, that the veth interface must be activated (up) and that the hardware address that we will use is 00:16:3e:42:1d:a7. The name of network device inside the container defaults to `eth0` but in this case we set another name: `lxcnet0`. In addition, we can select the interface name in the host. In this case `vethmycontainer` Otherwise, by default LXC selects a random name like `veth3VcOob`. The default configuration for the previous parameters are in the file **/etc/lxc/default.conf**:

```
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = lxcbr0
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
```

Finally, after making any changes we will have to restart the container:

```
lxc-stop -n mycontainer
lxc-start -n mycontainer -d
```

1.4.13.3 Adding More Interfaces

We can add more interfaces to a container by simply adding more L2 parameters:

```
lxc.utsname = mycontainer
# First Interface
lxc.network.type = veth
lxc.network.flags = up
# Second Interface
lxc.network.type = veth
lxc.network.flags = up
```

By default, the second interface is called `eth1` and so on.

1.4.13.4 Other Switches

If we want to configure a switch other than `brctl`, we can use a script to connect the interface to the bridge:

```
lxc.network.type = veth
lxc.network.flags = up
# lxc.network.link = br1 # This has to be disabled.
lxc.network.script.up = /etc/lxc/ovs-br1.sh
lxc.network.name = eth1
```

The previous configuration executes the script /etc/lxc/ovs-br1.sh when the container is started. This script could have a configuration like the following:

```
 #!/bin/bash
BRIDGE="br1"
ovs-vsctl --may-exist add-br $BRIDGE
ovs-vsctl --if-exists del-port $BRIDGE $5
ovs-vsctl --may-exist add-port $BRIDGE $5
# Mode SDN
ovs-vsctl set-fail-mode $BRIDGE secure
```

The previous configuration starts (if not already started) a bridge called `br1` with openvswitch, adds the link to the bridge and finally sets the bridge in SDN mode.

1.4.13.5 IP

As mentioned, by default the container obtains the IP addresses from a DHCP server but we can also configure them:

```
# the ip may be set to 0.0.0.0/24 (not assigned) or
# skip these lines if you like to use a dhcp client inside the container
lxc.network.ipv4 = 1.2.3.5/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

If we are not using a DHCP server it is convenient to set the IP address to 0.0.0.0/24.

1.4.13.6 Types of Interfaces

In LXC for Ubuntu, there are five network virtualization types available: *empty*, *veth*, *macvlan*, *vlan* and *phys*. Additionally, the type *none* is available on more recent version.

The **empty** network type creates only the loopback interface. In fact, empty network type creates EMPTY NETWORK NAMESPACE in which the container runs.

The **veth** network type has been already described. In simplified terms is works as follows:

- A pair of veth devices is created on the host. Future container's network device is then configured via DHCP server (it's actually dnsmasq daemon) which is listening on the IP assigned to the LXC network bridge (in this case that's lxcbr0). We can verify this by running `sudo netstat -ntlp | grep 53`. The bridge's IP address will serve as the container's default gateway as well as its nameserver.
- The host part of the veth device pair is bridged to the configured bridge (by default `lxcbr0`).
- Slave part of the pair is then moved to the container, renamed to eth0 by default and finally configured in the container's network namespace.
- Once the container's init process is started it brings. up particular network device interface in the container and we can start networking.

The **mcvlan** (MAC VLAN) is a way to take a single network interface and create multiple virtual network interfaces with different MAC addresses asigned to them.

1.4.14 Unprivileged containers

Unprivileged containers are run without root access and are safer to run since they run in the context of a non-sudo user. This can come in handy in cases where we must share a server with other users (but do not want to provide sudo access) or when we do not want to provide our containers full access to our underlying system for security reasons.

First, install systemd-services and uidmap:

```
# apt-get install systemd-services uidmap
```

Next, we need to create LXC configuration files for our user. This is how the system LXC config files match to the user config files:

```
/etc/lxc/lxc.conf => ~/.config/lxc/lxc.conf
/etc/lxc/default.conf => ~/.config/lxc/default.conf
/var/lib/lxc => ~/.local/share/lxc
/var/lib/lxcsnaps => ~/.local/share/lxcsnaps
/var/cache/lxc => ~/.cache/lxc
```

At a minimum create the .config/lxc/default.conf, .local/share/lxc to store containers and .cache/lxc for the downloaded container templates cache in our user's home directory.

```
hypervisor~user1$ chmod +x $HOME
hypervisor~user1$ mkdir .config/lxc .local/share/lxc .cache/lxc
```

When a new user is created in Ubuntu 14.04 LTS, it is automatically assigned a range of 65536 user and group ids, which start at 100000 to avoid conflicts with system users. If we look at /etc/subuid and /etc/subgid, we will see something like:

```
user1:100000:65536
user2:165536:65536
```

This indicates that user1 has user and group ids that go from 100000 to 165535 and user2 has user and group ids that go from 165536 to 231072. If we need it, the commands to allocate additional uids and gids to our username are the following:

```
# usermod --add-subuids 100000-165536 user1
# usermod --add-subgids 100000-165536 user1
```

To provide container isolation and avoid potential conflicts, the user and group ids in each unprivileged container (which will start at 0) will then be mapped to the user's range of allocated ids on the system. Now in .config/lxc create a default.conf file with the config below.

```
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:xx:xx:xx
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

Note that each user must specify its own allocated range of user and group ids to map to. To setup unprivileged containers the first step is to allow the user to hook into the container network through the bridge lxcbr0. To do so, we need to create and configure the file /etc/lxc/lxc-usernet for unprivileged containers. A configuration line in this file can be the following:

```
user1 veth lxcbr0 5
```

In this example, we let user1 be able to create up to 5 veth interfaces in the bridge `lxcbr0`.

Finally, to create unprivileged containers we need to use the 'download' template type. These container OS templates are designed to support unprivileged containers. Its important to clarify at this point containers created without the download template type will not work in unprivileged mode as those container OS templates are not designed to. So let's create our first unprivileged container:

```
hypervisor~user1$ lxc-create -t download -n mycontainer1 -- -d ubuntu -r trusty -a amd64
```

This should create the mycontainer1 container in our user's .local/share/lxc folder. Now start the container:

```
hypervisor~user1$ lxc-start -n mycontainer1 -d
```

This should start the container and we can use `lxc-console` to enter the container.

1.5 Docker

Figure 1.2: Docker logo.

1.5.1 Definition

Docker is an open-source project that automates the deployment of applications in containers. Is a ontainer-based software framework for automating deployment of applications. It is a very important tool for building software that run in the backend.

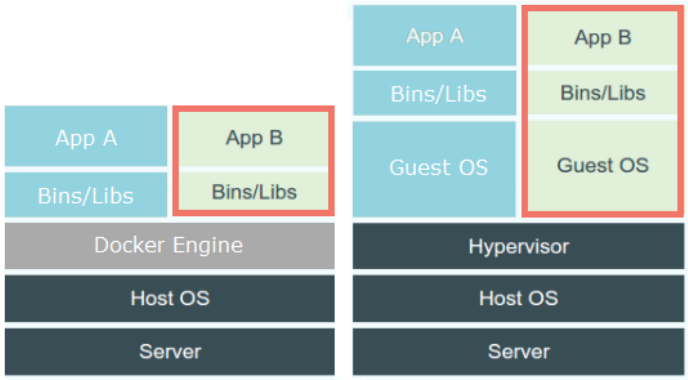


Figure 1.3: Docker vs VM

With Docker we can easily create lightweight, portable and self-sufficient containers from any application with better performance than Virtual Machines. Docker also allows us to deploy and scale more easily, because Docker containers can run almost everywhere: on desktops, physical servers, virtual machines, into data centers, in public or private clouds.

Docker uses a client-server architecture where docker client and daemon communicate via sockets or through a RESTful API. The daemon operates and distributes Docker containers, so does the heavy lifting of building. The Docker client is the primary user interface to Docker, it accepts commands from the user and communicates with the Docker daemon.

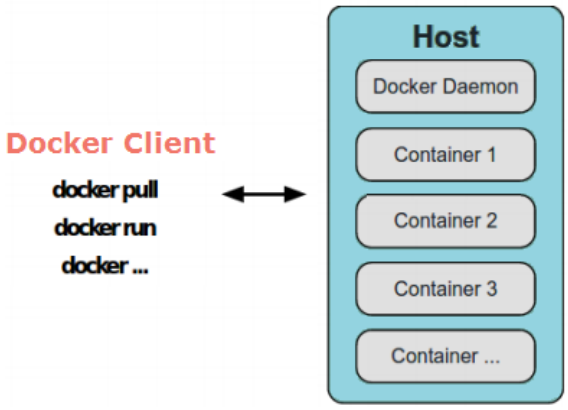


Figure 1.4: Docker architecture

1.5.2 Docker Components

A Docker project is formed by the following elements:

1.5.2.1 Docker Images

Docker images are ready-only templates from which Docker containers are launched. They are very similar to efault operating-system disk images which are used to run applications on servers or desktop computers. Docker provides a

simple way to build new images or update existing images, or we can download Docker images that other people have already created. Docker images are the build component of Docker.

1.5.2.2 Docker Registries

Docker registries are public or private stores from which we upload or download images. Docker Hub is the public Docker registry, and it provides a huge collection of existing images. These images can be created by the user or can be images that others have previously created. Docker registries are the distribution component of Docker.

1.5.2.3 Dockerfiles

Dockerfiles are scripts containing a successive series of instructions, directions, and commands which must be executed to form a new docker image. The Dockerfiles replace the process of doing everything manually and repeatedly.

1.5.2.4 Docker Containers

Docker containers are similar to a directory, which can be packed like any other, then can be shared and run across various different machines and platforms (hosts). A Docker container holds everything that is needed for an application to run. Each container is created from a Docker image and is an isolated and secure application platform. Docker containers are the run component of Docker.

1.5.3 Installing Docker on Ubuntu

Ubuntu Trusty (14.04 LTS 64-bit) comes with a 3.13.0 Linux kernel, and a docker.io package which installs all its prerequisites from Ubuntu's repository.

Docker can be installed by executing these commands on any Linux system that supports LXC:

First, we must ensure the list of available packages is up to date before installing anything new:

```
$ sudo apt-get update
```

Next, Install the docker-io package:

```
$ sudo apt-get install docker.io
```

Link and fix paths with the following two commands:

```
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
$ sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/docker.io
```

Finally, if we want Docker to start when the server boots:

```
$ sudo update-rc.d docker.io defaults
```

Typing `docker` we can check if Docker was successfully installed.

Docker, like many Linux tools, does not have a graphical user interface. And it is only supported by linux platforms, but it can also be installed in OSX or Windows by doing some additional steps.

1.5.4 Creating a Docker container

For creating a container with Docker we should first create a file called `Dockerfile`, which will contain the instructions for the `build command`.

Define the operating system:

```
FROM ubuntu
```

Then, we run the build command. Wee need to specify a name with the `-t` parameter and the location of the Dockerfile, which can be a dot if we are already in the correct folder.

```
$ sudo docker build -t <container name>
```

Once the container is created we can execute a process inside the container:

```
$ sudo docker run <container name> echo "Hello world!"
```

Another example:

```
$ sudo docker run <container name> ping google.com
```

To execute commands inside a container in background:

```
$ sudo docker run -d test ping google.com
```

This Dockerfile example setup a SSH daemon service in a container which can be used to remotely connect to it:

```
FROM ubuntu:latest

RUN apt-get update

RUN apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd

#Defining port for SSH
EXPOSE 22

CMD    ["/usr/sbin/sshd", "-D"]
```

1.5.5 Creating a Docker container from an image

We also can create a new Docker container, from one of the images available from Docker, make some changes and save it locally for future use.

First, we must see if we have any images in our local Docker library:

```
$ sudo docker images
```

If there si any image installed, the previous command lists the available local images which we can use to create a Docker container. If there is no istalled image available, we must download one from the central Docker repository. We must choose which image to download and use to create our first Docker container. There are literally thousands of images available on the central repository and all can be downloaded through the docker command.

For example, to look for ubuntu image, and download it:

```
$ sudo docker search ubuntu
```

The above command will display a list of all images available containing the word ubuntu.
To download the basic ubuntu 14.04 image:

```
$ sudo docker pull ubuntu:14.04
```

Now, if we try againg the initial command, we can see that we have an installed image:

```
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              14.04               2103b00b3fdf        2 days ago          188.3 MB
```

The next step is to create a container and make the required changes before saving it as a new image for use in the future. Creating a container in Docker is done with the run command followed by a command to run within the container. Now we will create the Docker container with the run commnad and specify the bash shell to be executed on completion. Then, we can use the bash session to customize the image.

```
$ sudo docker run -i -t 2103b00b3fdf  /bin/bash
root@32103b0b3f78:/#
```

Now we have an active shell running on the container which we can be used to install an application, or define configuration such as LDAP SSH login. Let's keep it simple and just run an upgrade with apt-get and then exit.

```
 root@32103b0b3f78:/# apt-get update
 root@32103b0b3f78:/# apt-get upgrade
 root@32103b0b3f78:/# exit
```

Once made the appropriate changes, we proceed to save the save the container as a new image which can be used to create future Docker containers. We'll need to specify the container ID as well as the name of the image to use. We can use a new image name or overwrite the existing image name.

```
$ sudo docker commit 2103b00b3fdf ubuntu:14.04
```

With these steps we have created a new Docker container, from one of the images available from the central Docker repository, made some changes and saved it locally for future use.

1.5.6 Docker basic commands

The following commands are only a small sample of what Docker allow to do.

We can see all available commands with the Docker client by running the docker binary without any options:

```
$ sudo docker
```

With Docker we can also get a list of all the running containers (loaded in memory) with:

```
$ sudo docker ps -l
```

We can attach to a running process by passing the first 3 letters of the `CONTAINER ID`:

```
$ sudo docker attach <first 3 letters of CONTAINER_ID>
```

To inspect the configuration file of a docker container we can run:

```
$ sudo docker inspect test
```

We can kill a process executing in background with the `kill` option:

```
4 sudo docker kill <first 3 letters of CONTAINER_ID>
```

Or we can stop the process:

```
$ sudo docker stop <first 3 letters of CONTAINER_ID>
```

Killing is different from stopping a process running inside a container. With `docker stop` the main process inside the container will receive a SIGTERM signal, and after a grace period, a SIGKILL signal. And with `docker kill` the main process inside the container will be sent SIGKILL, or any signal specified with option –signal.

Then, to restart a container:

```
$ sudo docker restart <first 3 letters of CONTAINER_ID>
```

To run containers in interactive mode (inside the container):

```
$ sudo docker run -i -t <container name>
```

To exit from interactive mode just type: `Ctrl-p + Ctrl-q`.

Changes made inside a Docker container filesystem are not persistent. So, if we shutdown or restart a container we will lose it. This is because containers in Docker are destroyed and created again. In order to make changes permanent we have to commit it:

```
$ sudo docker commit <first 3 letters of CONTAINER_ID> <container name>
```

Quitting a container and completely unload it from memory can be done with this command:

```
$ sudo docker rm <first 3 letters of CONTAINER_ID>
```

The Docker Software is the responsible for containing and managing the Docker containers. And the Docker Client is responsible for providing instructions to the Docker Software. Since now they were running on the same computer, but if they are running on different computers then we need to connect them together. With this command we pass the IP address and port number of the Daemon process:

```
$ sudo docker run -h <IP address> -p <port number> <container name> echo "Hello World!"
```

1.5.7 DockerHub

DockerHub is a cloud platform that manages the lifecycle of distributed apps with cloud services for building and sharing containers and automating workflows. With DockerHub we can upload our containers. The idea behind DockerHub is similar to GitHub project, but with containers.

So, we can pack applications inside the containers and then upload the container. Then, if we make our container public or we share it, people can download it and run these applications inside an stable and working environment.

In order to upload and download a container we use the `push` and `pull` commands. Once we have been logged in our DockerHub account we can even search containers shared in DockerHub from the terminal using the `search` command.

1.5.8 Running a Web Application

With this example we will build our web application running a Python Flask application. First, we will start with the docker run command:

```
$ sudo docker run -d -P training/webapp python app.py
```

With the -d flag we tell Docker to run the container in the background. The -P flag tells Docker to map any required network ports inside our container to our host, so we can view our web application. Then, with training/webapp we've specified a pre-build image that contains a simple Python Flask web application. Finally, with python app.py we've

specified a command for our container to launch our web application.

To see our running container we can use next command:

```
$ sudo docker ps -l
CONTAINER ID  IMAGE                    COMMAND        CREATED         STATUS
3103a10b3tfc  training/webapp:latest  python app.py  40 seconds ago  Up 30 seconds
 PORTS                          NAMES
 0.0.0.0:49155->5000/tcp        nostalgic_morse
```

We can see that Docker has mapped port 5000 (the default Python Flask port) inside the container, to port 49155, on the local Docker host.

Now, we can use a web browser to see our application on port 49155:

Figure 1.5: Docker Web Application.

We can also view our web application's logs using next command:

```
$ sudo docker logs -f nostalgic_morse
* Running on http://0.0.0.0:5000/
10.0.2.2 - - [13/March/2014 19:10:31] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [13/March/2014 19:10:31] "GET /favicon.ico HTTP/1.1" 404 -
```

With flag -f we force docker logs command to act like the tail -f command and watch the container's standard out. Here we can see the logs from Flask showing the application running on port 5000 and the access log entries for it.

We can stop our Web Application Container using:

```
$ sudo docker stop nostalgic_morse
nostalgic_morse
```

And with next command we can check if the container has been stopped:

```
$ sudo docker ps -l
```

Chapter 2

Storage Technologies

2.1 Introduction

Computer technology evolves rapidly, and this has contributed to the obligatory need to develop devices that can respond to the demand for more storage space, portability and durability, as well as improving the various systems that manage our data. Starting with punch cards, and going through the tapes, magnetic disks (floppy disks) and the various current hard disks, the evolution has always pursued a larger storage capacity and larger access speeds. This evolution has led us from counting our capacity in bits, to have a current technology with almost unlimited space available.

Technologies explained in this document provide control and monitoring tools, and visualization techniques for all types of users, but they are more oriented to make life easier for professional administrator rather than people responsible only of their home network. These technologies are some of the most commonly used methods (**LVM**, **RAID**, **BTRFS**) so as to provide flexibility and robustness in all storage systems, which help achieve systems with higher performance and better protected systems.

We also explain the different existing methods or architectures to store data over the network (**DAS**, **NAS**, **SAN**), the advantages or disadvantages that they provide and the storage system technologies that they are based on (File Storage, Block Storage).

Finally, we describe evolution tendency of storage systems technologies, explainig some new concepts like Object Storage and Hybrid Storage, and showing an example of some of the new Open Source distributed storage and File System solutions, like Ceph platform.

But before proceding to provide a deeper introduction to these different storage technologies, we make a brief description of some related concepts and technologies, like filesystem, the existing types, and the different techniques that they use.

2.1.1 Filesystem

A filesystem (or file system) is a hierarchical way that an operating system structures data to store or reatrive it in storage devices. It separates data in individual groups, giving them different names, so that an individual file can be located by describing the path to that file. Information that might describe a file and its contents, such as its owner, who can access the file, and its size, etc. are conveniently stored as metadata in a file system.

There are many types of filesystems and the main differences between them are the OS, access speeds, bearing size, CPU usage, and the different functions or techniques to implement. In Windows systems, some of the most common are FAT32, exFAT, NTFS. In Linux, the most used file systems are ext4, F2FS, XFS and, recently, Btrfs.

2.1.2 Block Devices

Data structures like cylinders, tracks, and sectors, used by traditional disk drives, are not used anymore by the modern systems and subsystems that use disk drives. Cylinder, track, and sector addresses have been replaced by a method called logical block addressing (LBA), which makes disks much easier to work with by presenting a single flat address space. With logical block addressing, the disk drive controller maintains the complete mapping of the location of all tracks, sectors, and blocks in the disk drive. There is no way for an external entity like an operating system or subsystem controller to know which sector its data is being placed in by the disk drive. As there are always going to be bad sectors on any disk drive manufactured, disk manufacturers compensate this by reserving spare sectors for remapping other sectors that go bad.

In the disk, files are stored in blocks. A block is a chunk of data, and when appropriate blocks are combined, it creates a file. A block has an address, and the application retrieves a block by making a SCSI (or ATA) call to that address. It is a very microscopic way of controlling storage. How the blocks are combined or accessed is left up to the application. There is no storage-side metadata associated with the block, except for the address, but is not metadata about the block. In other words, the block is simply a chunk of data that has no description, no association and no owner. A block only takes on meaning when the application controlling it combines it with other blocks.

Allowing this hight level of control to the applications, under the right circumstances, can ensure the extraction of the best performance from a given storage array. This is the reason why block storage has been the best option for performance-centric applications, mostly transactional and database-oriented.

2.1.3 Journaling and Barriers

Updating a filesystem usually requires several separate operations. This makes possible to leave data structures in an invalid intermediate state if there is a system crash between these operations. For example, deleting a file involves two steps: (1) removing its directory entry and (2) marking space for the file and its inode as free. If a crash occurs between steps (1) and (2), there will be an orphaned inode and hence a storage leak. On the other hand, if only step 2 is performed first before the crash, the not-yet-deleted file will be marked free and possibly be overwritten by something else. Detecting and recovering from such inconsistencies normally requires a complete walk of its data structures, for example by a tool such as `fsck` (the file system checker).

Journaling filesystems are designed to try to avoid disk corruption resulting from system crashes. Journaling is a feature that keeps track of the changes that will be made before committing them to the main file system. These changes are first written to the journal without changing the rest of the filesystem. Once all of those changes have been journaled, a "commit record" is added to the journal to indicate that everything else there is valid. Only after the journal transaction has been committed in this fashion can the kernel do the writes in the filesystem. If there is a system crash in the middle, the information needed to safely finish the job can be found in the journal. For this reason, such file systems are quicker to bring back online and less likely to become corrupted than filesystems without journaling. In general, it is not even necessary to run a filesystem integrity checker after a crash.

However, journaling filesystems are exchanging integrity for performance. In first place, since we have to write first the information in the journal and then in the filesystem, this has an impact on the performance of the disk I/O. As a result, most journaling filesystems only apply the journaling to the metadata. This ensures that there will be no filesystem corruption caused by a partial metadata update.

In second place, the filesystem code must, before writing the commit record, be absolutely sure that all of the transaction's information has made it to the journal. For this purpose, just doing the writes in the proper order is insufficient; current disk drives maintain large internal caches and might reorder operations for better performance. So the filesystem must explicitly instruct the disk to get all of the journal data onto the media before writing the commit record. If the commit record gets written first, the journal may be corrupted. The kernel's block I/O subsystem makes this capability available through the use of **barriers**. In essence, a barrier forbids the writing of any blocks after the

barrier until all blocks written before the barrier are committed to the media. By using barriers, filesystems can make sure that their on-disk structures remain consistent at all times. The problem is that barriers also impact severely the performance. For this reason, in many filesystems, barriers are disabled by default because they have a serious impact on performance.

In practice, it turns out that the journal in many filesystems is normally contiguous on the physical media and this helps to prevent reordering. In normal usage, the commit record will land on the block just after the rest of the journal data, so there is no reason for the drive to reorder things. The commit record will naturally be written just after all of the other journal log data has made it to the media.

Finally, if we want to enable barriers we can use `barrier=1` as an option to the `mount` command, either on the command line or in /etc/fstab:

```
# mount -t ext3 -o barrier=1 <device> <mount point>
```

2.1.4 Disk Flushes

When local block devices have write caching enabled, writes to these devices are considered completed as soon as they have reached the volatile cache. If a power outage occurs, the last writes are never committed to the disk, potentially causing data loss. To counteract this, we can use disk flushes. A disk flush is a write operation that completes only when the associated data has effectively been written to disk, rather than to the cache. Disk controllers with battery-backed write cache, are able to flush all pending writes out to disk from the battery-backed cache, ensuring that all writes committed to the volatile cache are actually transferred to stable storage. In this case, we can safely disable disk flushes, which improves write performance.

2.1.5 Partitions

Partitions are containers with a type of filesystem, that we need to create when we start to work with a blank drive. We must first create at least one container with a file system.

We can have one partition that contains all the storage space on the drive or divide the space into twenty different partitions. Either way, we need at least one partition on the drive. After creating a partition, the partition is formatted with a file system. There are several good reasons why we might partition our disk device in several partitions such as to improve performance, to install more than one Operating System, to share data among Operating Systems, to install different file systems, to make computer maintenance tasks quicker (e.g. refragmentation) or to limit the space of a disk area. In Linux, we can use `fdisk` or `gparted` to create partitions.

2.2 LVM

2.2.1 Introduction

Logical Volume Manager (LVM) allows for a layer of abstraction between the operating system and the disks/partitions it uses. In traditional disk management, the operating system looks for what disks are available (/dev/sda, /dev/sdb, etc.) and then looks at what partitions are available on those disks (/dev/sda1, /dev/sda2, etc.). With LVM, disks and partitions can be abstracted to contain multiple disks and partitions into one device. We can create logical partitions that can span across one or more physical hard drives. First, the hard drives are divided into physical volumes, then those physical volumes are combined together to create the volume group and finally the logical volumes are created from volume group.

A **physical volume** (PV) is a block device. It can be a hard drive, a partition, an SD card, a floppy, a RAID device, an encryption device, even a logical volume (LV) can be used as PV. To simplify, we can say that a PV is a storage

source, a device that provides space.

To use the storage of a PV, this must belong to a **volume group** (VG). We can say that a VG is a kind of "virtual hard disk" that can be composed of one or more PVs, and grows by simply adding more PVs. Unlike a real disk, a VG can grow over time if we give him more PVs. On a machine with a single disc we can create a VG that is composed of a single PV and, if over time we ran out of space in the VG, we can add more PVs to the VG and the rest is transparent to file systems, processes or users.

Logical volumes (LV) are the devices that we finally use with LVM to create filesystems, swap disks for virtual machines, etc ... We can say that LVs are like partitions and the final product of LVM, with which we will work really. Unlike traditional partitions, LVs can grow (while space available in VG) regardless of the position they have, even expanding for different PVs.

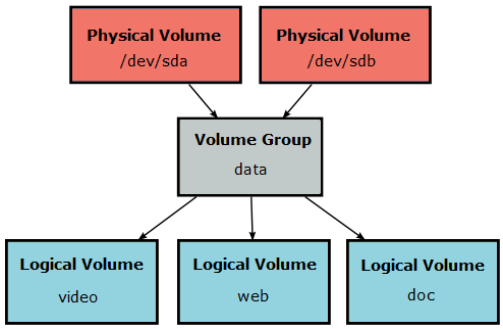


Figure 2.1: LVM scheme.

Because volume groups and logical volumes are not physically tied to a hard drive, it makes it easy to dynamically resize and create new disks and partitions. In addition, LVM can give us features that our file system is not capable of doing. For example, the Ext3 filesystem does not have support for live snapshots, but if we are using LVM, we have the ability to take a snapshot of our logical volumes without unmounting the disk.

Before we start, we have to install the lvm2 package:

```
$ sudo apt-get install lvm2
```

To create a LVM, we need to run through the following steps:

1. Select the physical storage devices for LVM
2. Create the Volume Group from Physical Volumes
3. Create Logical Volumes from Volume Group

2.2.2 Select Physical Storage Devices

To select and test the configuration of the Physical Storage Devices for LVM we use the `pvcreate`, `pvscan` and `pvdisplay` commands.

In this step, we need to choose the physical volumes that will be used to create the LVM. We can create the physical volumes using pvcreate command as shown below.

```
$ pvcreate /dev/sda6
Physical volume "/dev/sda6" successfully created
$ pvcreate /dev/sda7
Physical volume "/dev/sda7" successfully created
```

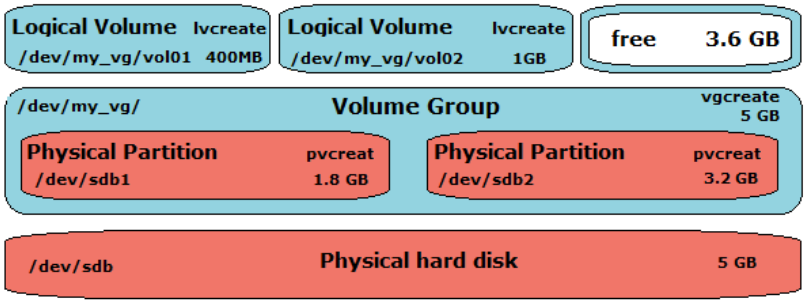


Figure 2.2: Example LVM.

If the physical volumes are already created, we can view them using the pvscan command as shown below.

```
$ pvscan
  PV /dev/sda6                      lvm2 [1.86 GB]
  PV /dev/sda7                      lvm2 [1.86 GB]
  Total: 2 [3.72 GB] / in use: 0 [0   ] / in no VG: 2 [3.72 GB]
```

We can view the list of physical volumes with attributes like size, physical extent size, total physical extent size, the free space, etc. with `pvdisplay`:

```
$ pvdisplay
--- Physical volume ---
  PV Name               /dev/sda6
  VG Name
  PV Size               1.86 GB / not usable 2.12 MB
  Allocatable           yes
  PE Size (KByte)    4096
  Total PE              476
  Free PE               456
  Allocated PE          20
  PV UUID               m67TXf-EY6w-6LuX-NNB6-kU4L-wnk8-NjjZfv

  --- Physical volume ---
  PV Name               /dev/sda7
  VG Name
  PV Size               1.86 GB / not usable 2.12 MB
  Allocatable           yes
  PE Size (KByte)    4096
  Total PE              476
  Free PE               476
  Allocated PE          0
  PV UUID               b031x0-6rej-BcBu-bE2C-eCXG-jObu-0Boo0x
```

PE (Physical Extents) are nothing but equal-sized chunks. The default size of extent is 4MB.

2.2.3 Create a Volume Group

To create a Volume Group (VG) we have to use the `vgcreate` and `vgdisplay` commands. Volume groups are nothing but a pool of storage that consists of one or more physical volumes. Once we create the physical volume, we can create the volume group (VG) from these physical volumes (PV). In this example, the volume group vol_grp1 is created from the two physical volumes as shown below.

```
$ vgcreate vol_grp1 /dev/sda6 /dev/sda7
  Volume  group "vol_grp1" successfully created
```

LVM processes the storage in terms of extents. We can also change the extent size (from the default size 4MB) using -s flag.

The `vgdisplay` command lists the created volume groups:

```
$ vgdisplay
  --- Volume group ---
  VG Name                vol_grp1
  System ID
  Format                   lvm2
  Metadata Areas           2
  Metadata Sequence No  1
  VG Access                read/write
  VG Status                resizable
  MAX LV                   0
  Cur LV                    0
  Open LV                  0
  Max PV                   0
  Cur PV                    2
  Act PV                   2
  VG Size                  3.72 GB
  PE Size                  4.00 MB
  Total PE                 952
  Alloc PE / Size          0 / 0
  Free  PE / Size          952 / 3.72 GB
  VG UUID                  Kk1ufB-rT15-bSWe-5270-KDfZ-shUX-FUYBvR
```

We can view the short list of volume groups created with:

```
# vgs
```

2.2.4 Create Logical Volumes

To create a Logical Volume (LV) we have to use `lvcreate`, `lvdisplay` command.

Now, everything is ready to create the logical volumes from the volume groups. `lvcreate` command creates the logical volume with the size of 80MB.

```
$ lvcreate -l 20 -n logical_vol1 vol_grp1
  Logical volume "logical_vol1" created
```

Or using the option -L we can use megas or gigas:

```
$ lvcreate -L 400 -n vol01 mynew_vg
$ lvcreate -L 1G -n vol01 mynew_vg
```

To view the available logical volumes with its attributes, we can use the `lvdisplay` command:

```
$ lvdisplay
  --- Logical volume ---
  LV Name                /dev/vol_grp1/logical_vol1
  VG Name                vol_grp1
  LV UUID                 ap8sZ2-WqE1-6401-Kupm-DbnO-2P7g-x1HwtQ
  LV Write Access      read/write
  LV Status              available
  # open                  0
  LV Size                  80.00 MB
  Current LE             20
  Segments               1
  Allocation               inherit
  Read ahead sectors  auto
  - currently set to     256
```

```
  Block device            252:0
```

After creating the appropriate filesystem on the logical volumes, it becomes ready to use for the storage purpose.

```
$ mkfs.ext3 -m 0 /dev/vol_grp1/logical_vol1
```

The `-m` option specifies the percentage reserved for the super-user, set this to 0 if we wish not to waste any space, the default is 5%.

We can view the short list of logical volumes created with:

```
# lvs
```

2.2.5 Resizes

We can extend the size of the logical volumes after creating it by using the `lvextend` command:

```
$ lvextend -L100 /dev/vol_grp1/logical_vol1
  Extending logical volume logical_vol1 to 100.00 MB
  Logical volume logical_vol1 successfully resized
```

The previous command changes the size of the logical volume from 80MB to 100MB. We can also add additional size to a specific logical volume as shown below.

```
$ lvextend -L+100 /dev/vol_grp1/logical_vol1
  Extending logical volume logical_vol1 to 200.00 MB
  Logical volume logical_vol1 successfully resized
```

Imagine that now we have a new hard disk (sdc) and we make a single partition sdc1. We could generate a different mount point and would take advantage of the space, but for something we have done the previous logical volume.

With the `vgextend` command we can add partitions to an already created VG:

```
# pvcreate /dev/sdc1
# vgextend logical_vol1 /dev/sdc1
```

Now we can extend the logical volume in 1GB:

```
$ lvextend -L+1G /dev/vol_grp1/logical_vol1
```

The last step is to indicate the filesystem that more space is available:

```
# xfs_growfs /mount_point
```

2.2.6 Remove logical volume

To remove a LV:

```
# lvremove /dev/mynew_vg/vol02
```

To remove a volume group (VG):

```
# vgremove /dev/mynew_vg
```

2.2.7 Snapshots

LVM snapshots allow the administrator to create a new block device which presents an exact copy of a logical volume, frozen at some point in time. Typically this would be used when some batch processing, a backup for instance, needs to be performed on the logical volume, but we do not want to halt a live system that is changing the data. When the snapshot device has been finished with the system administrator can just remove the device.

Snapshots only consume the space when changes are made to the source logical volume to snapshot volume. In case the snapshot runs out of storage, we can use the command `lvextend` to grow and if we need to shrink the snapshot we can use `lvreduce`. Let's illustrate snapshots with an example. First, check for free space in volume group to create a new snapshot:

```
# vgs
# lvs
```

Let's create a snapshot for one of my volume named `lv_mylv`. For demonstration purpose, we are going to create only 1GB snapshot volume using following commands.

```
# lvcreate -L 1GB -s -n lv_mylv_snap /dev/vg_myvg/lv_mylv
```

Or we can use the long format:

```
# lvcreate --size 1G --snapshot --name lv_mylv_snap /dev/vg_myvg/lv_mylv
```

Where `-s` (`--snapshot`) creates the snapshot and `-n` (`--name`) is the name for the snapshot.
If we want to remove a snapshot:

```
# lvremove /dev/vg_myvg/lv_mylv_snap
```

To show the data about our snapshot:

```
# lvdisplay vg_myvg/lv_mylv_snap
```

One of the informations is the COW-table size. COW means Copy on Write (whatever changes was made to the `lv_mylv` volume will be written to this snapshot). If we copy more than 1GB of files in `lv_mylv`, we will get an error message saying "Input/output error". This means out of space in snapshot. If the logical volume become full it will get dropped automatically and we cannot use it any more, even if we extend the size of snapshot volume.

A solution is to have the same size of source while creating a snapshot, `lv_mylv` size was 10G, if I create a snapshot size of 10GB it will never over flow like above because it has enough space to take snap of our volume.

Another solution is to extend the snapshot when it is close to be full. For this purpose, we can use the following command:

```
# lvextend -L +1G /dev/vg_myvg/lv_mylv_snap
```

Finally, another solution is to use a feature of LVMs that allows to extend the snapshots automatically. To do so, we have to modify the /etc/lvm/lvm.conf file. In particular the following two parameters:

```
snapshot_autoextend_threshold = 75
snapshot_autoextend_percent = 20
```

With these values, if the snapshot volume reach 75%, it will automatically expand the size of snap volume by 20% more. This will save snapshot from overflow drop.

Another feature of snapshots is that they can be restored (useful if something went wrong in the source volume). To restore (or merge) a snapshot, we need to unmount the corresponding filesystem first:

```
# unmount /mnt/lv_mylv/
```

Then, to restore the snapshot:

```
# lvconvert --merge /dev/vg_myvg/lv_mylv_snap
```

After the previous command, the snapshot volume is removed automatically.

2.2.8 Thin Provisioning

Thin Provisioning is a new way of allocating and distributing disk storage resources. Involves using virtualization technology to give the apprerance of having more physical resources that are actually available. Instead of reserving a portion of the disk storage space for each of the users that are connected, a device with "thin provisioning" defines a thin pool inside one of the large volume groups, and define the thin volumes inside that thin pool. So, every user has the space that requires, but we do not reserve anything to anybody, we deliver capacity as each user requires.

By means of “thin provisioning", a common repository of information is defined and for each application will be defined the volumes and size that each application demands, but the actual physical space will be delivered as the record of information progresses. Thus prevents the waste of unused spaces.

The benefits of this technology are multiple, but the most relevant are two. On one side, it could acquire between 13% and 30% less storage, which generates a saving in economic investment. On the other side, and most important, is the recovery of full control of storage by the storage administrators. Through traditional provisioning, the capacity planning are held by the owners of the applications, but with thin provisioning, storage system administrators are those who define storage requirements, using monitoring tools and managing the free space of each application as a common free space.

Setup Thin Pool an Volumes

In this example we will show how to setup a thin pool and thin volumes in a Volume Group with 15GB.

First we create the Volume Group using:

```
# vgcreate vg_thin /dev/sdb1
```

Next, before creating the thin pool and thin volumes, we can check the available size of Logical Volume:

```
# vgs
  VG           #PV  #LV  #SN  Attr     VSize   VFree
  vg_thin       1    0    0   wz--n-   15.28g  15.28g
```

Then, to create a thin pool of 15GB, with mo_tp_pool name in our Volume Group, we use:

```
# lvcreate -L 15G --thinpool mo_tp_pool vg_thin
  Logical volume "lvol0" created
  Logical volume "mo_tp_pool" created
```

Next, we want to create three virtual thin volumes of 5GB each in our thin pool:

```
# lvcreate -V 5G --thin -n thin_vol_usr1 vg_thin/mo_tp_pool
  Logical volume "thin_vol_usr1" created
# lvcreate -V 5G --thin -n thin_vol_usr2 vg_thin/mo_tp_pool
  Logical volume "thin_vol_usr2" created
# lvcreate -V 5G --thin -n thin_vol_usr3 vg_thin/mo_tp_pool
  Logical volume "thin_vol_usr3" created
```

The next step is to create the File System, but first we must create mount points for each thin volume:

```
# mkdir -p /mnt/usr1 /mnt/usr2 /mnt/usr3
```

After copying some files in the created directories, we create the File System for the created thin volumes:

```
# mkfs.ext4 /dev/vg_thin/thin_vol_usr1 && mkfs.ext4 /dev/vg_thin/thin_vol_usr2 && mkfs.ext4
/dev/vg_thin/thin_vol_usr3
```

Now, we mount all three users volumes to the created mount points:

```
# mount /dev/vg_thin/thin_vol_usr1 /mnt/usr1/ && mount /dev/vg_thin/thin_vol_usr2 /mnt/usr2/ &&
mount /dev/vg_thin/thin_vol_usr3 /mnt/usr3/
```

To list the mounted points:

```
# df -h
  Filesystem                          Size    Used   Avail  Use%  Mounted on
  /dev/mapper/vg_mo-LogVol01           34G    2.2G     31G    7%  /
  tmpfs                               938M       0    939M    0%  /dev/shm
  /dev/vda1                           485M     39M    421M    9%  /boot
  /dev/mapper/vg_thin-thin_vol_usr1   5.0G    138M    4.6G    3%  /mnt/usr1
  /dev/mapper/vg_thin-thin_vol_usr2   5.0G    138M    4.6G    3%  /mnt/usr2
  /dev/mapper/vg_thin-thin_vol_usr3   5.0G    138M    4.6G    3%  /mnt/usr3
```

So as to see the real potential of thin provisioning technology, we can fill up some space in our user thin volumes, and then check the space available.

```
# df -h
  Filesystem                          Size    Used   Avail  Use%  Mounted on
  /dev/mapper/vg_mo-LogVol01           34G    2.2G     31G    7%  /
  tmpfs                               938M       0    939M    0%  /dev/shm
  /dev/vda1                           485M     39M    421M    9%  /boot
  /dev/mapper/vg_thin-thin_vol_usr1   5.0G    596M    4.1G   13%  /mnt/usr1
  /dev/mapper/vg_thin-thin_vol_usr2   5.0G    1.4G    3.4G   29%  /mnt/usr2
  /dev/mapper/vg_thin-thin_vol_usr3   5.0G    2.3G    2.4G   49%  /mnt/usr3
```

To see the space available in the thin pool:

```
# lvdisplay vg_thin/mo_tp_pool
  --- Logical volume ---
  LV Name                          mo_tp_pool
  VG Name                          vg_thin
  LV UUID                          dxkVxN-Vkdm-gDNA-gUq7-Curt-8Yrd-rqfzOY
  LV Write Access                  read/write
  LV Pool transaction ID 3
  LV Pool metadata                 mo_tp_pool_tmeta
  LV Pool data                     mo_tp_pool_tdata
  LV Pool chunk size               64.00 KiB
  LV Zero new blocks               yes
  LV status                        available
  # open                           0
  LV Size                          15.00 GiB
  Allocated pool data              29.41%
  Allocated metadata               7.03%
  Current LE                       480
  Segments                         1
  Allocation                       inherit
  Read ahead sectors               auto
  - currently set to               256
  Block device                     253:5
```

As we can see, in out thin pool we have only 29.41% of data written totally.

We also can use Over Provisioning in our system. To do it, we only need to create a new thin volume of 5GB of storage space and give it to a 4th user. So, with Over Provisioning we give more space than what we have. This kind of system it will only work if our 4 users don't fill up their volumes.

It is noteworthy that thin pool are just a logical volume, so if we need to extend the size of thin pool we can use the same command like we've used for extend logical volumes, but we can't reduce the size off thin pool.

2.3 RAID

2.3.1 Introduction

RAID (redundant array of independent disks) is a storage technology that combines multiple disk drive components (array of disks) into a logical unit. The objective of this system is to protect data against hardware failure by adding redundancy. The general principle is: the data is stored across multiple physical disks instead of just one, giving the system a configurable level of redundancy. Depending on the amount of redundancy, in case of unexpected disk failure, we can reconstruct the data from the remining disks. Data is distributed across the drives in one of several ways called "RAID levels", depending on the level of redundancy and performance required. RAID can be implemented by hardware or by software.

2.3.2 RAID Levels

RAID 0

RAID 0 implements a "striped" disk array, the data is broken down into blocks and each block is written to a separate disk drives. I/O performance is greatly improved by spreading the I/O load across many channels and drives. This level provides the best performance and there is no calculation overhead involved. The design is very simple and easy to implement.

The main disadvantages of RAID 0 is that it is not fault-tolerant. The failure of just one drive will result in all data in an array being lost. For this reason, this level should never be used in mission critical environments. In this context, possible applications using RAID 0 can be video/image production and editing or in general, any application requiring high bandwidth.

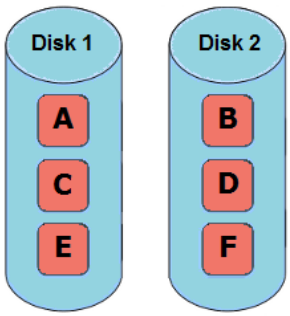


Figure 2.3: RAID 0.

RAID 1

RAID 1, also known as "mirroring", it's the simplest and most commonly used configuration. Consists of an exact copy (or mirror) of a set of data on two or more disks. In its classic form, it used two physical disks of the same size and provides a new logical volume,again of the same size. Data are stored identically in both discs, so, in a RAID 1 pair there is one write or two reads possible per mirrored pair of disks. That is to say, twice the read transaction rate

of single disks and the same write transaction rate as single disks. No rebuild is necessary in case of a disk failure, we just use the mirror and replace the failed disk. So, RAID 1 provides redundancy of data. Transfer rate per block is equal to that of a single disk.
When used with several disks, RAID 1 can sustain multiple simultaneous drive failures. Also RAID 1 implemetations usually support hot swap of failed disks. The design is very simple and easy to implement. The main disadvantage of RAID 1 is that it has the highest disk overhead of all RAID types. In this context, possible applications using RAID 1 can be financial operations, critical data or any application requiring very high availability.

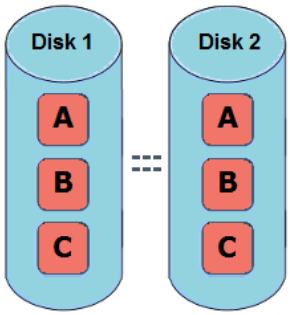


Figure 2.4: RAID 1. Block Mirrored.

RAID 5

RAID 5 consists of block-level striping with distributed parity. Unlike in RAID 4 (which is not used in practice), parity information is distributed among the drives. This avoids the bottleneck problem in RAID-4, and also allows getting more performance out of the disk when reading, as all drives will then be used.

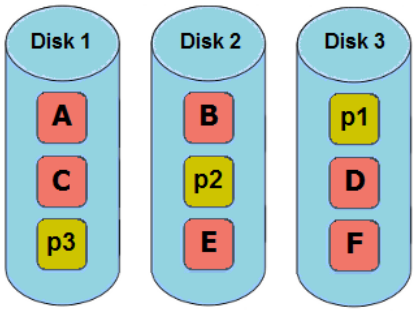


Figure 2.5: RAID 5. Blocks Striped. One Distributed Parity.

This is perhaps the most useful RAID mode when one wishes to combine a larger number of physical disks, and still maintain some redundancy. RAID-5 can be (usefully) used on three or more disks, with zero or more spare-disks. If one of the disks fail, all data are still intact, thanks to the parity information. If spare disks are available, reconstruction will begin immediately after the device failure. If two disks fail simultaneously, or before the raid is reconstructed, all data are lost. RAID-5 can survive one disk failure, but not two or more. Both read and write performance usually increase, but can be hard to predict how much.

Reads are almost similar to RAID-0 reads, writes can be either rather expensive (requiring read-in prior to write, in order to be able to calculate the correct parity information, such as in database operations), or similar to RAID-1 writes (when larger sequential writes are performed, and parity can be calculated directly from the other blocks to be written). The write efficiency depends heavily on the amount of memory in the machine, and the usage pattern of the

array. Heavily scattered writes are bound to be more expensive.

This RAID level is recommended for file and application servers, database servers, Web, E-mail, etc. because it is a very versatile RAID level.

RAID 6

This is an extension of RAID-5 to provide more resilience. RAID-6 can be (usefully) used on four or more disks, with zero or more spare-disks. The big difference between RAID 5 and RAID 6 is that in RAID 6 there are two different parity information blocks, and these are distributed evenly among the participating drives. Since there are two parity blocks; if one or two of the disks fail, all data is still intact. If spare disks are available, reconstruction will begin immediately after the device failure(s). Read performance is almost similar to RAID 5 but write performance is worse.

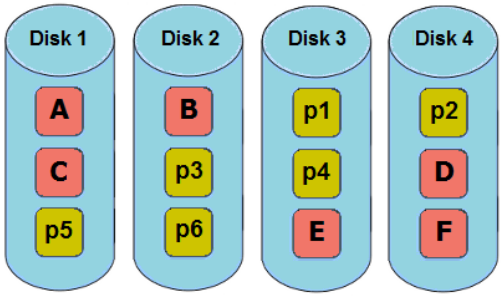


Figure 2.6: RAID 6. Blocks Striped. Two Distributed Parity.

Nested RAID

Levels of nested RAID (also known as hybrid RAID) combine two or more of the standard levels of RAID to gain performance, additional redundancy, or both. Typical examples are RAID 0+1 and RAID 1+0, also called RAID 01 and RAID 10 respectively.

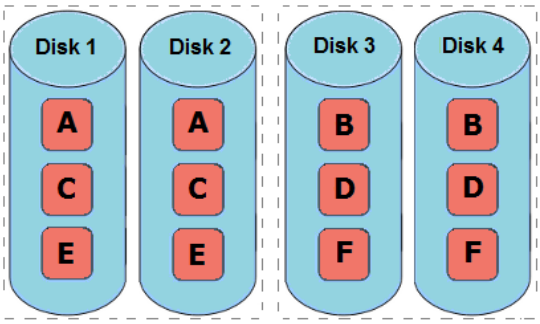


Figure 2.7: RAID 10. Block Mirrored and Blocks Striped.

RAID 10 is also called as RAID 1+0. It is also called as “stripe of mirrors” It requires minimum of 4 disks To understand this better, it is just a way of grouping the disks in pair of two (for mirror). For example, if we have a total of 6 disks in RAID 10, there will be three groups–Group 1, Group 2, Group 3 as shown in the Figure. Within the group, the data is mirrored. Across the group, the data is striped. i.e Block A is written to Group 1, Block B is written to Group 2, Block C is written to Group 3. Disks within the group are mirrored and groups are striped.

RAID 01 is also called as RAID 0+1 It is also called as “mirror of stripes” In most cases this will be implemented as minimum of 4 disks. To understand this better, create two groups. For example, if we have total of 6 disks, we create two groups with 3 disks each. Within the group, the data is striped. That is to say, in the Group 1 which contains three disks, the 1st block will be written to 1st disk, 2nd block to 2nd disk, and the 3rd block to 3rd disk. So, block A is written to Disk 1, block B to Disk 2, block C to Disk 3. Across the group, the data is mirrored. The Group 1 and Group 2 will look exactly the same. Disk 1 is mirrored to Disk 4, Disk 2 to Disk 5, Disk 3 to Disk 6. This is why it is called “mirror of stripes”. Disks within the groups are striped and groups are mirrored.

Performance and storage capacity is the same on both RAID 10 and RAID 01. The main difference is the fault tolerance level because of the way controllers are usually implemented:

- In RAID01, if a disk fails on a group, we are forced to replace it and then rebuild the group. While the group is not rebuilt it is considered as failed. RAID 01 would, therefore, only be practically able to lose one disk. If a second disk fails on the other group for some reason, even if it held another part of the RAID0 data, the controller will fail the RAID0 portion of that group as well.
- In RAID 10, the mirroring is on the lower layer rather than the upper layer. Thus, we might lose up one disk per group and still have a functioning RAID array. For this reason, RAID10 is preferable to RAID01.

2.3.3 RAID in Linux

Required Software

A few useful commands to get us started:

- `lsblk` to see mount points
- `df` to see file systems and
- `fdisk` to see partition types

In addition, we have `mdadm`, a simple user space command to use RAID. To install this tool type:

```
$ sudo apt-get install mdadm
```

The post-installation configuration raises two questions. The first one asks whether the root filesystem of our Linux will be in a RAID in order to start it before the boot sequence. In our example, will use RAID only for data discs, so the answer the first question is none. The second question asks about whether we want the RAID disks to be initialized automatically at each startup of the machine or we must do it manually. In this case, we want to do it automatically (using /etc/mdadm/mdadm).

Loop Disks

Instead of using real disks, we will use loop disks in a file.

```
# dd if=/dev/zero of=/virtualfs bs=1024 count=3145728
```

Then we check if he loop device loop0 is in use:

```
# losetup /dev/loop0
```

If loop0 is in use we can use loop1, loop2,... Linux has 8 loop devices by default.
If free, we will use it to mount the image file system in the LXC default path:

```
# losetup /dev/loop0 /virtualfs
```

Creating RAIDs

The basic command to create a RAID is:

```
# mdadm --create MD_DEVICE --level=X --raid-devices=Y SD_DEV_A SD_DEV_B ...
```

Or the abbreviated format:

```
# mdadm --create MD_DEVICE -l X -n Y SD_DEV_A SD_DEV_B ...
```

Where X is the level of RAID (Linux supports among others 0, 1, 5, 6 and 10) and Y the number of disks that form the array. MD_DEVICE is the device name that will receive the new logical disk (which should be in the directory /dev and, by convention, called md followed by a number, for example /dev/md0). SD_DEV_Z is a disk (from /dev). We can also add the option `--verbose` for more detailed information creation process and possible faults. An example to create a RAID 0 with three disks:

```
# mdadm --create --verbose /dev/md0 --level=0 --raid-devices=3 /dev/sda /dev/sdb /dev/sdc
```

The tool supports, apart from these, other advanced options. For further information, check the manual (`man mdadm`). In the following example we create a RAID 5:

```
# mdadm --create --verbose /dev/md0 -l 5 -n 3 /dev/sda /dev/sdb /dev/sdc
```

If everything is ok, the command will finish with showing some informational messages and indicating that the disc md0 has been initialized and started. Besides, we can check the status with:

```
# lsblk -fm
```

or

```
# more /proc/mdstat
```

To use the RAID disk, we have to format and mount it. For example, if we use the ext4 filesystem and /mnt/data as mount directory:

```
# mkfs.ext4 /dev/md0
# mkdir /mnt/data
# mount /dev/md0 /mnt/data
```

If we use the command `df` to show information about the partition space and the percentage of use:

```
# df -k /mnt/data
....
```

We see that, indeed, we have about XXXMB RAID5 for the size three discs we mounted 500GB.

On the other hand, during installation we told `mdadm` to start all RAID arrays automatically, but that does not mean the corresponding filesystems get mounted. To do so, we have to add a line like the following to the file /etc/fstab:

```
/dev/md0  /mnt/data  ext4  defaults  0  1
```

Management

Let's now have a look at various management commands. If we want to stop the volume we use the argument `--stop`. Do not forget to unmount it before to prevent loss of data:

```
# umount /dev/md0
# mdadm --stop /dev/md0
```

Then, to start it again manually:

```
# mdadm --assemble /dev/md0 /dev/sdb /dev/sdc /dev/sdd
```

And if we had included the line indicated earlier in fstab, to remount the disk in our file system just type:

```
# mount /mnt/data
```

To view information on all arrays we have running on the machine we use the argument –detail:

```
# mdadm --detail --verbose --scan
....
```

For more detail about a particular RAID array:

```
# mdadm --detail --verbose /dev/md0
....
```

We can also use the argument `--examine` to get information about the status of any of the disks that make up our RAID:

```
# mdadm --examine /dev/md0
....
```

Errors

It must be stressed that if the RAID was not fully initialized (for example because we used very large disks) the volume will not boot properly. Therefore, it is very important not start using RAID arrays until they are fully initialized. We can check this by monitoring the file /proc/mdstat. We should see a progress bar while building or rebuilding the RAID. This bar disappears after reaching 100%, then it is save to use the RAID.

Now, we will simulate a break in one of our discs. We can turn off the machine and disconnect one of the disks in the RAID. After restarting, we will have lost our RAID fault tolerance but our data is still in /mnt/data.

To make the fault tolerance work again we must add a new disk:

```
# mdadm /dev/md0 --add /dev/sdf
```

After the previous command, the RAID is rebuilt automatically. Monitor /proc/mdstat to check when the rebuilding process ends.

We should take into account that when a machines losses a disk the device names can change. It is very important to check before rebuilding the RAID with `lsblk`, for example, which devices are still part of the RAID and which device is the one that we want to add.

Hot Spare Disks

A hot spare disk is a disk used to replace a failed disk in both hardware and software RAID systems configuration. With hot spare disk we reduce the mean time to recovery (MTTR) for the RAID redundancy group, thus reducing the probability of a second disk failure and the resultant data loss that would occur. When employing a hot spare disk, time for the data to be generated onto the spare disk is required. During this time the system is exposed to data loss due to a subsequent failure. Automatic switching to a spare disk reduces the time of exposure to that risk compared to manual discovery and implementation.

With mdadm we can define spare disks to automatically being used in case of failure. If we want to add a new RAID disk to be used as a hot spare, we just add it to a RAID already created with the same command used earlier to add a disk to a broken RAID:

```
# mdadm /dev/md0 --add /dev/sdg
```

Increasing the Size

If we want to add disks to increase the size of the RAID (not to be used as a hot spare disks), we have to use the following commands:

```
# mdadm /dev/md0 --add /dev/sde
# mdadm --grow /dev/md0 --raid-devices=4
```

Wait until the RAID is rebuilt and then, we have to make our filesystem inside the RAID also grow. This can be done for EXT systems as follows:

```
# umount /dev/md0
# e2fsck -f /dev/md0
# resize2fs /dev/md0
# mount /dev/md0 /mnt/data
```

Removing

If we want to eliminate one of the disks in the RAID, for example, to replace it with another one, we have to mark it as failed. After that, we can remove it. Example:

```
# mdadm /dev/md0 --fail /dev/sde
# mdadm /dev/md0 --remove /dev/sde
```

If we want to remove the complete RAID, we have to use this command:

```
# mdadm --remove /dev/md0
```

Monitoring

The `mdadm` command also has an argument called `--monitor` that lets us generate alerts when an incident occurs in the RAID. In Debian-like systems, the monitor is started as a daemon that records incidents in the log file of the machine (/var/log/syslog) using the following command:

```
# mdadm --monitor --pid-file /run/mdadm/monitor.pid --daemonise --scan --syslog
```

Finally, we can send incidents to our email account adding the option `--mail`. For example:

```
# mdadm --monitor --pid-file /run/mdadm/monitor.pid --daemonise --scan
--syslog --mail user@example.com
```

2.4 LVM vs RAID

Both RAID and LVM provide distinct advantages when compared to the simple case of a desktop computer with only one hard disk in which usage patterns do not change over time. But between the two sistems, which use?

Of courese, the most appropriate answer depend on current and anticipated requirements. If the requirements are to protect data against hardware failure, obviously RAID is the answer, as LVM really does not solve this problem. On the other hand, if we need a flexible storage scheme in which volumes are independent of the physical layout of the disks, RAID is not very helpful and LVM is the natural choice.

Then, of course, there's the really interesting use case, in which the storage system must be resistant to hardware failures and also flexible in the allocation of volumes. Neither LVM or RAID can solve both requirements themselves; no matter, this is the situation in which we will use both at the same time, or rather, one over the other. To do this, first we must ensure redundancy in the data, grouping discs in fewer large RAID arrays and then use these RAID arrays as LVM physical volumes. The strength of this configuration is that when a disk fails, we only need to reconstruct a small amount of RAID arrays, thereby limiting the time used by the administrator to recover.

2.5 BTRFS

2.5.1 Introduction

Linux has a wealth of filesystems from which to choose, but we are facing a number of challenges with scaling to the large storage subsystems that are becoming common in today's data centers. Filesystems need to scale in their ability to address and manage large storage, and also in their ability to detect, repair and tolerate errors in the data stored on disk. B-Tree File System is GPL-licensed. This project was started in 2007 by Chris Mason at Oracle and it is based on the idea of storing data and metadata in B-tree structures. Since Sun Microsystems announced a new filesystem called ZFS, which would be implemented in Solaris and OpenSolaris, the Linux community began looking for an equivalent non-proprietary filesystem, and that is how BTRFS started.

Btrfs is the next-generation Linux filesystem all cram-full of advanced features designed for maximum data protection and massive scalability such as copy-on-write, storage pools, checksums, 16-exabyte filesystems, journaling, online grow and shrink, and space-efficient live snapshots. If we're accustomed to using LVM and RAID to manage our data storage, Btrfs can replace these. A snapshot is a copy of a Btrfs subvolume at a particular point in time. It's many times faster than making a traditional backup, and incurs no downtime. We can make snapshots of a filesystem whenever we want, and then quickly roll back to any of them. Next we provide a list of the current of Btrfs features according to the official documentation:

- Extent based file storage (an extent is a contiguous area of storage reserved for a file)
- 2^64 byte == 16 EiB maximum file size
- Space-efficient packing of small files
- Space-efficient indexed directories
- Dynamic inode allocation
- Writable snapshots, read-only snapshots
- Subvolumes (separate internal filesystem roots) C
- hecksums on data and metadata (crc32c)
- Compression (zlib and LZO)
- Built-in RAID functionality (File Striping, File Mirroring, File Striping+Mirroring, Striping with Single and Dual Parity implementations).
- SSD (Flash storage) awareness (TRIM/Discard for reporting free blocks for reuse) and optimizations (e.g. avoiding unnecessary seek optimizations, sending writes in clusters, even if they are from unrelated files. This results in larger write operations and faster write throughput)
- Efficient Incremental Backup
- Background scrub process for finding and fixing errors on files with redundant copies
- Online filesystem defragmentation
- Offline filesystem check
- Conversion of existing ext3/4 file systems
- Seed devices. Create a (readonly) filesystem that acts as a template to seed other Btrfs filesystems. The original filesystem and devices are included as a readonly starting point for the new filesystem. Using copy on write, all modifications are stored on different devices; the original is unchanged.
- Subvolume-aware quota support
- Send/receive of subvolume changes
- Batch, or out-of-band deduplication (happens after writes, not during)

Additional features in development, or planned, include:
- Very fast offline filesystem check
- Object-level mirroring and striping
- Alternative checksum algorithms
- Online filesystem check
- Other compression methods (snappy, LZ4)
- Hot data tracking and moving to faster devices

BTRF is now considered stable, although is under heavy development.

2.5.2 Create a Filesystem

Let's show this filesystem practically. First, we must make sure btrfs-tools are installed in our system:

```
$ sudo apt-get install btrfs-tools
```

Now let's create four dummy storage devices:

```
# dd if=/dev/zero of=/tmp/btrfs-vol0.img bs=1G count=1
# dd if=/dev/zero of=/tmp/btrfs-vol1.img bs=1G count=1
# dd if=/dev/zero of=/tmp/btrfs-vol2.img bs=1G count=1
# dd if=/dev/zero of=/tmp/btrfs-vol3.img bs=1G count=1
```

Then, we attach them to loop devices so that they can be used as regular disk drives:

```
# losetup /dev/loop0 /tmp/btrfs-vol0.img
# losetup /dev/loop1 /tmp/btrfs-vol1.img
# losetup /dev/loop2 /tmp/btrfs-vol2.img
# losetup /dev/loop3 /tmp/btrfs-vol3.img
```

BTRFS uses different strategies to store data and for the filesystem metadata. With no other options this command creates a three-node RAID array, using RAID 0 for data and RAID 1 for metadata. By default the behavior is that **metadata** is replicated on all of the devices. If a single device is used the metadata is duplicated inside this single device (useful in case of corruption or bad sector, there is a higher chance that one of the two copies is clean). On the other hand, **data** is spread amongst all of the devices. This means no redundancy; any data block left on a defective device will be inaccessible.

To create a BTRFS volume made of multiple devices with default options, we can type:

```
# mkfs.btrfs -f -L mybtrfs /dev/loop0 /dev/loop1 /dev/loop2
```

The `mkfs.btrfs` command has more interesting options such as:

```
-f forces an overwrite of any existing filesystems.
-L creates a filesystem label, which is any name you want to give it.
-m LEVEL is the level of redundancy for the metadata.
   At this moment, LEVEL can take the values single, raid0 or raid1.
-d LEVEL is the level of redundancy for the data.
```

To view the btrfs filesystems in a system we can use these commands:

```
# btrfs device scan
# btrfs filesystem show
```

Option names of `btrfs` can also be abbreviated. Example:

```
# btrfs fi sh
```

We can also use the traditional `blkid` command to gather information (including the UUIDs and UUID_SUBs):

```
# blkid  /dev/loop*
```

Notice that the UUIDs on the three partitions in our storage volume are the same, but the UUID_SUBs are unique. If we run the `blkid` command before creating the storage pool, the UUIDs will also be unique.

2.5.3 Mount the Filesystem

To mount our btrfs filesystem we create a directory and use the `mount` command. Mounting any single device mounts the whole works, like this:

```
# mkdir /mnt/mybtrfs
# mount /dev/loop1 /mnt/mybtrfs
```

In addition, for mount the filesystem when the system reboots we can add an entry in /etc/fstab: Note. We use our label or the UUID (not the UUID_SUB) like one of these examples:

```
LABEL=mybtrfs   /mnt/mybtrfs   btrfs   defaults   0 0
UUID=b6a05243   /mnt/mybtrfs   btrfs   defaults   0 0
```

To view the size, the typical commands `du` and `df` are not suitable because they do not understand Btrfs metadata, RAID, and how storage is managed. Measuring available space on a Btrfs volume is tricky because of these factors. The following command has to be used:

```
# btrfs filesystem df /mnt/mybtrfs
Data, RAID0: total=9.00GB, used=6.90GB
Data: total=8.00MB, used=0.00
System, RAID1: total=8.00MB, used=4.00KB
System: total=4.00MB, used=0.00
Metadata, RAID1: total=1.00GB, used=46.01MB
Metadata: total=8.00MB, used=0.00
```

We could also try it on any raw device in the storage pool:

```
# btrfs filesystem show /dev/loop1
...
```

Finally, if we use raid1 or raid10 for data AND metadata and we have a usable submirror accessible (consisting of 1 drive in case of RAID1 or the two drive of the same RAID0 array in case of RAID10), we can mount the array in degraded mode in the case of some devices are missing (e.g. dead SAN link or dead drive):

```
# mount -o degraded /dev/loop0 /mnt/mybtrfs
```

If we use RAID0 (and have one of ours drives inaccessible) the metadata or RAID10 but not enough drives are on-line to even get a degraded mode possible, btrfs will refuse to mount the volume.

2.5.4 Change Redundancy

Without having to format the filesystem, we can convert the data blocks, which by default are RAID0 to RAID1. Example:

```
# btrfs fi df /mnt/mybtrfs
.....
# btrfs balance start -dconvert=raid1 /mnt/mybtrfs
Done, had to relocate 1 out of 3 chunks
```

Check the space usage again.

```
# btrfs fi df /mnt/mybtrfs
Data, RAID1: total=1.00GiB, used=768.00KiB
System, RAID1: total=32.00MiB, used=16.00KiB
Metadata, RAID1: total=1.00GiB, used=112.00KiB
```

Now, in RAID1, the system should still be able to survive a corrupted disk. We will simulate a corrupted disk by using `dd` to write some zeros at several zones of one of the drives:

```
# dd if=/dev/zero of=/dev/sdb bs=1M count=1K seek=1K
# dd if=/dev/zero of=/dev/sdb bs=1M count=1K seek=10K
# dd if=/dev/zero of=/dev/sdb bs=1M count=1K seek=200K
# dd if=/dev/zero of=/dev/sdb bs=1M count=1K seek=100K
```

Now, we will scrub to manually check the data blocks and recover corrupted ones. The scrub starts in the background and Btrfs will silently detect and repair corrupted blocks as it finds them.

```
# btrfs scrub start /mnt/test
scrub started on /mnt/test, fsid befcfef9-54c0-4c72-9283-23096ab5f909 (pid=21488)
```

We will run a status check once, to see that the scrubbing is still in progress:

```
# btrfs scrub status /mnt/mybtrfs
scrub status for befcfef9-54c0-4c72-9283-23096ab5f909
 scrub started at Tue Aug 26 13:19:25 2014, running for 110 seconds
 total bytes scrubbed: 2.83GiB with 169424 errors
 error details: csum=169424
 corrected errors: 169424, uncorrectable errors: 0, unverified errors: 0
# diff -qr /mnt/mybtrfs/data /media/nas/data
```

Finally, we run the status command again to check that the scrub is complete, and get the repair statistics:

```
# btrfs scrub status /mnt/mybtrfs
scrub status for befcfef9-54c0-4c72-9283-23096ab5f909
 scrub started at Tue Aug 26 13:19:25 2014 and finished after 1647 seconds
 total bytes scrubbed: 84.27GiB with 238575 errors
 error details: csum=238575
 corrected errors: 238575, uncorrectable errors: 0, unverified errors: 0
```

2.5.5 Add/remove devices

Now, we are going to add a new device, /dev/loop3, and check filesystem information.

```
# btrfs device add /dev/loop3 /mnt/mybtrfs
# btrfs fi sh
Label: none uuid: befcfef9-54c0-4c72-9283-23096ab5f909
 Total devices 3 FS bytes used 42.14GiB
 devid 1 size 1.82TiB used 44.03GiB path /dev/sdb
 devid 2 size 1.82TiB used 44.03GiB path /dev/sdc
 devid 3 size 2.73TiB used 0.00 path /dev/sdd
Btrfs v3.12
```

Notice that the new disk, which has 2.73TB usable capacity, is empty. It remains empty until some new data is written. However, it is possible to manually re-balance the disks:

```
# btrfs balance start /mnt/mybtrfs
Done, had to relocate 45 out of 45 chunks
```

Let's check the filesystem to observe the operation:

```
# btrfs fi sh
Label: none uuid: befcfef9-54c0-4c72-9283-23096ab5f909
 Total devices 3 FS bytes used 42.14GiB
 devid 1 size 1.82TiB used 22.03GiB path /dev/sdb
 devid 2 size 1.82TiB used 22.00GiB path /dev/sdc
 devid 3 size 2.73TiB used 44.03GiB path /dev/sdd
Btrfs v3.12
```

Now, we are going to remove a drive:

```
# btrfs device delete /dev/loop3 /mnt/mybtrfs
```

Let's check the filesystem statistics:

```
# btrfs fi sh
Label: none uuid: befcfef9-54c0-4c72-9283-23096ab5f909
 Total devices 2 FS bytes used 42.14GiB
 devid 1 size 1.82TiB used 44.03GiB path /dev/sdb
 devid 3 size 2.73TiB used 44.03GiB path /dev/sdd
Btrfs v3.12
```

And verify the data:

```
# diff -qr /mnt/mybtrfs/data /media/nas/data
```

2.5.6 Shrinking/Growing Volumes

A common practice in system administration is to leave some head space, instead of using the whole capacity of a storage pool (just in case). With btrfs one can easily shrink volumes. Let's shrink the volume a bit (about 25%):

```
# btrfs filesystem resize -500m /mnt/mybtrfs
```

The previous command does an on-line resize, there is no need to umount/shrink/mount. However, a BTRFS volume requires a minimal size, if the shrink is too aggressive the volume won't be resized. On the other hand, we can also grow a volume:

```
# btrfs filesystem resize +150m /mnt
Resize '/mnt' of '+150m'
```

Finally, we can also use all of the possible space for the volume:

```
# btrfs filesystem resize max /mnt
Resize '/mnt' of 'max'
```

2.5.7 Subvolumes

A way of understanding subvolumes in BTRFS is them as boxes. Each one of those can contain items and other smaller boxes ("sub-boxes") which in turn can also contains items and boxes (sub-sub-boxes) and so on. Each box and items has a number and a name. The top level box has only a number (zero). More technically, a Btrfs subvolume can be thought of as a separate POSIX file namespace.

A subvolume in btrfs is not like an LVM logical volume, which is quite independent from each other, a btrfs subvolume has its hierarchy and relations between other subvolumes. A subvolume in btrfs can be accessed in two ways:

1. **From the parent subvolume**. When accessing from the parent subvolume, the subvolume can be used just like a directory. It can have child subvolumes and its own files/directories.

2. **Separate mounted filesystem**. This is achieved using the mount command (or fstab) with the subvol or subvolid mount option. On a mounted subvolume, one can access files/directories/subvolumes inside it, but nothing in parent subvolumes.

Also every btrfs filesystem has a default subvolume as its initially top-level subvolume. Subvolumes can be nested. Nested subvolumes appear as subdirectories within their parent subvolumes, similar to the way top-level subvolume presents its subvolumes as subdirectories. Deleting a subvolume deletes all subvolumes below it in the nesting hierarchy, and for this reason the top-level subvolume cannot be deleted. Any Btrfs file system always has a default subvolume, which is initially set to be the top-level subvolume, and is mounted by default if no subvolume selection option is passed to `mount`. The default subvolume can be changed if required.

We can also mount a subvolume without mounting the top-level subvolume that contains it. Creating new subvolumes is just as easy as creating new directories, by using the `btrfs` command. The syntax is:

```
create [-i <qgroupid>] [<dest>]<name>

Create a subvolume <name> in <dest>.
If <dest> is not given, subvolume <name> will be created in the currently directory.
-i <qgroupid> adds the new subvolume to a qgroup (this option can be given multiple times).
```

Note. Quota groups are described later in Section 2.5.9. The following examples creates three new subvolumes on a mounted top-level subvolume:

```
# btrfs subvolume create /mnt/mybtrfs/sub1
# btrfs subvolume create /mnt/mybtrfs/sub2
# btrfs subvolume create /mnt/mybtrfs//sub2/sub3
```

To see our subvolumes:

```
# btrfs subvolume list /mnt/mybtrfs/
ID 260 gen 22 top level 5 path sub1
ID 261 gen 22 top level 5 path sub2
ID 262 gen 22 top level 5 path sub2/sub3
```

To mount a subvolume all by itself, without the top-level subvolume, first unmount the top-level subvolume, and then mount the subvolume using its ID:

```
# umount /mnt/mybtrfs/
# mount -o subvolid=261 /dev/loop1 /mnt/mybtrfs/
```

We used /dev/loop1 because it is part of the Btrfs storage pool (we can use any block device in the pool to mount it). Now let's make subvolid 261 the default subvolume:

```
# btrfs subvolume set-default 261 /mnt/mybtrfs
```

Then unmount it, and remount it just like the top-level subvolume example:

```
# umount /mnt/mybtrfs/
# mount /dev/loop1 /mnt/mybtrfs/
```

And behold, we will see only the single subvolume. To put everything back the way it was, re-set the default using the 0 ID, which is always the top-level subvolume:

```
# btrfs subvolume set-default 0 /mnt/mybtrfs/
```

Unmount, remount, and everything is back to the default. To delete a subvolume, make sure it is mounted, and then:

```
# btrfs subvolume delete /mnt/mybtrfs/sub2/sub3
```

We cannot delete a subvolume that contains another subvolume, so we have to start at the end of the line and work backwards.

To configure a subvolume mount in /etc/fstab:

```
LABEL=mybtrfs   /mnt/mybtrfs defaults,subvolid=269 0 0
```

As a final remark, we would like to mention that a subvolume in Btrfs is quite different from a traditional Logical Volume Manager (LVM) logical volume. With LVM, a logical volume is a separate block device, while a Btrfs subvolume is not.

2.5.8 Snapshots

A Btrfs snapshot is actually a subvolume that shares its data (and metadata) with some other subvolume, using Btrfs' copy-on-write capabilities, and modifications to a snapshot are not visible in the original subvolume. Once a writable snapshot is made, it can be treated as an alternate version of the original file system. For example, to rollback to a snapshot, a modified original subvolume needs to be unmounted and the snapshot needs to be mounted in its place. At that point, the original subvolume may also be deleted.

The copy-on-write (CoW) nature of Btrfs means that snapshots are quickly created, while initially consuming very little disk space. Since a snapshot is a subvolume, creating nested snapshots is also possible. Taking snapshots of a subvolume is not a recursive process; thus, if a snapshot of a subvolume is created, every subvolume or snapshot that the subvolume already contains is mapped to an empty directory of the same name inside the snapshot. Let's see snapshots in action. First copy some files into one of our subvolumes, and then make a snapshot of it:

```
# btrfs subvolume snapshot /btrfs/sub1 /btrfs/sub1/snapshot
```

Our new snapshot will automatically appear in our filesystem, just like a new subvolume, and it will include all the files in the original subvolume. Remember, it behaves independently of the original subvolume, so we can do whatever we want to it without affecting the original. Suppose we make a big mess and we want to roll back to a known good state. It's a good thing we made a snapshot before the mess happened. First unmount the mangled subvolume, then mount the snapshot in its place. If we decide we do not need the mangled subvolume anymore we can delete it and rename the snapshot with the same name as the mangled subvolume, so we do not have to change configuration files like /etc/stab. Simple use the `mv` command for renaming:

```
# mv /btrfs/snapshotname /btrfs/subvolumename
```

2.5.9 Quota Groups

In many situations, it would be nice to be able to divide a volume into subvolumes but not to allow any given subvolume to soak up all of the available storage space. These needs can be met through the Btrfs subvolume quota group mechanism. A quota group (or qgroup) imposes an upper limit to the space a subvolume or snapshot may consume.

However, we must mention that Btrfs quotas not normal, per-user disk quotas (those can be managed on Btrfs just like with any other filesystem). Btrfs subvolume quotas, instead, track and regulate usage by subvolumes, with no regard for the ownership of the files that actually take up the space. By default, Btrfs filesystems do not have quotas enabled. To turn this feature on, run:

```
# btrfs quota enable path
```

Limits can be applied to subvolumes with a command like:

```
# btrfs qgroup limit 30M /mnt/1/subv
```

The limits are somewhat approximate. The quota group is somewhat more flexible than has been shown so far; it can, for example, organize quotas in hierarchies that apply limits at multiple levels (for further information, check the Btrfs documentation).

2.6 DAS

Traditionally application servers used to have their own storage devices attached to them. This is also called Direct-attached storage (DAS). DAS refers to a digital storage system directly attached to a server or workstation, without a storage network in between. For years, a parallel interface (several wires) using the ATA or SCSI standards has been widely used to make DAS connections:

- **SCSI** (Small Computer System Interface) provides an interface for data transmission. In addition, we can attach multiple devices to a single SCSI port, so that SCSI is really an I/O bus rather than simply an interface.
- **ATA** (Advanced Technology Attachment), also known as **IDE** (Integrated Drive Electronics), is a disk drive implementation that integrates the controller on the disk drive itself. ATA is used to connect hard disk drives, CD-ROM drives and similar peripherals.

The need for increased bandwidth in storage systems made the SCSI and ATA standards using parallel interfaces an inefficient option. For this reason, serial interfaces were developed. In contrast to multiple parallel data stream, in a serial interface data is transmitted serially, that is in a single steam, by wrapping multiple bits into packets. Serial interfaces are able to move that single stream faster than parallel technology. The serial interface reduces the amount of wires needed to transmit data, making for much smaller cable size and making it easier to route and install devices. The serial interface also implements two data ports to enable complete failover redundancy. If one path fails, there is still communication along a separate and independent path. Serial versions of SCSI and ATA were developed:

- **SAS** (Serial Attached SCSI) is an evolution of parallel SCSI into a point-to-point serial peripheral interface in which controllers are linked directly to disk drives. SAS is a performance improvement over traditional SCSI because SAS enables multiple devices (up to 128) of different sizes and types to be connected simultaneously with thinner and longer cables. In addition, SAS drives can be hot-plugged.
- **SATA** (Serial ATA) is an evolution of the Parallel ATA physical storage interface. Serial ATA is a serial link.

SAS and SATA increased several orders of magnitude the speed of their predecessors. Another important fact is that the backplanes of SAS and SATA devices were made identical. This is important since it means that SAS and SATA drives can operate in the same environment while SCSI and ATA could not (they were physically incompatible).

In general, SAS are faster and more expensive drives than SATA drives. On the other hand, SATA drives are low-cost and high-capacity drives. Then, since SAS and SATA drives are compatible, we can use faster SAS drives for primary storage and offloading older data to cheaper SATA disks in the same subsystem, something that could not be achieved with SCSI and ATA. This is typically called a SAS infrastructure.

To connect to devices, the host uses a host bus adapter (HBA). The HBA is a hardware device that provides I/O processing and physical connectivity between a host system, such as a server, and a storage device. An HBA port can easily support a multitude of SATA and SAS drives. As more SATA drives and SAS backplanes are added to a storage enclosure, SAS expanders on those backplanes can be cascaded together.

With only a minimal number of ports on the HBA required, and with more ports available on the expanders, a SAS infrastructure becomes increasingly cost effective as the number of drives rises. When a growing organization moves into online/transactional applications that involve critical data needing high availability to multiple, concurrent users, adding high-performance SAS drives requires no modification to the existing infrastructure.

2.7 NAS

2.7.1 Introduction

A Network-attached storage (NAS) is a file-level computer data storage server connected to a computer network, providing data access to a heterogeneous group of clients. NAS systems are networked appliances which contain one or

more hard drives, often arranged into logical, redundant storage containers or RAID. NAS uses file-based protocols such as NFS (popular on UNIX systems) and SMB/CIFS (Server Message Block/Common Internet File System). NAS units rarely limit clients to a single protocol.

Regarding DAS, NAS can make it easier to promote file sharing and scale high-performance storage than direct-attached storage (DAS), which is non-networked storage that connects to a dedicated server or computer hard drive. Multiple NAS boxes can be attached to a network to expand storage capacity. NAS systems also provide a single view of available storage to LAN users. Conversely, DAS requires that the storage for each server be managed separately, complicating data transfer in a networked environment, although DAS may provide higher performance than NAS for accelerating certain applications.

NAS provides both storage and a file system. This is often contrasted with SAN (Storage Area Network), which provides only block-based storage and leaves file system concerns on the "client" side. One way to loosely conceptualize the difference between a NAS and a SAN is that NAS appears to the client OS (operating system) as a file server (the client can map network drives to shares on that server) whereas a disk available through a SAN still appears to the client OS as a disk, visible in disk and volume management utilities (along with client's local disks), and available to be formatted with a file system and mounted.

Finally, there are also clustered NAS, which is a NAS that is using a distributed file system running simultaneously on multiple servers. The key difference between a clustered and traditional NAS is the ability to distribute (e.g. stripe) data and metadata across the cluster nodes or storage devices. Clustered NAS, like a traditional one, still provides unified access to the files from any of the cluster nodes, unrelated to the actual location of the data.

2.7.2 NFS and CIFS

Currently, the two main protocols to build a NAS are NFS and CIFS. NFS (Network File System) comes from the Unix world. In terms of history, this protocol was initially conceptualized and used by Sun Microsystems, back in 1984.

On the other hand, CIFS (Common Internet File System) is actually the public version of SMB (Server Message Block protocol), invented by Microsoft. The mechanism enables joint sharing of multiple devices such as printers, files, and even serial ports, among various users and administrators. CIFS is regarded as the more chatty, or talkative network system protocol, when compared to NFS.

2.7.3 NFS Configuration

In this section we provide the configuration necessary to set an easy NFS configuration without security. For a more professional use of NFS, we should probably consider that the NFS user permissions are based on user ID (UID). UIDs of any users on the client must match those on the server in order for the users to have access. The typical ways of doing this are manual password file synchronization, LDAP and NIS. Additionally, portmap lockdown can be used through the files /etc/hosts.deny and /etc/hosts.allow.

Server

First, in the server we must install the required packages. This can be done as follows:

```
# apt-get install nfs-kernel-server
```

For easier maintenance we will isolate all NFS exports in single directory, where the real directories will be mounted with the –bind option. Let's say we want to export our users' home directories in /home/users. First we create the export filesystem:

```
# mkdir -p /export/users
```

It's important that /export and /export/users have 777 permissions as in this case, we will be accessing the NFS share from the client without authentication. Now mount the real users directory with:

```
# mount --bind /home/users /export/users
```

To save us from retyping this after every reboot we can use /etc/fstab:

```
/home/users     /export/users    none     bind   0   0
```

There are three configuration files that relate to an NFS server: /etc/default/nfs-kernel-server, /etc/default/nfs-common and /etc/exports. The only important option in /etc/default/nfs-kernel-server for now is NEED_SVCGSSD. It is set to "no" by default, which is fine, because we are not activating NFSv4 security this time. If we want ID names to be mapped, both the client and server require the /etc/idmapd.conf file to have the same contents with the correct domain names.

To export our directories to a local network 192.168.1.0/24 we add the following two lines to /etc/exports:

```
/export         192.168.1.0/24(rw,fsid=0,insecure,no_subtree_check,async)
/export/users   192.168.1.0/24(rw,nohide,insecure,no_subtree_check,async)
```

Let's comment some options that can be used by the server in the NFS share:

- **rw**. This option gives the client computer both read and write access to the volume.
- **sync/async**. This option forces NFS to write changes to disk before replying. This results in a more stable and consistent environment, since the reply reflects the actual state of the remote volume. We can use async, which is faster but less stable.
- **nosubtreecheck**. This option prevents subtree checking, which is a process where the host must check whether the file is actually still available in the exported tree for every request. This can cause many problems when a file is renamed while the client has it opened. In almost all cases, it is better to disable subtree checking.
- **norootsquash**. By default, NFS translates requests from a root user remotely into a non-privileged user on the server. This was supposed to be a security feature by not allowing a root account on the client to use the filesystem of the host as root. This directive disables this for certain shares.
- **insecure**. This option allows clients such as Mac OS X to connect on ports above 1024.
- **nohide**.This option allows the client to map the sub-exports within the psuedo filesystem.
- **fsid=0**. This option indicates that this this the default share.

Now restart the service:

```
# service nfs-kernel-server restart
```

Client

On the other hand, in the client we must install the required packages:

```
# apt-get install nfs-common
```

On the client we can mount the complete export tree with one command:

```
# mount -t nfs -o proto=tcp,port=2049 nfs-server:/ /mnt
```

In NFSv4, we can use nfs-server:/ instead of nfs-server:/export. The root export :/ defaults to export with fsid=0. We can also mount an exported subtree with:

```
# mount -t nfs -o proto=tcp,port=2049 nfs-server:/users /home/users
```

To save us from retyping this after every reboot we add the following line to /etc/fstab:

```
nfs-server:/      /mnt/nfs   nfs  auto,wsize=32768,rsize=32768      0 0
1.2.3.4:/var      /mnt/var   nfs  auto,nobarrier,noatime,nfsver=4   0 0
```

Let's comment some common options that the client can use to mount the NFS share:

- **auto/noauto**. This options stop the init scripts from mounting it.
- **sync/async**. With the sync option, NFS requests are replayed only after the changes have been committed to stable storage. With async, there can be replies to requests before any changes made by that request have been committed to stable storage (e.g. disc drive). Using this option usually improves performance, but at the cost that an unclean server restart (i.e. a crash) can cause data to be lost or corrupted.
- **nobarrier**. This option disables using barriers. This greatly increases performance but might affect filesystem integrity.
- **noatime**. This option disables access time updates on every file read.
- **nodiratime**. This option is like noatime but for directories.
- **nfsvers=4**. This option specifies which version of the NFS protocol to use ().
- **rsize=num and wsize=num**. These settings speed up NFS communication for reads (rsize) and writes (wsize) by setting a larger data block size, in bytes, to be transferred at one time. Be careful when changing these values; some older Linux kernels and network cards do not work well with larger block sizes. For NFSv2 or NFSv3, the default values for both parameters is set to 8192. For NFSv4, the default values for both parameters is set to 32768.
- **hard or soft**. These specify whether the program using a file via an NFS connection should stop and wait (hard) for the server to come back online, if the host serving the exported file system is unavailable, or if it should report an error (soft). If hard is specified, the user cannot terminate the process waiting for the NFS communication to resume unless the intr option is also specified. If soft is specified, the user can set an additional timeo=<value> option, where <value> specifies the number of seconds to pass before the error is reported. Using soft mounts is not recommended as they can generate I/O errors in very congested networks or when using a very busy server.
- `sec=mode`. This option specifies the type of security to utilize when authenticating an NFS connection. sec=sys is the default setting, which uses local UNIX UIDs and GIDs.
- **bg**. This option specifies that the NFS mount must continue in the background.

We can check our NFS mounted shares with:

```
$ df -h
```

If we want to see all of the NFS shares that we have mounted, we can type:

```
$ mount -t nfs
```

2.8 SAN

2.8.1 Introduction

As previously mentioned, there is a proliferation of servers with many applications and information. If we set each server with its DAS, the problem is that in big structures with many servers, storage may remain isolated and under-utilized. This implies a high cost of managing information. SAN provides a solution for this.

A SAN can be defined as a set of storage devices that are accessible over the network at a block-level. This differs from a Network Attached Storage (NAS) device in that a NAS runs its own filesystem and presents that volume to the network. SAN storage does not need to be formatted by the client machine. Whereas a NAS usually is presented with the NFS or CIFS. SAN are mainly implemented with three technologies:

- Fiber Channel (FC). This is not just the transmission medium but also an architecture and several protocols.
- IP. A SAN can also be implemented using IP.
- FC over Ethernet (FCoE). FC can be used over Ethernet.

2.8.2 FC

Fiber channel is a high-speed network technology (Gb/s), highly scalable. A single FC network can accommodate millions of devices. FC is a protocol rather than the physical media. In fact, today there are implementations of SAN using cooper cables too. SAN is used for the connection between servers and storage arrays. Clients still connect to servers using IP. FC has very little transmission overhead since it is a network technology designed for storage.

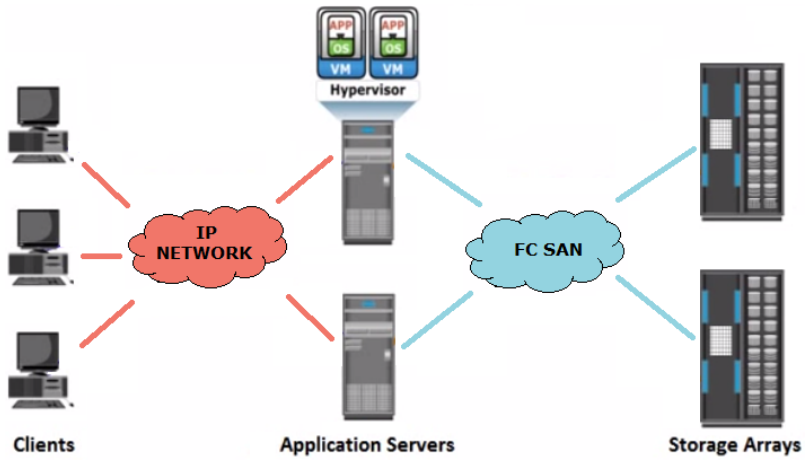


Figure 2.8: SAN and IP Networks.

The FC SAN has the following components:

- Node (server and storage) ports or HBA (Host Bus Adapter). Each port has separate transmit link (Tx) and a receive link (Rx).
- Cables. SAN uses optical fiber cables for long distance and might use cooper cables for short distances. The optical cables are single-mode (long distances up to 10km) or multimode (medium distances because of modal dispersion).
- Interconnection devices: hubs, switches and directors.
  - Hubs provide limited connectivity and scalability.
  - Switches and directors are intelligent devices. Switches have a fixed number of ports while directors are always modular. Line cards or blades can be inserted. High-end switches and directors contain redundant components.
- SAN management software.

FC has three interconnectivity options: point-to-point, fiber channel arbitrated loop and fiber channel switched fabric.

In point-to-point each server is connected to the storage array. This is used in DAS, however, for SAN is not scalable since as the number of servers grow this topology requieres more and more physical cards and cables.

With an FC Hub a shared loop is created. Nodes must artitrate to gain control and it can be used implementing a ring or start logical topology (like Ethernet hubs or token ring). The limitation of FC-AL is that only one device can perform I/O operations at a time. FC-AL supports up to 126 nodes and the addition or removal of a node causes momentary pauses in loop traffic (we need to reinitialize the loop).

Finally, the third option is FC-SW creates a logical space in which all nodes communicate using switches. In the FC terminology this switched network is called a “fabric”. Connection between switches are called ISLs (Interswitch Links). This type of topology provides dedicated paths between nodes. The addition/removal of nodes in this case does not affect traffic of other nodes. For each single fabric there is a domain identifier and each port has an address

for communication.

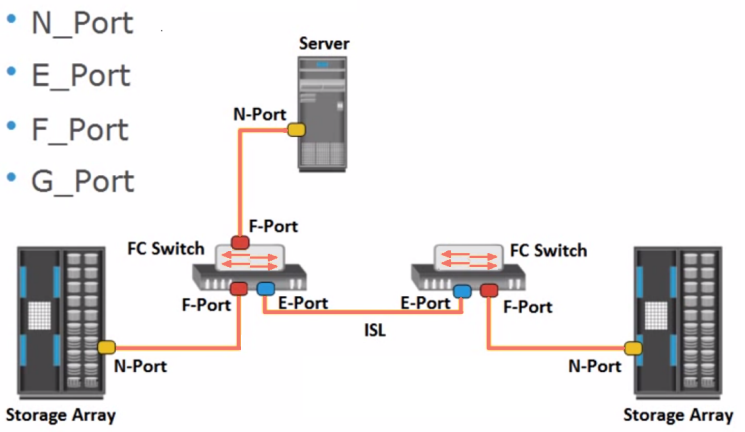
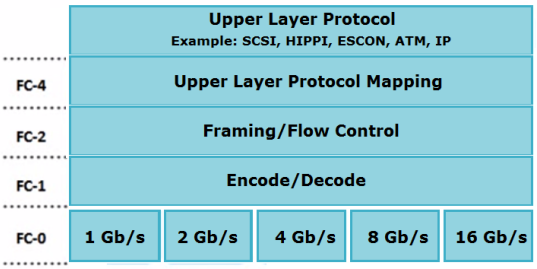


Figure 2.9: FC Ports.

N: end node
E: Expansion port
F: port on a switch to an end node.
G: can be F or E (autodetected).
FC Protocol.

Traditional channels for disk communications using protocols like SCSI have limitations in the number of devides that can be connected and distance constrains. Network technologies are more flexible but they have higher overhead. FC provides the benefits of both channel and network technologies. It is a network technology specifically designed for storage with low overhead. Figure 2.10 shows the protocol stack of the FC protocol.



FC Layer	Function	Features Specified by FC Layer
FC-4	Mapping interface	Mapping upper layer protocol (e.g. SCSI) to lower FC layers
FC-3	Common services	Not implemented
FC-2	Routing, flow control	Frame structure, FC addressing, flow control
FC-1	Encode/decode	8b/10b or 64b/66b encoding, bit and frame synchronization
FC-0	Physical layer	Media, cables, connector

Figure 2.10: FC Protocol Stack.

As shown in the Figure 2.10, as the top layer (FC-4) we can use SCSI (which is the typical situation). In a switched fabric, a FC address is assigned to nodes during fabric login. The format of the address is as shown in Figure 2.11. The Domain ID is a unique number provided to each switch in the fabric (there are 239 possible addresses because some are reserved). The area is used to define a group of ports (e.g. all the ports in a port card). Therefore, the maximum possible number of node ports in a switched fabric is: $239\,domains \times 256\,areas \times 256\,ports = 15,663,104$.

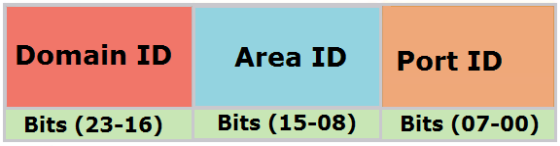


Figure 2.11: FC Addresses.

In addition FC uses world wide names (WWN). WWN are unique world wide 64 bit identifiers. They are similar to MAC addresses in the Ethernet. WWNN identify nodes and WWPN identify ports (see Figure 2.12). These identifiers are static (unlike FC addresses).

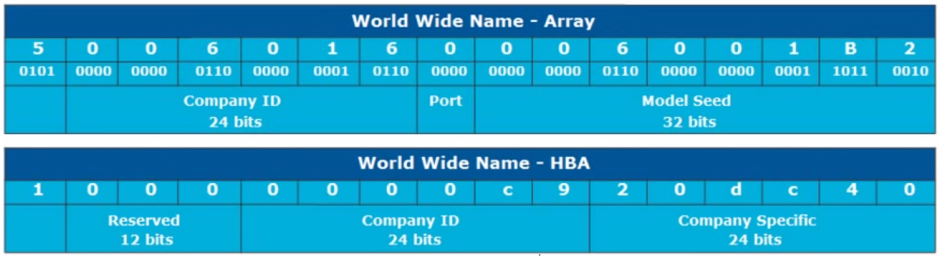


Figure 2.12: WWN.

An FC name server maps WWN to FC addresses. FC addresses are used for routing and WWN for naming.

FC data is organized as Frame, Sequence and Exchange. A frame is the fundamental unit of data transfer. Each frame consists of fie parts: SOF (4 bytes), frame header (24 bytes), data (0-2112 bytes), CRC (4 bytes) and EOF (4 bytes). Sequences are a contiguous set of frames that correspond to an information unit. An information unit is a protocol-specific upper layer message that is sent to another port to perform a certain action. Finally, an exchange enables two N_Ports to identify and manage a set of information units, which in turn, include one or more sequence.

The downside to Fibre Channel is the expense. A Fibre Channel switch often runs many times the cost of a typical Ethernet switch and comes with far fewer ports.

2.8.3 FC over IP (FCIP)

Another possibility to connect distributed FC SAN islands is to use FCIP (see Figure 2.13). In this case, we create virtual FC links over existing IP networks that are used to transport FC data between different FC SANs. FC frames are encapsulated onto IP packets. This is an effective and reduced cost way of providing a recovery solution.

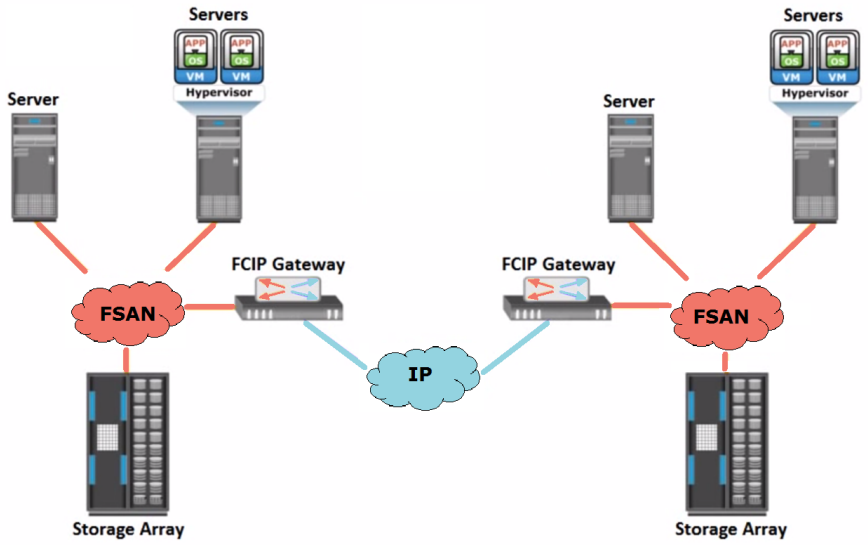


Figure 2.13: FCIP Topology.

The FCIP protocol stack is shown in Figure **??**.

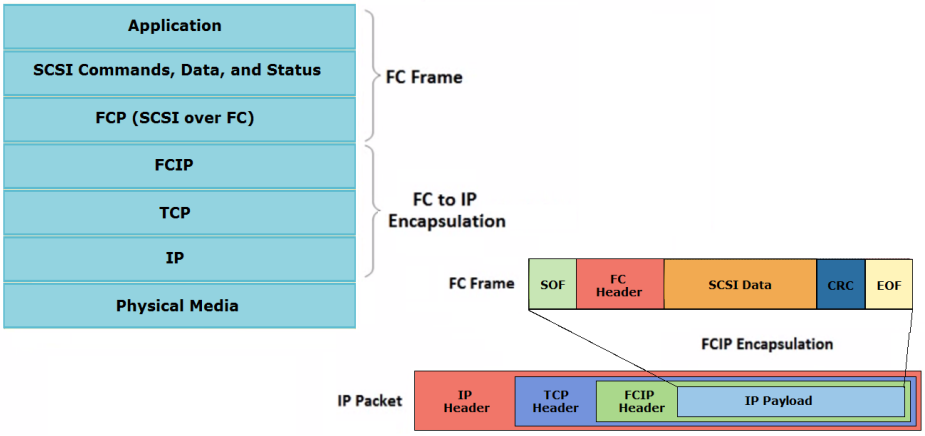


Figure 2.14: FCIP Protocol Stack.

2.8.4 FC over Ethernet (FCoE)

FC over Ethernet (FCoE) is a protocol that transports FC data over Ethernet. This option is interesting because it enables the consolidation of FC SAN traffic and Ethernet traffic onto a common Ethernet infrastructure. This reduces the number of adapters, switch ports, cables, power and cooling cost and floor space (see Figure 2.15 for a typical FC deployment with out FCoE). The cost is also reduced because the Ethernet switches are much cheaper than FC switches and eases data center management.

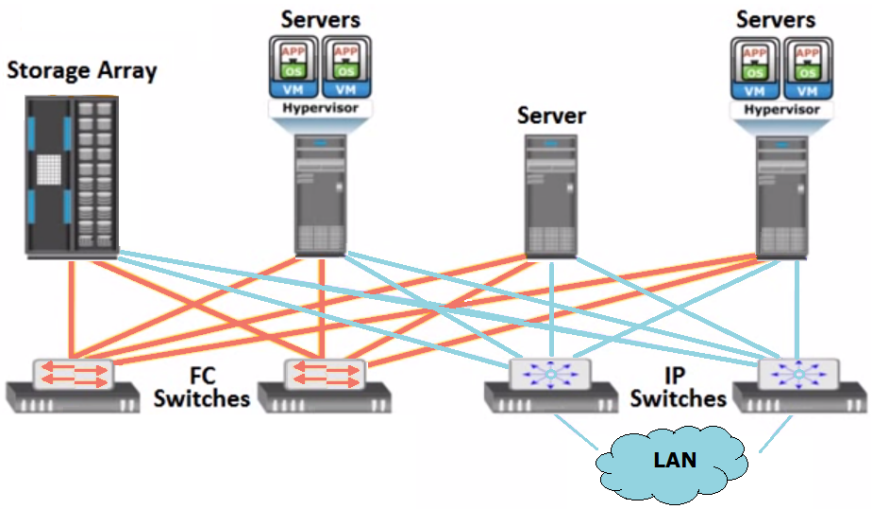


Figure 2.15: Data Centers with FC Before FCoE.

Figure 2.16 shows a data center deploymentwith FCoE.

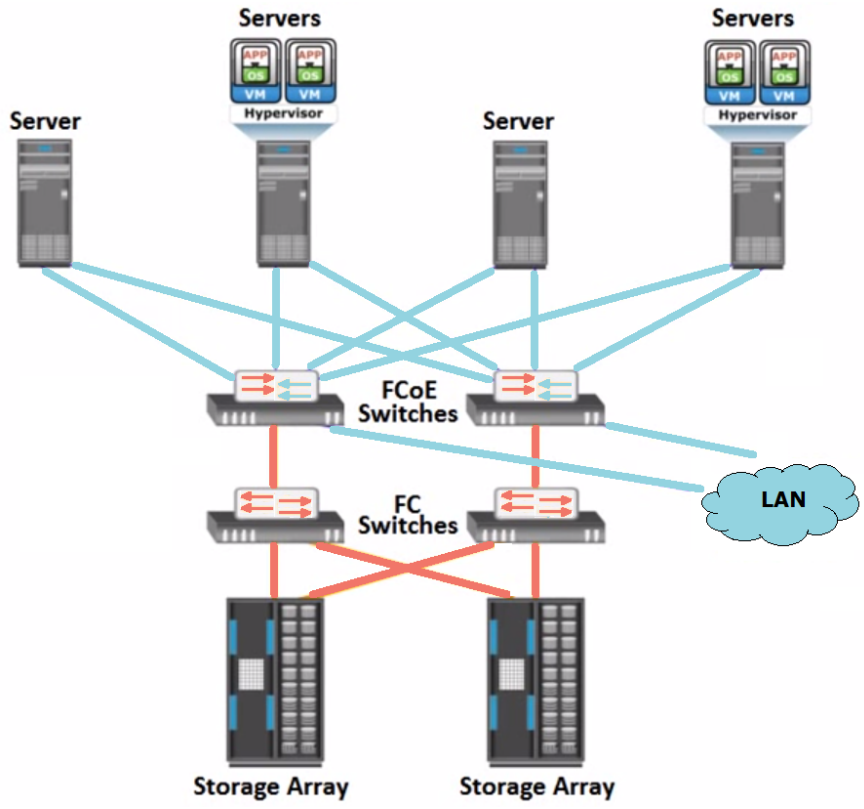


Figure 2.16: Data Centers with FCoE.

As shown, the number of cables, connectors etc. is reduced. The key components of a FCoE are cables, converged network adapters (CNA) and FCoE switches. CNAs (see Figure 2.17) provides the functionality of both, a standard

NIC and an FC HBA. This eliminates the need to deploy separate adapters and cables for FC and Ethernet communications. The CNA card contains separate modules (ASICs) for Gigabit Ethernet, FC and FCoE. In particular, the FCoE ASIC encapsulates FC frames into Ethernet frames. The FCoE switch provides both Ethernet and FC switch functionalities. Consists of FCF, Ethernet bridge and a set of CEE ports and FC ports. The switch forwards the frames based on the Ethertype.

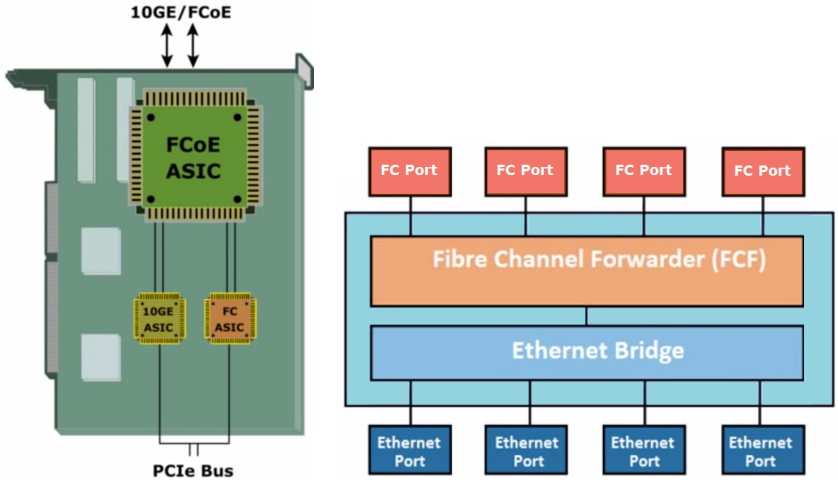


Figure 2.17: CNA and FCoE Switch.

The FCoE protocol stack is shown in Figure 2.18.

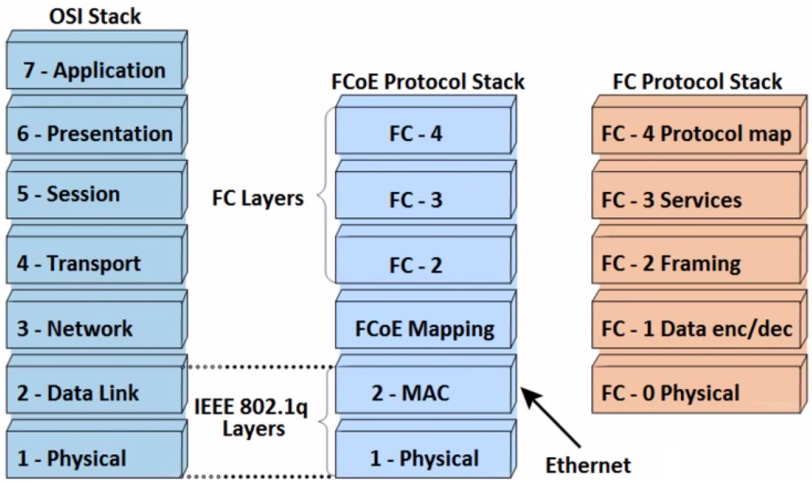


Figure 2.18: FCoE Protocol Stack.

2.8.5 iSCSI

iSCSI is the same SCSI protocol used for local disks, but encapsulated inside IP to allow it to run over the network in the same way any other IP protocol does. Because of this, and because it is seen as a block device, it often is almost indistinguishable from a local disk from the point of view of the client's operating system and is completely transparent to applications. The iSCSI protocol runs over TCP ports 860 and 3260.

The main advantage of using IP is that we can use existing network infrastructure and that the cost is much reduced compared to FC SANs. In addition, IP is a mature and very robust technology with many solutions for disaster and recovery. It is also possible to mix iSCSI SANs with FC SANs. In this case, an iSCSI gateway is used (see Figure 2.19).

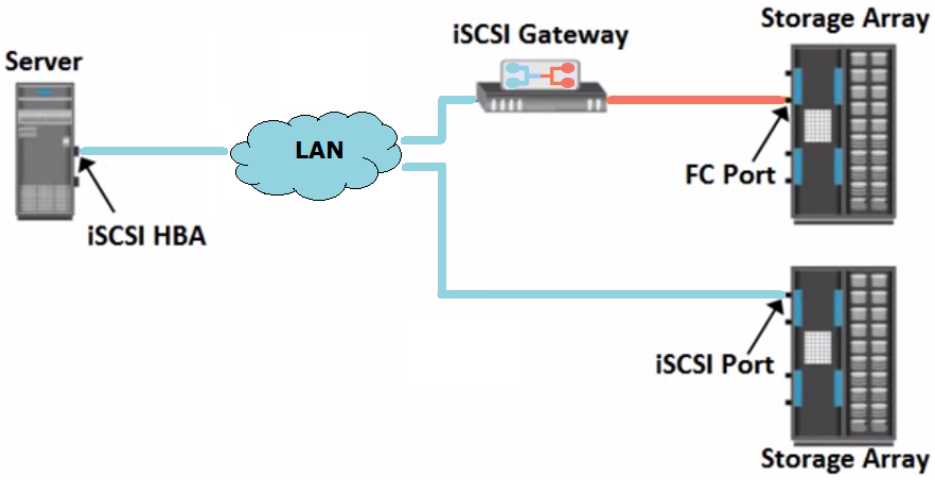


Figure 2.19: iSCSI Components.

The iSCSI protocol refers to clients as initiators and iSCSI servers as targets. There are three possible ways of connecting a iSCSI device:

- Standard NIC with software iSCSI initiator. In this case, the NIC provides the network interface and the software initiator provides the iSCSI functionality. This option requires host processor cycles for iSCSI and TCP/IP processing.

- TCP Offload Engine (TOE) NIC with a software iSCSI initiator. In this case, the TCP processing load is moved off the host onto the NIC card. Software initiator provides the iSCSI functionality and therefore, it only requires host CPU cycles for iSCSI processing.

- iSCSI HBA. In this case, both, the iSCSI and the TCP/IP processing are offloaded from the host processor. Also, this configuration provides the simplest option for booting from the SAN.

The iCSI protocol stack can be observed in Figure **??**.

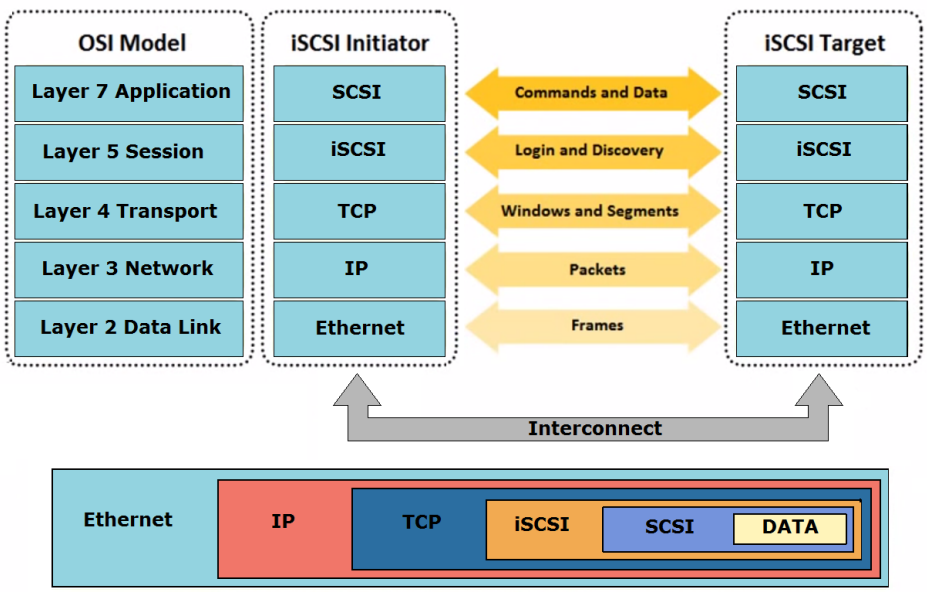


Figure 2.20: iSCSI Protocol Stack.

The iSCSI session layer is responsible for implementing the login and discovery of iSCSI devices.
In other words, for iSCSI communication, the initiator must discover the location and name of the target on the network. A possibility is to manually configure in the initiator the target's network portal. Then, the initiator issues a SendTargets command and the target responds with the required parameters. Another possibility is to use an Internet Storage Name Service (iSNS). In this case, initiators and targets register themselves with the iSNS server. Then, the initiator can query the iSNS server for a list of available targets.

Target have names which are formatted specially. The official term for this name is the iSCSI Qualified Name (IQN). The format is:

```
iqn.yyyy-mm.(reversed domain name):label
```

where iqn is required, yyyy signifies a four-digit year, followed by mm (a two-digit month) and a reversed domain name, such as com.example. The label is a user-defined string in order to better identify the target.

An example is:

```
iqn.2015-01.com.example:myiscsi-target
```

For each target, exported disks must have a unique LUN. LUN stands for logical unit number. A LUN represents an individually addressable (logical) SCSI device that is part of a physical SCSI device (target). In an iSCSI environment, LUNs are essentially numbered disk drives. An initiator negotiates with a target to establish connectivity to a LUN; the result is an iSCSI connection that emulates a connection to a SCSI hard disk. Initiators treat iSCSI LUNs the same way as they would a raw SCSI or IDE hard drive; for instance, rather than mounting remote directories as would be done in NFS or CIFS environments, iSCSI systems format and directly manage filesystems on iSCSI LUNs.

Target

The Linux kernel has adopted LIO-Target as the standard iSCSI target for Linux. LIO-Target is available in Linux kernels 3.1 and higher.

To get LIO to work, we only need to install a command line administration tool called targetcli to create and publish our iSCSI targets. To install this tool:

```
# apt-get install --no-install-recommends targetcli python-urwid
```

The LUNs (disks) presented by the iSCSI target can be entire disks, partitions, plain files on the filesystem, ram disks, etc.

The first step is to create the backing store for the LUN. In this example, we will use a file-backed LUN, which is just a normal file on the filesystem of the iSCSI target server:

```
# targetcli
/> cd backstores/
/backstores> ls
o- backstores                [...]
o- fileio                    [0 Storage Object]
o- iblock                    [0 Storage Object]
o- pscsi                     [0 Storage Object]
o- rd_dr                     [0 Storage Object]
o- rd_mcp                    [0 Storage Object]
/backstores> cd fileio
/backstores/fileio> help create
/backstores/fileio> create lun0 /root/iscsi-lun0 2g  (create 2GB file-backed LUN)
```

Now the LUN is created. Next we will set up the target so client systems can access the storage. In first place, we create the iqn and target port group:

```
/backstores/fileio/lun0> cd /iscsi
/iscsi>  create wwn=iqn.2003-01.org.setup.lun.test

Created target iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1.
Selected TPG Tag 1.
Successfully created TPG 1.
Entering new node /iscsi/iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1/tpgt1
```

For simplicity, we disable authentication (we can enable it and it is CHAP):

```
/iscsi/iqn.2...a672ba1/tpgt1> set attribute authentication=0
```

And, we create the target LUN:

```
/iscsi/iqn.2...a672ba1/tpgt1> cd luns
/iscsi/iqn.20...a1/tpgt1/luns> create /backstores/fileio/lun0
Selected LUN 0.
Successfully created LUN 0.
Entering new node /iscsi/iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1/tpgt1/luns/lun0
```

iSCSI traffic can consume a lot of bandwidth, so we'll probably want the iSCSI traffic to be on a dedicated (or SAN) network, rather than our public network.
To do so:

The `create` command, creates a portal to listen for connections:

```
/iscsi/iqn.2...gt1/luns/lun0> cd ../../portals
/iscsi/iqn.20...tpgt1/portals> create 10.10.102.164

Using default IP port 3260
Successfully created network portal 10.10.102.164:3260.
Entering new node
/iscsi/iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1/tpgt1/portals/10.10.102.164:3260
/iscsi/iqn.20...102.164:3260> cd ..
/iscsi/iqn.20...tpgt1/portals> create 10.11.102.164

Using default IP port 3260
Successfully created network portal 10.11.102.164:3260.
Entering new node
```

```
/iscsi/iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1/tpgt1/portals/10.11.102.164:3260
```

Now, we'll just need to register the iSCSI initiators (client systems). To do this, we'll need to find the initiator names of the systems. For Linux, this will usually be in /etc/iscsi/initiatorname.iscsi.

(register initiator — this IQN is the IQN of the initiator — do this for each initiator that will access the target)

```
/iscsi/iqn.20...102.164:3260> cd ../../acls
/iscsi/iqn.20...a1/tpgt1/acls> create iqn.1994-05.com.redhat:f5b312caf756

Successfully created Node ACL for iqn.1994-05.com.redhat:f5b312caf756
Created mapped LUN 0.
Entering new node
/iscsi/iqn.2003-01.org.linux-iscsi.murray.x8664:sn.31fc1a672ba1/tpgt1/acls/iqn.
 1994-05.com.redhat:f5b312caf756
```

Now, remember to save the configuration. Without this step, the configuration will not be persistent.

```
/iscsi/iqn.20...102.164:3260> cd /
/> saveconfig
/> exit
```

The configuration files are stored in the directory /etc/target.

Initiator

Now, we need to connect our initiators to the target. After the connection is made, the client systems will see a new disk. The disk will need to be formatted before use. To configure our Linux box as an iSCSI initiator, we have to install the open-iscsi package:

```
$ sudo apt-get install open-iscsi
```

Once the open-iscsi package is installed, edit /etc/iscsi/iscsid.conf changing the following:

```
node.startup = automatic
```

We can check which targets are available in a server by using the iscsiadm utility. Example:

```
$ sudo iscsiadm -m discovery -t st -p 192.168.0.1:3260

  192.168.0.1:3260,1 iqn.2007-01.org.debian.foobar:CDs
  192.168.0.1:3260,1 iqn.2007-01.org.debian.foobar:USB
```

Where:

```
-m: determines the mode that iscsiadm executes in.
-t: specifies the type of discovery.
-p: option indicates the target IP/Port.
```

Next let's open a session to a given Target:

```
$ sudo iscsiadm  -m node  --targetname "iqn.2007-01.org.debian.foobar:CDs" \
--portal "192.168.0.1:3260" --login
```

If this is successful, a new /dev/sdx device will show up, and automatically linked to /dev/disk/by-path/ip-*. We can check the dev device also with `dmesg | grep sd`. We can then mount, format, etc. the new device.

Once we have finished working, we can release the iSCSI target.

```
iscsiadm  -m node  --targetname "iqn.2007-01.org.debian.foobar:CDs" --portal "192.168.0.1:3260" --logout
```

The iSCSI initiator also has an iqn which we will find in /etc/iscsi/initiatorname.iscsi. After installation of open-iscsi it just contains "GenerateName=yes". During the first start of the demon the iqn gets generated.

2.8.6 SAN versus NAS

Greater support for applications. Some applications and application deployment scenarios either aren't certified or simply cannot operate over NFS. Some of the limitations, like the lack of NFS support with Microsoft Exchange Server, will always exist with NFS. Microsoft Cluster Services (MCS) is one example where SCSI pass thru is required to deploy and SCSI-3 reservation support. While this capability is not in the current VVols beta, one should expect that VMware will look to provide this in a future release and in turn retire the need for RDMs.

Enhanced storage bandwidth capabilities. This is an area where SAN has been and will continue to be more capable than NAS. The VVols beta currently does not include NFS v4.1 with pNFS, the latter is required for link aggregation. As such each VVol will be limited to a single link for IO with NFS and multi-links with SAN via VMware's native or 3rd party multipathing software.

Enhanced storage IO capabilities. Like the previous item, this is an area where SAN has been and will continue to be more capable than NAS. As the IO to an NFS VVol is limited to a single Ethernet link, any congestion cannot be addressed by VMware's native or 3rd party multipathing software. Link congestion could manifest itself in a storage VMotion operation which will actual add load to the link, exacerbating the problem before it can shift IO access for the VM to another link.

Although a SAN often is faster than a NAS, it also is less flexible. For example, the size of or the filesystem of a NAS usually can be changed on the host system without the client system having to make any changes. With a SAN, because it is seen as a block device like a local disk, it is subject to a lot of the same rules as a local disk. So, if a client is running its /usr filesystem on an iSCSI device, it would have to be taken off-line and modified not just on the server side, but also on the client side. The client would have to grow the filesystem on top of the device.

There are some significant differences between a SAN volume and a local disk. A SAN volume can be shared between computers. Often, this presents all kinds of locking problems, but with an application aware that its volume is shared out to multiple systems, this can be a powerful tool for failover, load balancing or communication. Many filesystems exist that are designed to be shared. GFS from Red Hat and OCFS from Oracle (both GPL) are great examples of the kinds of these filesystems.

2.9 Object Storage

Today we have more and more digital data, and many of them are difficult to control because they are unstructured and displayed in various formats such as emails, documents, images or videos. This growth of unstructured data is a challenge for storage managers as companies require data to be accessible immediately to meet the requirements of their users.

It has been attempted to solve this problem using traditional storage management system like NAS, which manage data as a file hierarchy, or SAN (Block Storage) which manages data as blocks within sectors and tracks, but these are proving to be expensive and inadequate.

An emerging alternative used for example by Quantum LattusCaringo Swarm, EMC Atmos, Hitachi HCP or OpenStack Swift is Object Storage. Also called object-based storage, it is a new storage architecture that manages data as objects. Each object typically includes three things:

- **The data itself.** It can be anything that users want to store, from an image to a instructions manual.
- **A variable amount of metadata.** This metadata is defined by the creator of the object storage. It contains contextual information about what the data is, its confidenciality, its putpose and anything else that is relevant to the way in which data must be used.

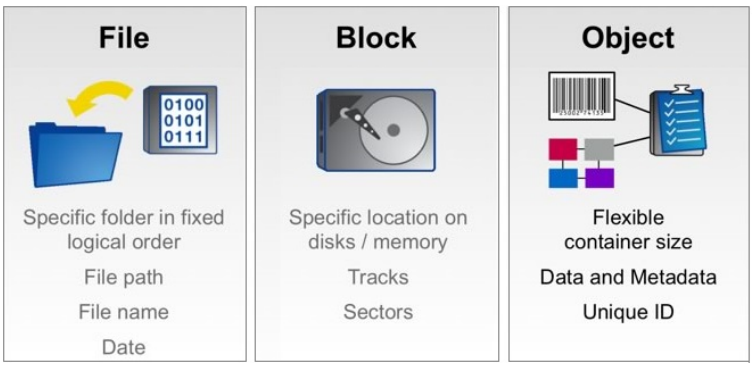


Figure 2.21: File vs Block vs Object.

- **A globally unique identifier.** Each object has an adress (identifier) in order for the object to be found over a distributed system. This ID serves as a pointer to a specific object, much the same way a universal resource locator (URL) points to a specific file on a web server. This way, users are able to find the data without having to know the physical location of the data.

Object storage can be implemented at multiple levels, including the device level (object storage device), the system level, and the interface level. In each case, object storage seeks to enable capabilities not addressed by other storage architectures, like interfaces that can be directly programmable by the application, a namespace that can span multiple instances of physical hardware, and data management functions like data replication and data distribution at object-level granularity.

Object storage systems are accessed using a REST-based interface, using lower-level PUT and GET commands. This allows applications to directly access data without using traditional file system protocols and for object storage to be easily connected to via the internet.

2.9.1 Benefits versus previous technologies

As explained in previous sections file storage is a type of storage that stores data in a hierarchical structure, with directories, sub-directories and files. In file storage, when a file is stored to disk it is split into thousands of pieces, each with its own address. When that file is needed the user enters the server name, directory and file name, the file system finds all of its pieces and reassembles them. It is great and works rightly when the number of files is not very large or when we know exactly where our files are stored, but when we manage large amounts of data, like servers with astronomical number of files, this hierarchical structure has no sense, and folders are just a waste of space and time.

Regarding to block storage, we know that a block is a chunk of data, and when appropriate blocks are combined, it creates a file. With block storage, files are split into evenly sized blocks of data, each with its own address but with no additional information (metadata) to provide more context for what that block of data is.

Object storage, by contrast, doesn't split the file. The file is stored as a single object complete with metadata, and is assigned an ID and stored as close to contiguously as possible. When a user need a content, he only needs to present the ID to the system and the content will be fetched along with all the metadata, which can include security, authentication, etc. This can happen directly over the Web, eliminating the need for Web servers and load balancers.

In Object storage there is no limit on the type or amount of data and metadata, which makes Object storage powerful and customizable. Because Object storage technologie treats all types of data and can save it as an object and store multiple copies of it, often on different storage servers and drives, it is said that Object storage can replace RAID for data availability.

2.9.2 What Object Storage Provides

Aside from the benefits of object storage versus previous technologies that we explained in the previous chapter, object-based architectures provide the next characteristics:

2.9.2.1 File Storage compatibility

Object storage is a natural for unstructured data since files are also discrete data entities. By making each file an object, as is common with larger files, data access is achieved with a simple index instead of a complex system of i-nodes, directories and folders. A file system layer can be added to an object storage system to essentially perform protocol translations and create a compelling alternative to a traditional NAS system.

2.9.2.2 Scalability

Scalability it's the ability of a storage system to continue to function well as it is changed in size or volume in order to meet a user need. It has always been an important characteristic, a key attribute of storage systems, especially in cloud environments. Aside from expanding physical capacity, as explained below, scaling a storage system means handling information like where data is logically stored. The metadata.Metadata grows along with the data itself and processing it can become one of the limiting factors to system scalability.

However, legacy storage architectures are proving to be inadequate when it comes to supporting the kinds of data growth that's all too common in today's data center environments. Object Storage Systems are now stepping in to address this issue.

In order to explain how Object Storage deals the scalability problem, first we will more fully explain the importance of metadata in the scalability problem.

Scalability and Metadata

In order for a storage system to scale it must hold more raw capacity, of course, but it also needs to process more metadata. This is the information that controls where the data is stored and how it's accessed. Metadata also supports other functions, such as data protection and storage system features. Clearly, the system's ability to process metadata can directly impact how large it scales.

The amount of processing power a system has can govern how much metadata it can administer, similar to the way a larger storage controller can handle more data I/O. This is why scalable controller architectures can support greater overall capacity. In addition, the number of background processes the controller has to perform on that metadata for things like data protection also impact how much CPU power is left for storage growth.

But efficiency also plays a part in metadata processing, specifically the way metadata is organized and handled. This can be especially true with network attached storage (NAS) systems, some of which require an unusual amount of system resources to be reserved in order to process snapshots and other OS overhead.

Object-based storage architectures offer some significant advantages when it comes to metadata processing that enable these systems to scale much larger than traditional storage architectures.

Let's see how traditional storage systems have handled growth.

How Traditional Storage Scales

Traditional storage arrays have a central controller which handles data I/O, data protection (RAID and replication, typically) and supports storage features like snapshots. This kind of physical architecture adds disk drives behind the storage controller, which in most cases has a fixed amount of processing power, and 'scales up' as it grows. This can create a bottleneck in I/O processing, limiting the amount of capacity that the system can support.

Scale-out storage architectures addressed this physical limitation, to some extent, by spreading the controller function out into multiple modules, as they did with the storage capacity. This enabled them to scale processing power as disk drives were added, effectively increasing their overall capacity.

But, like scale-up storage, they use a logical data architecture that stores data in blocks on the disk drives and organizes metadata in a hierarchy, somewhat like a file system tree. As the capacity grows the amount of data blocks grows and more importantly, the amount of metadata required to handle those blocks also grows.

When these systems get very large (in capacity and number of files), as they can in cloud applications for example, the way they organize metadata can impact how well they scale. In the traditional storage systems, the inefficiency of their hierarchical structure can become a limiting factor to growth.

How Object Storage Scales

As explained earlier, object storage systems, put data into objects, discrete 'containers' that each have a unique identifier, called an object ID (OID). Each object is accessed by its OID which can then provide access to all the data within that object. The result is a flatter, simpler metadata structure, more like an index than a hierarchy. This reduces the processing steps required to access data and improves the efficiency of the storage system. This efficiency in turn allows object storage systems to grow much larger than traditional block based storage systems.

But there are differences between object-based data architectures as well. Most object storage systems use a centralized metadata store which runs all metadata processing through a common module or node. Like scale-up physical architectures, which can bottleneck data I/O processing, centralizing metadata can bottleneck the metadata processing, and like legacy storage controllers, create a significant limitation on scalability.

Object storage systems like Cleversafe use a decentralized metadata architecture which allows the processing function to scale along with the amount of metadata that a growing storage system would generate. They actually store metadata as separate objects, like they do regular data, and distribute them into the object storage space as well. This potentially adds a step to the process for fetching data if metadata is used to locate the OID, but for most web-scale environments it's a small trade off to get near limitless object-count scalability. By distributing the metadata index across thousands of storage nodes they can produce extremely traversing and location of objects within the system.

2.9.2.3 Economics

Some object storage systems use RAID and replication to maintain data safety and integrity, a process that can generate extra copies of data. But even with those extra copies object storage's ability to support scale-out configurations that use commodity hardware can still make these systems lower in cost than traditional, scale-up storage. However, when erasure coding is included, object-based architectures can be used to create even lower-cost disk storage infrastructures. Erasure coding as a disk-level data protection mechanism, generates much less redundant data than RAID. In a typical implementations this efficiency can reduce the amount of storage capacity needed by 2-3x, driving significant cost savings.

2.9.2.4 Data Resiliency

Object storage with RAID and replication can assure there are always a minimum number of copies of each object in a particular data set and that copies are maintained in a different location in order to provide disaster recovery protection. Object storage with erasure coding can provide even better data protection by enabling the system to sustain many more data failures and recover much more quickly than a comparable RAID system. And, when objects are distributed geographically, object storage can provide a disaster recovery capability as well, one that's more efficient than simply making replicated copies.

2.9.2.5 Advanced functionality

An object storage system's REST interface allows it to be accessed directly by applications that are also REST-enabled, bypassing traditional storage protocols. REST makes web connectivity easier. Its flexible metadata structure also enables object storage systems to support advanced features like tagging, security, data tiering, encryption, etc.

It's clear that object-based architectures provide some unique capabilities that can be used to create solutions to address some of the storage challenges facing companies today. Handling Big Data, large file content archives, long term data retention and cloud-based storage are some of these challenges that these systems are being designed for.

2.10 HYBRID STORAGE

A method combining different storage systems, intended for companies or users who can not or do not want to completely change their traditional storage system. Storage architecture that provides block-based storage, files storage and object storage on the same network.

Currently there are several proprietary solutions for hybrid storage, but in this document we will focus on open source solutions such as the Ceph system, offered by RedHat.

Before we define and explain thoroughly the Ceph storage system, we must understand that,although Ceph can be used as a combination of the three storage systems discussed above, this will not always be the best option to choose, it will always depend on the case in which we want to use it.

For example, if we start from a base where we have hardware and want to take advantage and build up our infrastructure as we go needing, Ceph is a good option when we make heavy use of storage. But if we have to set up a new infrastructure and do not have the hardware or our company's medium / small, and we will not need more storage than the basic / minimum, maybe the traditional alternatives can result cheaper and easier to implement.

2.10.1 CEPH

Figure 2.22: Ceph logo.

2.10.1.1 Introduction

As we explained throughout this document, enterprise storage requirements have grown explosively over the last few years. Research has shown that data in large enterprises is growing at a rate of 40 to 60 percent annually, and many companies are doubling their data footprint each year. The international Data Corporation (IDC) analysts estimated that there were 54.4 exabytes of total digital data worldwide in the year 2000. By 2007, this reached 295 exabytes, and by the end of 2014, has reached 8,591 exabytes worldwide.

Traditional storage systems based on RAID or LVM and connected to servers via SAN or NAS, have solved storage problems since '80s. These monolithic structures work fine in small and medium environments, because in these scenes it's not possible to reach their inner limits, but because of the incredible growth of managed data, like Big Data or Cloud Computing, in several situations they are becoming unsuitable.

Firstly, when data grow, the traditional storage systems have computing capacity problems, because these systems can only be updated by adding new disks, and only in a few models it can also add new controllers. Also, the capacity growth has a side effect in a reduction in performances, since a single controller has a well-defined maximum computing capacity, and with it it has to manage a larger number of data.

Secondly, with data growth, RAID is becoming an inefficient redundancy technology. These is because, first of all, redundancy is achieved by sacrificing disk space. As an example, if we have a RAID 10 using a 1 TB disks, a stripe of 20 TB is wasting 10 TB, but if we think in more data, for example a storage of 1 PB is wasting 500 TB of space only for redundancy. If we choose a RAID 5 or RAID 6, we reduce the disk waste, but on the other side we are also lowering the performances, because write penalty is higher on these kind of RAID. With increasing of data, we will also be penalized with the time we will need to rebuild data because during these time RAID is running in a degraded state, and this implies additional impacts in performance.

So, worldwide storage demands systems able to work with this large amount of data, be unified, distributed and reliable, with high performance and most importantly, massively scalable up to the exabyte level and beyond. One of the available solutions of the worldwide growing data explosion is an open source distributed storage system called Ceph.

The reason why Ceph is emerging as a solution to work with this huge amount of data is its lively community and users who truly believe in the power of Ceph. As data generation is a never-ending process because we can not stop data generation, we need to bridge the gap between data generation and data storage.

As the Linux community have seen that Ceph is the right solution to today's and tomorrow's data storage needs because it's unified, distributed, cost-effective, and scalable, on March, 2010, the open source community had support for Ceph in the mainline Linux kernel. This has been a milestone for Ceph as there is no other competitor to join it there.

2.10.1.2 Overview

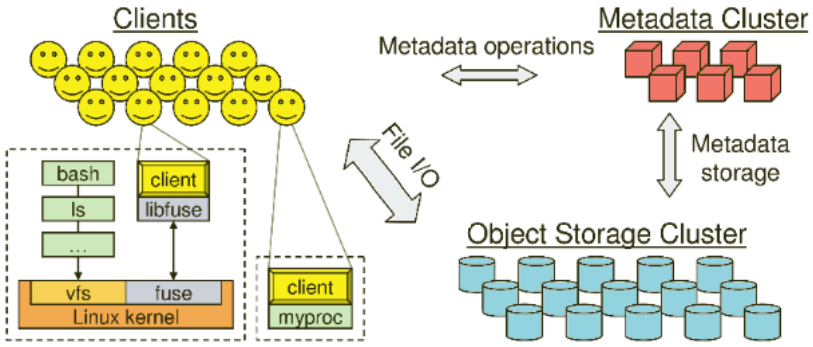


Figure 2.23: Ceph Architecture.

2.10.1.2.1 Definition

Ceph is an open source distributed storage system with high-performance and scalability, built on top of commodity components, demanding reliability to the software layer. We can say that Ceph is a scale out Software Defined Object Storage built on commodity hardware. Ceph storage software it's one of the pillars of the OpenStack project, which involved developers around the world working on different components of open source cloud platforms.

Ceph was developed to make possible the standardized storage of Object, Block and Files with a distributed x86 cluster. Ceph has the ability to manage vast amounts of data and delivers extraordinary scalability–thousands of clients

accessing petabytes to exabytes of data. A Ceph Node leverages commodity hardware and intelligent daemons, and a Ceph Storage Cluster accommodates large numbers of nodes, which communicate with each other to replicate and redistribute data dynamically.

2.10.1.2.2 Ceph name

The term Ceph is a common nickname given to pet octopuses. It can be considered as a short form for Cephalopod, which belongs to the mollusk family of marine animals. Ceph has octopuses as its mascot, which represents Ceph's parallel behavior to octopuses.

The word Inktank (the company behind Ceph) is related to cephalopods, because sometimes fishermen refer to cephalopods as inkfish due to their ability to squirt ink. This explains how cephalopods (Ceph) have some relation with inkfish (Inktank). So, Ceph and Inktank have a lot of things in common.

2.10.1.2.3 Ceph History

Sage Weil, Inktank founder and CTO, built the first working prototype of the company's open-source Ceph storage software as a computer science graduate student in UCSC's Baskin School of Engineering in 2003. The initial project prototype was the Ceph filesystem, written in approximately 40,000 lines of C++ code, which was made open source in 2006 under a Lesser GNU Public License (LGPL) to serve as a reference implementation and research platform. After earning his Ph.D. in 2007, Weil continued working to develop Ceph into a robust, massively scalable cloud storage platform.

In late 2007, Ceph was getting mature and then, DreamHost, a Los-Angeles-based web hosting and domain registrar company entered the picture. DreamHost incubated Ceph from 2007 to 2011. During this period, Ceph was gaining its shape; the existing components were made more stable and reliable, various new features were implemented, and future roadmaps were designed. During this time, several developers as Yehuda Sadeh, Weinraub, Gregory Farnum, Josh Durgin, Samuel Just, Wido den Hollander, and Loïc Dachary, who joined the Ceph bandwagon, started contributing to the Ceph project.

Later, in April 2012, Sage Weil founded Inktank comany. Inktank was formed to enable the widespread adoption of Ceph's professional services and support. Inktank is the company behind Ceph whose main objective is to provide expertise, processes, tools, and support to their enterprise-subscription customers, enabling them to effectively adopt and manage Ceph storage systems.

Finally, on April 30, 2014, Red Hat, Inc. an American multinational software company providing open source solutions, agreed to acquire Inktank for approximately $175 million in cash. Some of the customers of Inktank include Cisco, CERN, and Deutsche Telekom, and its partners include Dell and Alcatel-Lucent, all of which will now become the customers and partners of Red Hat for Ceph's software-defined storage solution.

2.10.1.3 Ceph technical terms

Before we explain all Ceph architecture and which products offers, we first explain some new concepts that Ceph introduce:

RADOS

Ceph Storage Cluster is based on an open source object storage service called Reliable Automatic Distributed Object Store (RADOS).

RADOS has the ability to scale to thousands of hardware devices by making use of management software that runs on each of the individual storage nodes. The software provides storage features such as thin provisioning, snapshots

and replication. RADOS distributes objects across the storage cluster and replicates objects for fault tolerance.

CRUSH
The Controlled Replication Under Scalable Hashing algorithm determines how the data is replicated and mapped to the individual nodes. CRUSH ensures that data is evenly distributed across the cluster and that all cluster nodes are able to retrieve data quickly without any centralized bottlenecks.

The algorithm determines how to store and retrieve data by computing data storage locations. CRUSH empowers Ceph clients to communicate with OSDs (explained below) directly rather than through a centralized server or broker. With an algorithmically determined method of storing and retrieving data, Ceph avoids a single point of failure, a performance bottleneck, and a physical limit to its scalability.

CRUSH requires a map of our cluster (contain a list of OSDs) , and uses the CRUSH map to pseudo-randomly store and retrieve data in OSDs with a uniform distribution of data across the cluster.

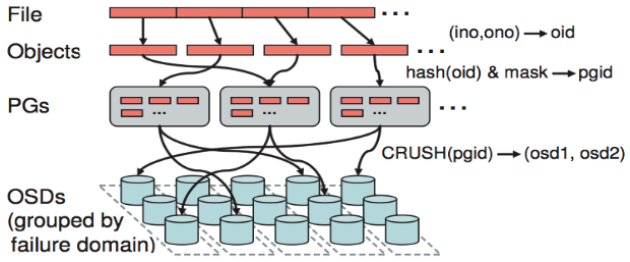


Figure 2.24: Crush Algorithm.

Ceph Node
Any single machine or server in a Ceph System.

Ceph OSDs
A Ceph OSD Daemon stores data as objects on a storage node, handles data replication, recovery, backfilling, rebalancing, and provides some monitoring information to Ceph Monitors by checking other Ceph OSD Daemons for a heartbeat. So, as Ceph OSDs run the RADOS service, calculate data placement with CRUSH and maintain their own copy of the cluster map they should have a reasonable amount of processing power. A Ceph Storage Cluster requires at least two Ceph OSD Daemons to achieve an active + clean state when the cluster makes two copies of our data (by default).

Ceph Monitor Maintains a master copy of the cluster map (cluster state), including the monitor map, the OSD map, the PG map, and the CRUSH map. Ceph maintains a history (called an "epoch") of each state change in the Ceph Monitors, Ceph OSD Daemons and PGs. They are not CPU intensive.

Ceph Metadata Server
A MDS stores metadata on behalf of the Ceph Filesystem (i.e., Ceph Block Devices and Ceph Object Storage do not use MDS). Ceph Metadata Servers make it feasible for Portable Operating System Interface (POSIX) file system users to execute basic commands like ls, find, etc. without placing an enormous burden on the Ceph Storage Cluster.

Placement Group (PG)
Aggregates objects within a pool (logical partitions for storing objects) because tracking object placement and object metadata on a per-object basis is computationally expensive.

Ceph Cluster Map

The set of maps comprising the monitor map, OSD map, PG map, MDS map and CRUSH map.

2.10.1.4 Ceph portfolio

Ceph is an enterprise-ready storage system that offers multiple products and gives support to a wide range of protocols and accessibility methods. The unified Ceph storage system supports **Block**, **File**, and **Object storage**, and all these storage systems are deployments of the Ceph flagship product, the **Ceph Storage Cluster**.

However, Ceph File System (CephFS) currently is under development because lacks a robust 'fsck' check and repair function, but Ceph Block Storage and Ceph Object Storage are ready and recommended for production usage.

All Ceph Block, Ceph File and Ceph Object read data from and write data to the Ceph Storage Cluster, so they need to have access to a running Ceph Storage Cluster so as to run properly.

2.10.1.5 Ceph Architecture

As explained in the Definition section, Ceph is made for big company's IT infrastructures as is able to manage vast amounts of data and deliver extraordinary scalability–thousands of clients accessing petabytes to exabytes of data. But not only these, Ceph also (and uniquely) delivers object, block and file storage in one unified System, being highly reliable, easy to manage and free.

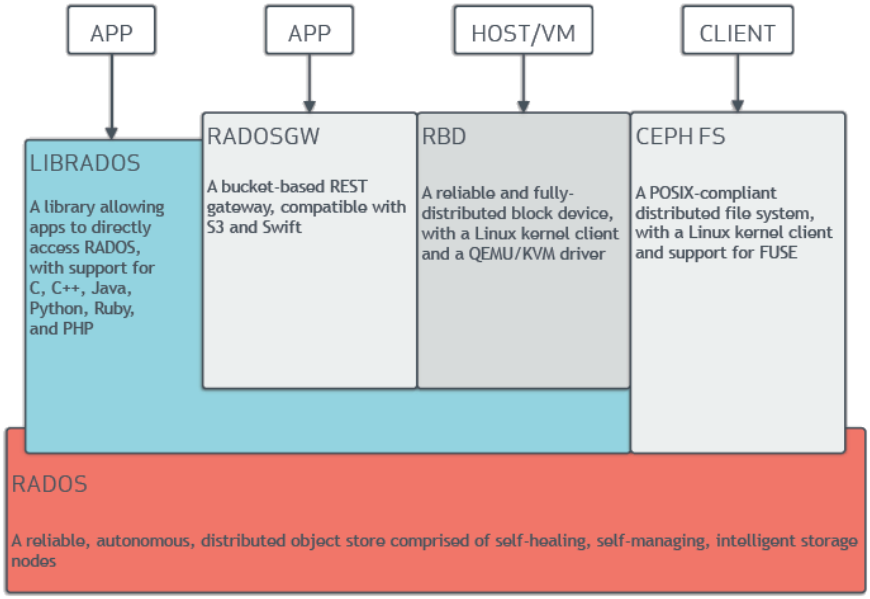


Figure 2.25: Ceph Architecture.

But behind the flagship products we have many components that are important in the functioning of Ceph. Like the Ceph Nodes, who leverages commodity hardware and intelligent daemons, and the Ceph Storage Cluster, that accommodates large numbers of nodes, which communicate with each other to replicate and redistribute data dynamically.

In these section we will see some of these components and we will explain the role that they have throughout the Ceph system.

2.10.1.5.1 Ceph storage architecture

In these section we will explain the different components and several software daemons (OSD and MON) that take care of the different Ceph Storage Cluster functionalities. One of the things that keeps storage costs down in comparison with an enterprise, proprietary black box storage system is that each of these daemons is separated from the others.

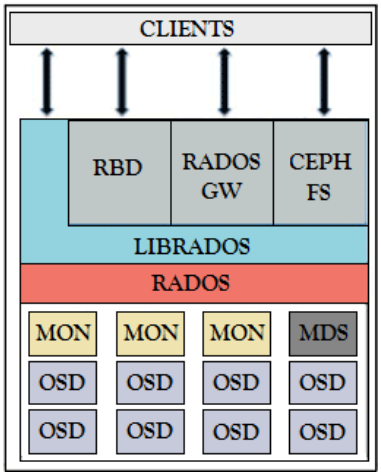


Figure 2.26: Ceph storage architecture.

Ceph Object Gateway (RGW): This component is formally known as RADOS gateway, and its function is to provide a RESTful API interface compatible with Amazon S3 (**Simple Storage Service**) and OpenStack Object Storege API (**Swift**). Also supports the multitenancy and OpenStack Keystone authentication services.

Ceph File System (CephFS): Offers a POSIX-compliant, distributed filesystem of any size. Relies on Ceph MDS to keep track of file hierarchy, that is, metadata. CephFS uses metadata to store file attributes such as the file owner, created date, last modified date, and so forth. But, at the moment, CephFS is not production ready, only a candidate for point-of-care tests. CephFS uses metadata to store file attributes such as the file owner, created date, last modified date, and so forth.

However, Ceph File System (CephFS) currently is under development because lacks a robust 'fsck' check and repair function, but Ceph Block Storage and Ceph Object Storage are ready and recommended for production usage.

Ceph Object Storage Device(OSD): When our application make a write operation in to our Ceph cluster, the data gets stored in OSDs in an object form. This component is the only one of our Ceph cluster where actual user data is stored and the same data is retrieved when a client makes a read operation. In a professional environment, one OSD daemon is tied to one physical disk in our cluster, but in the **Ceph Storage Cluster with LXC** section we will see that, for testing purposes, we can also tie an OSD daemon to a directory.

Ceph monitors(MONs): Track the health of the entire cluster by keeping a map of the cluster state, which includes OSD, MON, PG, and CRUSH maps. All the cluster nodes report to monitor nodes and share information about every change in their state. The monitor does not store actual data, only maintains a separate map of information for each component.

Ceph Metadata Server(MDS): Keeps the track of file hierarchy and stores metadata only for CephFS because the Ceph block devices and RADOS gateways do not require metadata. MDS does not serve data directly to clients, thus removing a single point of failure from the system.

Once we have explained all the Ceph storage components, lets see how they work together.

Summary: Ceph Storage Clusters are dynamic–like a living organism. Whereas, many storage appliances do not fully utilize the CPU and RAM of a typical commodity server, Ceph does. From heartbeats, to peering, to rebalancing the cluster or recovering from faults, Ceph offloads work from clients (and from a centralized gateway which doesn't exist in the Ceph architecture) and uses the computing power of the OSDs to perform the work. Let's see how some of these components work.

2.10.1.5.1.1 Ceph storage working mode

As we explained earlier, Ceph provides an infinitely scalable Ceph Storage Cluster based upon RADOS, and consists of two types of daemons: **MONs** and **OSDs**.

Storage cluster clients and each Ceph OSD Daemon use the **CRUSH** algorithm to efficiently compute information about data location, instead of having to depend on a central lookup table. Ceph's high-level features include providing a native interface to the Ceph Storage Cluster via **librados**, and a number of service interfaces built on top of librados.

Storing Data

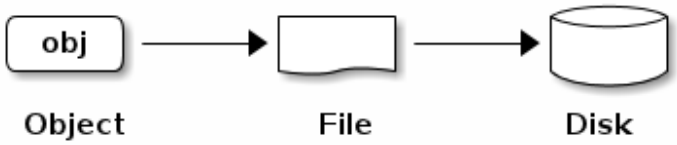


Figure 2.27: Ceph storing data.

The Ceph Storage Cluster receives data from **Ceph Clients**–whether it comes through a Ceph Block Device, Ceph Object Storage, the Ceph Filesystem or a custom implementation we create using librados–and it stores the data as objects. Each object corresponds to a file in a filesystem, which is stored on an Object Storage Device. Ceph OSD Daemons handle the read/write operations on the storage disks.

Ceph OSD Daemons store all data as objects in a flat namespace (e.g., no hierarchy of directories). An object has an identifier (ID), which is unique across the entire cluster, not just the local filesystem,a binary data, and metadata consisting of a set of name/value pairs. The semantics are completely up to Ceph Clients.

Scalability and High Availability
Ceph eliminates the centralized gateway used in traditional architectures (e.g., a gateway), and enable clients to interact with Ceph OSD Daemons directly, increasing both performance and scalability. Ceph OSD Daemons create object replicas on other Ceph Nodes to ensure data safety and high availability. Ceph also uses a cluster of monitors to ensure high availability. To eliminate centralization, Ceph uses an algorithm called **CRUSH**.

CRUSH brief Introduction
Ceph Clients and Ceph OSD Daemons both use the CRUSH algorithm to efficiently compute information about object location, instead of having to depend on a central lookup table. CRUSH provides a better data management mechanism compared to older approaches, and enables massive scale by cleanly distributing the work to all the clients and OSD daemons in the cluster. CRUSH uses intelligent data replication to ensure resiliency, which is better suited to hyper-scale storage.

Cluster Map Ceph depends upon Ceph Clients and Ceph OSD Daemons having knowledge of the cluster topology, which is inclusive of 5 maps collectively referred to as the "Cluster Map": the **Monitor Map**, the **OSD Map**, the **PG Map**, the **Crush Map** and the **MDS Map**.

High Availability Monitors

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.For added reliability and fault tolerance, Ceph supports a cluster of monitors.

In a cluster of monitors, latency and other faults can cause one or more monitors to fall behind the current state of the cluster. For this reason, Ceph must have agreement among various monitor instances regarding the state of the cluster. Ceph always uses a majority of monitors (e.g., 1, 2:3, 3:5, 4:6, etc.) and the PAXOS algorithm to establish a consensus among the monitors about the current state of the cluster.

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

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

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.

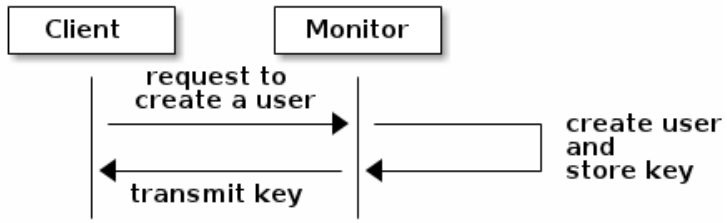


Figure 2.28: Cephx function.

To authenticate with the monitor, the client passes in the user name to the monitor, and the monitor generates a session key and encrypts it with the secret key associated to the user name. Then, the monitor transmits the encrypted ticket back to the client. The client then decrypts the payload with the shared secret key to retrieve the session key. The session key identifies the user for the current session. The client then requests a ticket on behalf of the user signed by the session key. The monitor generates a ticket, encrypts it with the user's secret key and transmits it back to the client. The client decrypts the ticket and uses it to sign requests to OSDs and metadata servers throughout the cluster.

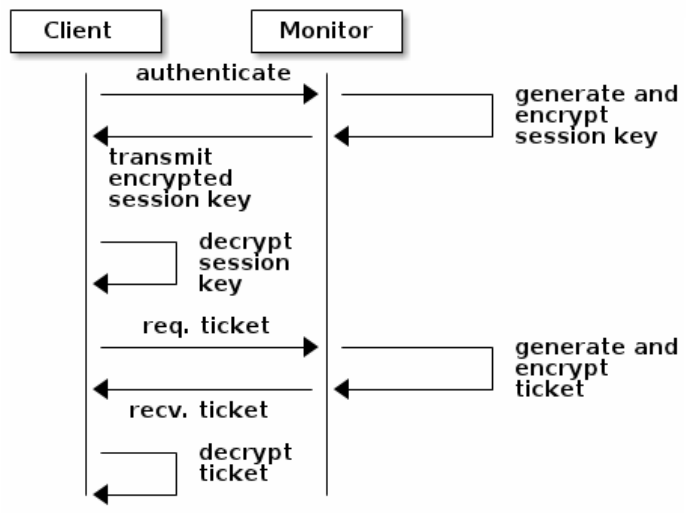


Figure 2.29: Cephx encrypted ticket.

The **cephx** protocol authenticates ongoing communications between the client machine and the Ceph servers. Each message sent between a client and server, subsequent to the initial authentication, is signed using a ticket that the monitors, OSDs and metadata servers can verify with their shared secret.

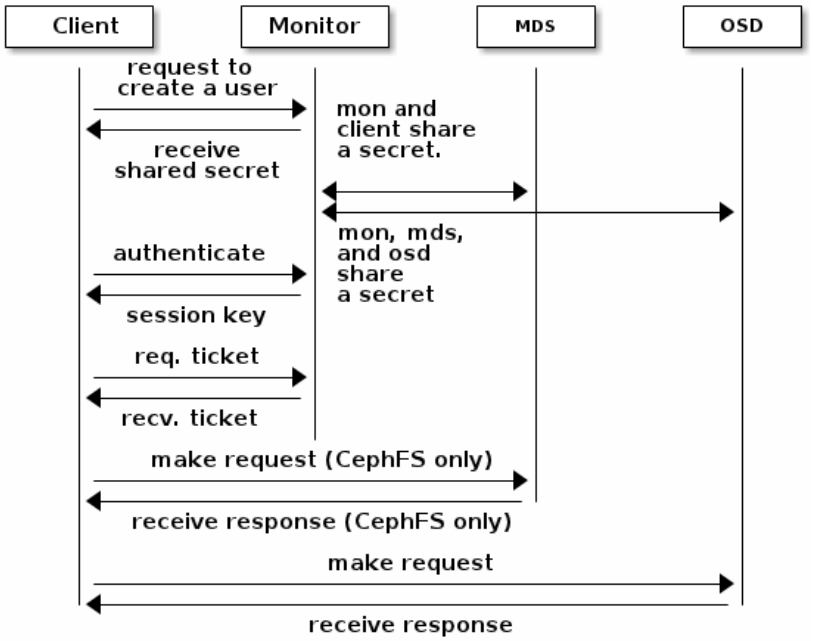


Figure 2.30: Cephx authentication.

The protection offered by this authentication is between the Ceph client and the Ceph server hosts. The authentication is not extended beyond the Ceph client. If the user accesses the Ceph client from a remote host, Ceph authentication is not applied to the connection between the user's host and the client host.

Smart Daemons Enable Hyperscale

In many clustered architectures, the primary purpose of cluster membership is so that a centralized interface knows which nodes it can access. Then the centralized interface provides services to the client through a double dispatch–which is a huge bottleneck at the petabyte-to-exabyte scale.

Ceph eliminates the bottleneck: Ceph's OSD Daemons AND Ceph Clients are cluster aware. Like Ceph clients, each Ceph OSD Daemon knows about other Ceph OSD Daemons in the cluster. This enables Ceph OSD Daemons to interact directly with other Ceph OSD Daemons and Ceph monitors. Additionally, it enables Ceph Clients to interact directly with Ceph OSD Daemons.

The ability of Ceph Clients, Ceph Monitors and Ceph OSD Daemons to interact with each other means that Ceph OSD Daemons can utilize the CPU and RAM of the Ceph nodes to easily perform tasks that would bog dawn a centralized server. The ability to leverage this computing power leads to several major benefits like that the OSDs can service Clients directly, Data Scrubbing or Replication.

So, as we can see in the following figure, the client writes the object to the identified placement group in the primary OSD. Then, the primary OSD with its own copy of the CRUSH map identifies the secondary and tertiary OSDs for replication purposes, and replicates the object to the appropriate placement groups in the secondary and tertiary OSDs (as many OSDs as additional replicas), and responds to the client once it has confirmed the object was stored successfully.

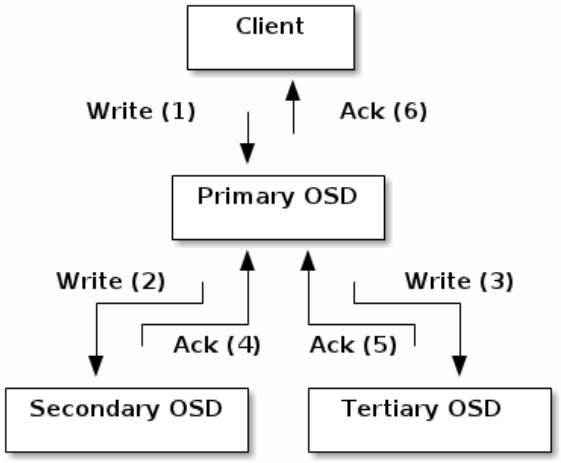


Figure 2.31: Client and daemons communication.

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

Dynamic Cluster Management

In previous **Scalability and High Availability** section, we explained how Ceph uses CRUSH, cluster awareness and intelligent daemons to scale and maintain high availability. Key to Ceph's design is the autonomous, self-healing, and intelligent Ceph OSD Daemon. Next we will take a deeper look at how CRUSH works to enable modern cloud storage infrastructures to place data, rebalance the cluster and recover from faults dynamically.

Pools

The Ceph storage system supports the notion of 'Pools', which are logical partitions for storing objects.

Ceph Clients retrieve a Cluster Map from a Ceph Monitor, and write objects to pools. The pool's size or number of replicas, the CRUSH ruleset and the number of placement groups determine how Ceph will place the data.

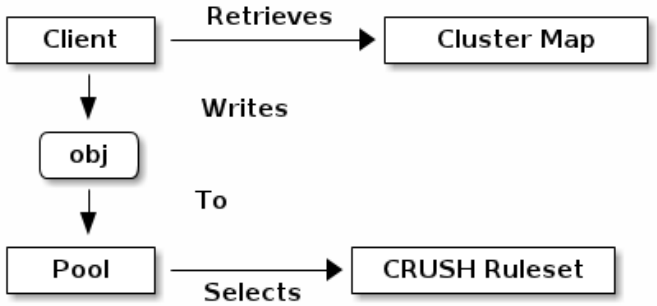


Figure 2.32: Client to pool.

Pools set at least the three parameters: Ownership/Access to Objects, Number of Placement Groups, and the CRUSH Ruleset to Use.

Mapping PGs to OSDs

Each pool has a number of placement groups. CRUSH maps PGs to OSDs dynamically. When a Ceph Client stores objects, CRUSH will map each object to a placement group.

Mapping objects to placement groups creates a layer of indirection between the Ceph OSD Daemon and the Ceph Client. The Ceph Storage Cluster must be able to grow (or shrink) and rebalance where it stores objects dynamically. If the Ceph Client "knew" which Ceph OSD Daemon had which object, that would create a tight coupling between the Ceph Client and the Ceph OSD Daemon. Instead, the CRUSH algorithm maps each object to a placement group and then maps each placement group to one or more Ceph OSD Daemons. This layer of indirection allows Ceph to rebalance dynamically when new Ceph OSD Daemons and the underlying OSD devices come online. The following diagram explains how CRUSH maps objects to placement groups, and placement groups to OSDs.

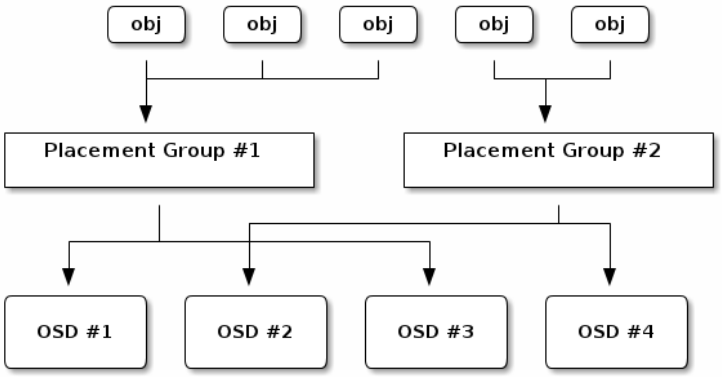


Figure 2.33: Crush to PG and PG to OSDs.

With a copy of the cluster map and the CRUSH algorithm, the client can compute exactly which OSD to use when reading or writing a particular object.

Rebalancing

When we add a Ceph OSD Daemon to a Ceph Storage Cluster, the cluster map gets updated with the new OSD. This changes the cluster map. Consequently, it changes object placement, because it changes an input for the calculations. The following diagram shows the rebalancing process (albeit rather crudely, since it is substantially less impactful with large clusters) where some, but not all of the PGs migrate from existing OSDs (OSD 1, and OSD 2) to the new OSD (OSD 3). Even when rebalancing, CRUSH is stable. Many of the placement groups remain in their

original configuration, and each OSD gets some added capacity, so there are no load spikes on the new OSD after rebalancing is complete.

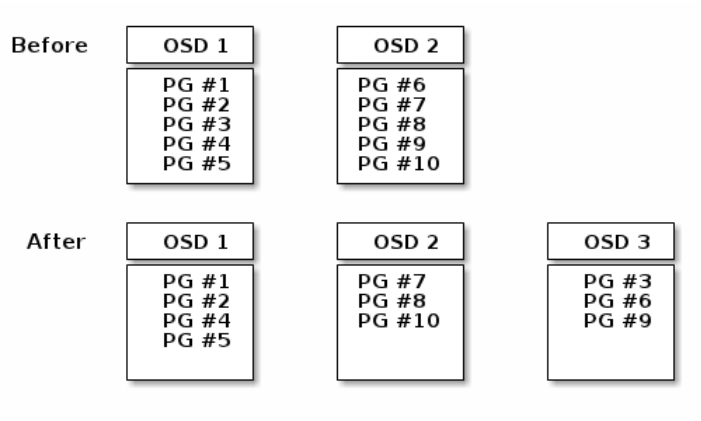


Figure 2.34: Rebalancing process.

Data Consistency

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

Extending Ceph

We can extend Ceph by creating shared object classes called 'Ceph Classes'. Ceph loads **.so** classes stored in the **osd class dir** directory dynamically (i.e., **$libdir/rados-classes** by default). When we implement a class, we can create new object methods that have the ability to call the native methods in the Ceph Object Store, or other class methods we incorporate via libraries or create ourself.

On writes, Ceph Classes can call native or class methods, perform any series of operations on the inbound data and generate a resulting write transaction that Ceph will apply atomically.

On reads, Ceph Classes can call native or class methods, perform any series of operations on the outbound data and return the data to the client.

2.10.1.5.1.2 Ceph protocol

Ceph Clients use the native protocol for interacting with the Ceph Storage Cluster. Ceph packages this functionality into the librados library so that we can create our own custom Ceph Clients. The following diagram depicts the basic architecture.

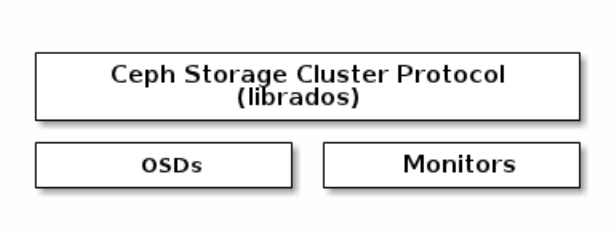


Figure 2.35: Ceph protocol.

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.

2.10.1.5.1.3 Ceph Clients

Ceph Clients include a number of service interfaces. These include:

- **Block Devices**: The Ceph Block Device (RBD) service provides resizable, thin-provisioned block devices with snapshotting and cloning. Ceph stripes a block device across the cluster for high performance. Ceph supports both kernel objects (KO) and a QEMU hypervisor that uses **librbd** directly–avoiding the kernel object overhead for virtualized systems.
- **Object Storage**: The Ceph Object Storage (a.k.a., RGW) service provides RESTful APIs with interfaces that are compatible with Amazon S3 and OpenStack Swift.
- **Filesystem**: The Ceph Filesystem (CephFS) service provides a POSIX compliant filesystem usable with **mount** or as a filesytem in user space (FUSE).

Ceph can run additional instances of OSDs, MDSs, and monitors for scalability and high availability. The following diagram depicts the high-level architecture.

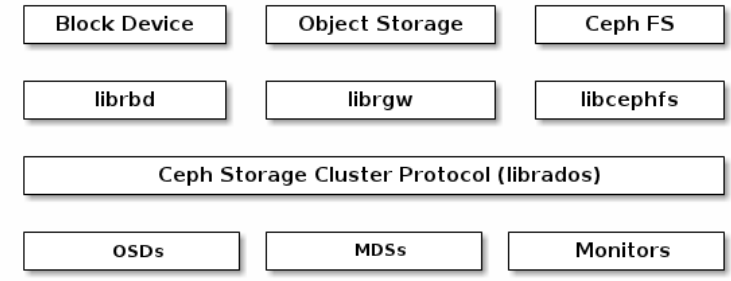


Figure 2.36: Ceph Clients.

CEPH OBJECT STORAGE

As we explained in **Ceph Storage components** section, the Ceph Object Storage daemon, **radosgw**, is a FastCGI service that provides a RESTful HTTP API to store objects and metadata. It layers on top of the Ceph Storage Cluster with its own data formats, and maintains its own user database, authentication, and access control. The RADOS Gateway uses a unified namespace, which means we can use either the OpenStack Swift-compatible API or the Amazon

S3-compatible API. For example, we can write data using the S3-compatible API with one application and then read data using the Swift-compatible API with another application.

CEPH BLOCK DEVICE STORAGE

A Ceph Block Device stripes a block device image over multiple objects in the Ceph Storage Cluster, where each object gets mapped to a placement group and distributed, and the placement groups are spread across separate **ceph-osd** daemons throughout the cluster. It's important to say that striping allows RBD block devices to perform better than a single server could.

Thin-provisioned snapshottable Ceph Block Devices are an attractive option for virtualization and cloud computing. In virtual machine scenarios, people typically deploy a Ceph Block Device with the **rbd** network storage driver in Qemu/KVM, where the host machine uses **librbd** to provide a block device service to the guest. Many cloud computing stacks use **libvirt** to integrate with hypervisors. We can use thin-provisioned Ceph Block Devices with Qemu and **libvirt** to support OpenStack and CloudStack among other solutions.

While Ceph do not provide **librbd** support with other hypervisors at this time, we may also use Ceph Block Device kernel objects to provide a block device to a client. Other virtualization technologies such as Xen can access the Ceph Block Device kernel object(s). This is done with the command-line tool **rbd**.

CEPH FILESYSTEM

The Ceph Filesystem (Ceph FS) provides a POSIX-compliant filesystem as a service that is layered on top of the object-based Ceph Storage Cluster. Ceph FS files get mapped to objects that Ceph stores in the Ceph Storage Cluster. Ceph Clients mount a CephFS filesystem as a kernel object or as a Filesystem in User Space (FUSE).

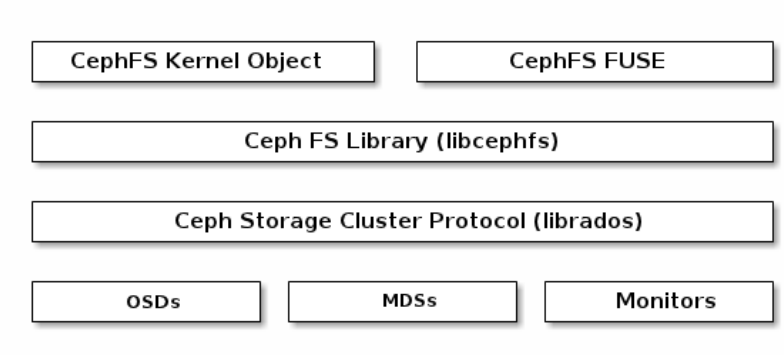


Figure 2.37: Ceph Filesystem.

The Ceph Filesystem service includes the Ceph Metadata Server (MDS) deployed with the Ceph Storage cluster. The purpose of the MDS is to store all the filesystem metadata (directories, file ownership, access modes, etc) in high-availability Ceph Metadata Servers where the metadata resides in memory. The reason for the MDS (a daemon called **ceph-mds**) is that simple filesystem operations like listing a directory or changing a directory (**ls, cd**) would tax the Ceph OSD Daemons unnecessarily. So separating the metadata from the data means that the Ceph Filesystem can provide high performance services without taxing the Ceph Storage Cluster.

Ceph FS separates the metadata from the data, storing the metadata in the MDS, and storing the file data in one or more objects in the Ceph Storage Cluster. The Ceph filesystem aims for POSIX compatibility. **ceph-mds** can run as a single process, or it can be distributed out to multiple physical machines, either for high availability or for scalability:

- **High Availability**: The extra **ceph-mds** instances can be standby, ready to take over the duties of any failed **ceph-mds** that was active. This is easy because all the data, including the journal, is stored on RADOS. The transition is triggered automatically by **ceph-mon**.
- **Scalability**: Multiple **ceph-mds** instances can be active, and they will split the directory tree into subtrees (and shards of a single busy directory), effectively balancing the load amongst all active servers.

2.10.1.6 Ceph Storage Cluster

Ceph Storage Cluster is the most important product of Ceph systems because all the other products are deployments of it. It is used to provide Object, File and Block storage over a network. So, the Ceph Object Storage, Ceph Block Devices and Ceph Filesystem (in progress) read data from and write data to the Ceph Storage Cluster.

The Ceph Cluster is based on the open source object storage service called RADOS, and it works running two types of daemons: the Ceph OSD Daemon (OSD), that stores data as objects on a storage node; and the Ceph Monitor (MON), that maintains a master copy of the cluster map. A Ceph Cluster can run containing thousands of OSDs but, for proper operation in a simulated scene, a Ceph minimal Storage Cluster will have at least one Ceph Monitor and two Ceph OSD Daemons for data replication because we want a lightweight monitoring service, but in a production environment, for a totally healthy cluster, we will need at least 3 OSD nodes and at least 3 Monitor nodes.

In the following sections we will configure our Ceph Storage Cluster using three different virtualization systems as examples: **Virtualbox**, **Linux Containers** and **Docker**, all three using **Ubuntu 14.04 LTS (Trusty Tahr)** release with **linux-3.16.0** kernel. In all three cases we will implement a but in all three cases we will need to meet certain common required settings as we will see next.

As we want to deploy a lightweight cluster in a quickly way, in all three examples we will use a deployment tool called **ceph-deploy**, that will help us to define the cluster and bootstrap a monitor.

2.10.1.6.1 Ceph-deploy tool

With this tool we can get Ceph up and running quickly with sensible initial configuration settings relying only upon SSH access to the servers, sudo, and some Python. With ceph-deploy we can develop scripts to install Ceph packages on remote hosts, create a cluster, add monitors, gather (or forget) keys, add OSDs and metadata servers, configure admin hosts, and tear down the clusters. Ceph deploy runs in our workstation, and does not require servers, databases, or any other tools. For other advanced users who want a fine-control over security settings, partitions or directory locations there are other tools such as Juju, Puppet, Chef or Crowbar.

With the ceph-deploy tool we will build our 3-node Ceph Storage Cluster as in the next figure:

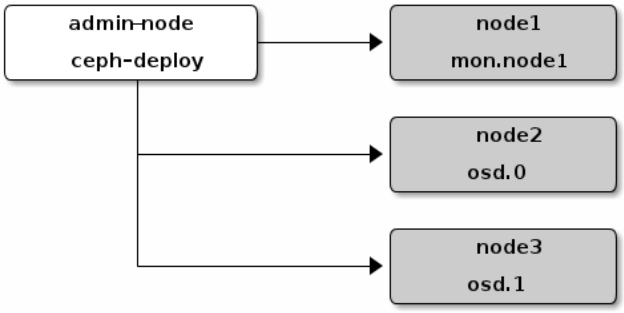


Figure 2.38: Cluster description.

From the above scheme we will use:

- **ceph-node1**:node in which we will install the ceph-deploy tool and from which we will deploy Ceph to the other nodes, and also will monitor the cluster.
- **ceph-node2**:node with the OSD module.
- **ceph-node3**:node with the OSD module.

Before we check and configure all the software requirements, we will explain the OS and Hardware recommendations that we will need to meet so as to deploy our Ceph Cluster.

2.10.1.6.2 OS recommendation

It is recommended to deploy Ceph on newer and long-term support releases of Linux.

LINUX KERNEL

It is recommended to use the v3.16.3 or later. If we use **brtfs**, it is recommended using a recent Linux kernel, such as v3.14 or later.

PLATFORMS

Ceph supports different releases depending of the Linux distribution that we are using:

- **FIREFLY (0.80)**:

  Distros:

  - **CentOS**:Release 6 and 7 with kernel linux-2.6.32 and linux-3.10.0 respectively.
  - **Debian**:Release 6.0 (Squeeze) and 7.0 (Wheezy) with kernel linux-2.6.32 and linux-3.2.0 respectively.
  - **Fedora**:Release 19 (Schrödinger's Cat) and 20 (Heisenbug) with kernel linux-3.10.0 and linux-3.14.0 respectively.
  - **RHEL**:Release 6 and 7 with kernel linux-2.6.32 and linux-3.10.0 respectively.
  - **Ubuntu**:Release 12.04 (Precise Pangolin) and 14.04 (Trusty Tahr) with kernel linux-3.2.0 linux-3.13.0 respectively.

- **Dumpling (0.67)**:

  Distros:

  - **CentOS**:Release 6.3 with kernel linux-2.6.32.
  - **Debian**:Release 6.0 (Squeeze) and 7.0 (Wheezy) with kernel linux-2.6.32 and linux-3.2.0 respectively.
  - **Fedora**:Release 18 (Spherical Cow) and 19 (Schrödinger's Cat) with kernel linux-3.6.0 and linux-3.10.0 respectively.
  - **OpenSuse**:Release 12.2 with kernel linux-3.4.0.
  - **RHEL**:Release 6.3 with kernel linux-2.6.32.
  - **Ubuntu**:Release 12.04 (Precise Pangolin), 12.10 (Quantal Quetzal) and 13.04 (Raring Ringtail) with kernel linux-3.2.0, linux-3.5.4 and linux-3.8.5 respectively.

- **Emperor (0.72), Argonaut (0.48), Bobtail (0.56), and Cuttlefish (0.61)**:

  The Ceph Emperor, Argonaut, Bobtail, and Cuttlefish releases are no longer supported, and users should update to the latest stable release (Dumpling (0.67) or Firefly (0.80)). (version 0.80).

In our case, we will use the Ubuntu 14.04 LTS (Trusty Tahr) release with kernel linux-3.16.0, so we will install the Firefly (0.80) Ceph release.

2.10.1.6.3 Hardware requirements

Ceph was designed to run on commodity hardware, which makes building and maintaining petabyte-scale data clusters economically feasible. When planning out our cluster hardware, we will need to balance a number of considerations, including failure domains and potential performance issues. Hardware planning should include distributing Ceph daemons and other processes that use Ceph across many hosts. Generally, it is recommended running Ceph daemons of a specific type on a host configured for that type of daemon and using other hosts for processes that utilize our data cluster (e.g., OpenStack, CloudStack, etc).

- **CPU**

  Ceph metadata servers dynamically redistribute their load, which is CPU intensive. So our metadata servers should have significant processing power (e.g., quad core or better CPUs). Ceph OSDs run the RADOS service, calculate data placement with CRUSH, replicate data, and maintain their own copy of the cluster map. Therefore, OSDs should have a reasonable amount of processing power (e.g., dual core processors). Monitors simply maintain a master copy of the cluster map, so they are not CPU intensive. We must also consider whether the host machine will run CPU-intensive processes in addition to Ceph daemons. For example, if our hosts will run computing VMs (e.g., OpenStack Nova), we will need to ensure that these other processes leave sufficient processing power for Ceph daemons. It is recommended running additional CPU-intensive processes on separate hosts.

- **RAM**

  Metadata servers and monitors must be capable of serving their data quickly, so they should have plenty of RAM (e.g., 1GB of RAM per daemon instance). OSDs do not require as much RAM for regular operations (e.g., 500MB of RAM per daemon instance); however, during recovery they need significantly more RAM (e.g., 1GB per 1TB of storage per daemon). Generally, more RAM is better.

- **Data Storage**

  We must plan our data storage configuration carefully. There are significant cost and performance tradeoffs to consider when planning for data storage. Simultaneous OS operations, and simultaneous request for read and write operations from multiple daemons against a single drive can slow performance considerably. There are also file system limitations to consider: btrfs is not quite stable enough for production, but it has the ability to journal and write data simultaneously, whereas XFS and ext4 do not.

In our case, as in all three technologies that we will use to implement the Ceph Cluster we work in sandbox environments imitating a fully functioning scenery, we will see that we have no need of so much computing power or storage space.

2.10.1.6.4 Prior Software configurations

The configurations explained in this section will be mentioned in the three examples of Ceph Cluster implementation that we will see in the following sections, because these are a common software prerequisites so as to implement a totally healthy Ceph Storage Cluster.

Next we will explain the steps to be followed depending on the distribution with which we work so as to correctly configure our nodes before we deploy the cluster.

2.10.1.6.4.1 Ceph-deploy setup

We add Ceph repositories to our ceph-deploy node called ceph-node1 so as to can install the **ceph-deploy** tool next.

ADVANCED PACKAGE TOOL (APT)

For **Debian** and **Ubuntu** distributions we must perform the next steps:

- 1. Add the release key:

```
wget -q -O- 'http://eu.ceph.com/git/?p=ceph.git;a=blob_plain;f=keys/release.asc' |
sudo apt-key add -
```

- 2. Add the Ceph packages to our repository. Replace **{ceph-stable-release}** with a stable Ceph release (e.g., cuttlefish, dumpling, emperor, firefly, etc.).

```
echo deb http://eu.ceph.com/debian-{ceph-stable-release}/ $(lsb_release -sc) main | sudo
tee /etc/apt/sources.list.d/ceph.list
```

For example in our case, that we use Ubuntu 14.04 release, we must introduce:

```
echo deb http://eu.ceph.com/debian-firefly/ $(lsb_release -sc) main | sudo tee
/etc/apt/sources.list.d/ceph.list
```

- 3. Update our repository and install **ceph-deploy**:

```
sudo apt-get update
sudo apt-get install ceph-deploy
```

RED HAT PACKAGE MANAGER (RPM)

For **Red Hat(rhel6, rhel7)**, **CentOS (el6, el7)**, and **Fedora 19-20 (f19-f20)** distributions we must follow next steps:

- 1. Add the package to our repository. We must open a text editor and create a Yellowdog Updater, Modified (YUM) entry. Use the file path **/etc/yum.repos.d/ceph.repo**. For example:

```
sudo vim /etc/yum.repos.d/ceph.repo
```

Then we must copy and paste the next example code, replacing **ceph-release** with the recent major release of Ceph (e.g., **firefly**). We must replace **distro** with our Linux distribution (e.g., **el6** for CentOS 6, **el7** for CentOS 7, **rhel6** for Red Hat 6.5, **rhel7** for Red Hat 7, and **fc19** or **fc20** for Fedora 19 or Fedora 20. Finally, we must save the contents to the **/etc/yum.repos.d/ceph.repo** file.

```
[ceph-noarch]
name=Ceph noarch packages
baseurl=http://ceph.com/rpm-{ceph-release}/{distro}/noarch
enabled=1
gpgcheck=1
type=rpm-md
gpgkey=https://ceph.com/git/?p=ceph.git;a=blob_plain;f=keys/release.asc
```

- 2. Next we must update our repository and install **ceph-deploy**:

```
sudo yum update
sudo yum install ceph-deploy
```

2.10.1.6.4.2 Ceph Node Setup

As Ceph daemons pass critical messages to each other, which must be processed before daemons reach a timeout threshold, if the clocks in Ceph monitor nodes are not synchronized, it can produce anomalies like for example daemons ignoring received messages or Timeouts triggered too soon/late when a message wasn't received in time. For that reason, first of all we must prevent issues arising from clock drift by installing NTP specially in Monitor nodes, that in our case will be the all three nodes.

INSTALLING NTP

NTP is a protocol designed to synchronize the clocks of computers over a network. To install it, we must proceed in different ways depending on the distribution in which we work:

- On Debian and Ubuntu:

```
sudo apt-get install ntp
```

- On CentOS and RHEL:

```
sudo yum install ntp ntpdate ntp-doc
```

Next we must ensure that we enable the NTP service and that each Ceph Node uses the same NTP time server.

The ceph-node1 must have password-less SSH access to the other Ceph nodes. When **ceph-deploy** logs in to a Ceph node as a user, that particular user must have password-less **sudo** privileges.

In order to resolve this, we must follow next steps.

INSTALLING SSH SERVER

In all Ceph Nodes we perform the following steps:

- 1. Install an SSH server on each Ceph Node:

```
sudo apt-get install openssh-server
```

or

```
sudo yum install openssh-server
```

- 2. Ensure that the SSH server is running on all Ceph Nodes.

```
sudo /etc/init.d/sshd restart
```

CREATE A CEPH USER

As explained earlier, the **ceph-deploy** utility must login to a Ceph node as a user that has password-less **sudo** privileges because it needs to install software and configuration files without prompting for password.

The newer versions of **ceph-deploy** support a **–username** option so we can specify any user that has password-less **sudo** (including **root**, but this is **NOT** recommended). In order to use **ceph-deploy –username {username}**, the user we specify must have password-less SSH access to the Ceph nodes, as **ceph-deploy** will not prompt for a password.

For that reason we will create a user in all Ceph nodes of our cluster. To improve easy of use, we will use the same user name across the cluster but we must not select obvious user names, because hackers typically use them with brute force hacks. Following next steps, substituting **{username}** for the user name we define, will help us to create a user with password-less **sudo**:

- 1. We create a user on each Ceph Node:

```
ssh user@ceph-server
sudo useradd -d /home/{username} -m {username}
sudo passwd {username}
```

  In our case, the user we decided to create is **molt**, so in the previous commands we replace **{username}** for **molt**.

- 2. We must ensure that our new user has sudo privileges in all Ceph Nodes:

```
echo "{username} ALL = (root) NOPASSWD:ALL" | sudo tee /etc/sudoers.d/{username}
sudo chmod 0440 /etc/sudoers.d/{username}
```

  Where **username** is **molt** in our case.

ENABLE PASSWORD-LESS SSH

Now, as explained earlier, since **ceph-deploy** will not prompt for a password, we must generate SSH keys on the **admin** Node (ceph-node1) and distribute the public key to each Ceph Node. When we configure the SSH, we leave the paraphrase empty and proceed with the default settings:

- 1. Generate the SSH keys, but we must be careful to not use **sudo** or the **root** user. In our case:

```
ssh-keygen

Generating public/private rsa key pair.
Enter file in which to save the key (/home/marius/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/marius/.ssh/id_rsa.
Your public key has been saved in /home/marius/.ssh/id_rsa.pub.
The key fingerprint is:
99:64:df:75:da:0e:74:62:30:7f:50:2a:ed:e5:c9:4f marius@ceph-node1
The key's randomart image is:
+--[ RSA 2048]----+
|            o ...|
|             = o |
|        o   . O =|
|       o + . *.X.|
|        S . . +oE|
|               +.|
|                o|
|                 |
|                 |
+-----------------+
```

- 2. Then we copy the SSH key IDs to ceph-node2 and ceph-node3 using the new created user. After this, we will be able to log in on the other nodes without a password for our created user.

  In our case, so as to copy the SSH key to ceph-node2 for the **molt** user from ceph-node1:

```
ssh-copy-id molt@ceph-node2

/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter
 out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted
 now it is to install the new keys
molt@ceph-node2's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'molt@ceph-node2'"
and check to make sure that only the key(s) you wanted were added.
```

Next we do the same with ceph-node3:

```
ssh-copy-id molt@ceph-node3
...
```

- 3. Now, in order to empower **ceph-deploy** to log in to Ceph nodes as the user we created without requiring us to specify **–username {username}** each time we execute **ceph-deploy**, we must modify the **/.ssh/config** file of our **ceph-deploy** admin node (in our case is ceph-node1). This has the added benefit of streamlining ssh and scp usage.

  In our case:

```
Host ceph-node1
   Hostname ceph-node1
   User molt
Host ceph-node2
   Hostname ceph-node2
   User molt
Host ceph-node3
   Hostname ceph-node3
   User molt
```

NOTE: In our Ceph Storage Cluster examples all the following steps aren't necessary. They only must be made if we use distros like CentOS or RHEL.

NETWORK CONFIGURATION

In order to build a healthy and high performance Ceph Storage Cluster, network configuration is critical. In our Ceph Storage Cluster, the Ceph Clients make requests directly to Ceph OSD Daemons and these perform data replication on behalf of Ceph Clients. This means replication and other factors impose additional loads on our Ceph Storage Cluster networks.

As we explained earlier, in our examples we will use a Quick Start configuration so as to set our Ceph Storage Cluster, and this will provide a trivial Ceph configuration file that sets monitor IP addresses and daemon host names only. Unless we specify a cluster network, Ceph assumes a single "public" network. In our sandbox environment, the Ceph works just fine with a public network only, but if we want a larger cluster, with a second cluster network we can get:

- Performance improvement:

  As we explained above, Ceph OSD Daemons handle data replication for the Ceph Clients. When Ceph OSD Daemons replicate data more than once, the network load between Ceph OSD Daemons easily dwarfs

the network load between Ceph Clients and the Ceph Storage Cluster and this can introduce latency and create a performance problem. The other cases that can introduce significant latency on the public network are recovery and rebalancing.

- Security improvement:

  When traffic between Ceph OSD Daemons is discontinue, the placement groups may no longer reflect an **active + clean** state, which may prevent users from reading and writing data. So as to prevent this, it is recommended to use two different networks.

So, if we want a large Ceph Storage Cluster it is recommended to work with two networks as we can see in the following figure. We will have a public (front-side) network and a cluster (back-side) network. In these cases, each Ceph Node will need to have more than one NIC in order to support two networks.

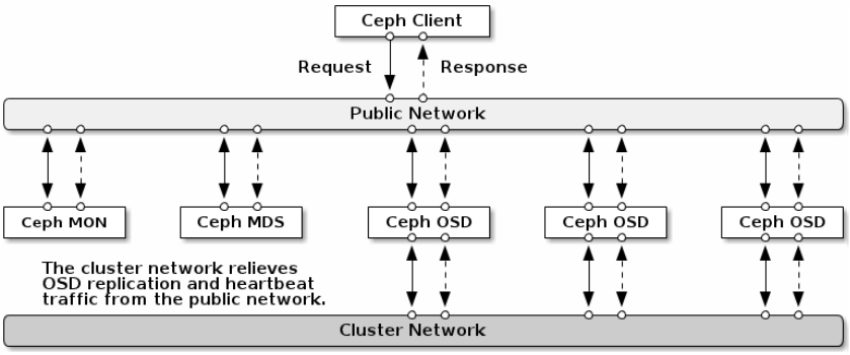


Figure 2.39: Ceph network.

ENABLE NETWORKING ON BOOT UP

In some distros like CentOS, Ceph OSDs can not peer with each other and report to the Ceph Monitors over the network because the networking is off by default. So, in this case, the Ceph Storage cluster cannot come online during boot up until we enable networking.

As networking is enabled by default in Debian and Ubuntu distros, we have no problem with the boot up, but if we use CentOS or RedHat distros we must ensure that, during boot up, our network interface/s turn/s on so that our Ceph daemons can communicate over the network. To do it, we must:

- Navigate to **/etc/sysconfig/network-scripts** and ensure that the **ifcfg-{iface}** file has **ONBOOT** set to yes.

OPEN REQUIRED PORTS

By default, in a Ceph Storage Cluster, the Ceph Monitors use port **6789** and Ceph OSDs use a port range of **6800:7300** to communicate. In some distributions like CentOS or RHEL, the default firewall configuration is fairly strict, so we must change our firewall settings to allow inbound requests so that clients can communicate with daemons on our network.

For example, if we use RHEL 7 distribution, so as to change to change the **firewalld** configuration we must add port **6789** for Ceph Monitor nodes and ports **6800:7300** for Ceph OSDs to the public zone and make this setting permanent in order to be enabled on reboot:

```
sudo firewall-cmd --zone=public --add-port=6789/tcp --permanent
```

And for the **iptables** we must add port **6789** for Monitors and ports **6800:7300** for OSDs like following and ensure that changes are persistent:

```
sudo iptables -A INPUT -i {iface} -p tcp -s {ip-address}/{netmask} --dport 6789 -j ACCEPT
```

```
/sbin/service iptables save
```

TTY

Also on CentOS and RHEL distros, we must change the **/etc/sudoers** configuration and set **requiretty** to off so as to can correctly execute **ceph-deploy** commands. To disable it we must execute **sudo visudo**, locate the **Defaults requiretty** setting and change it to **Defaults:ceph !requiretty** or comment it out to ensure that **ceph-deploy** can connect using the user we created above.

SELINUX

Also on CentOS and RHEL, we must configure **/etc/selinux/config** file and set SELinux from Enforcing (by default) to Permissive or disabling it entirely. If we only want to set SELinux to Permissive, we must execute the following command:

```
sudo setenforce 0
```

PRIORITIES/PREFERENCES

On CentOS and RHEL we must ensure that our package manager has priority/preferences packages installed and enabled:

```
sudo yum install yum-plugin-priorities
```

In RHEL 7 we may need to enable optional repositories like following:

```
sudo yum install yum-plugin-priorities --enablerepo=rhel-7-server-optional-rpms
```

Once we have checked and configured all the software requirements, in our all three examples we can proceed to **ceph-deploy** installation.

2.10.1.6.5 Ceph Storage Cluster examples

As we explained above, in this chapter we will see how to create a sandbox environment using three different virtualization technologies. In the first example we will use VirtualBox machines; in the second one, we will use LXC containers; in the third example we will use the Docker technology.

Since Ceph is an open source software-defined storage deployed on top of commodity hardware, with this virtual setups we will be able to imitate a fully functioning Ceph scenery and create the Ceph Storage Clusters as if we are working in a real environment.

In the three examples we will create a 3 Node Cluster with no single point of failure to provide highly redundant storage.

This example will create a 3 node Ceph cluster with no single point of failure to provide highly redundant storage. The below diagram shows the layout of our 3 Node Ceph Storage Cluster examples.

2.10.1.6.5.1 Ceph Cluster with VirtualBox

In order to test the Ceph-Deployment tool and to deploy our first Ceph Storage Cluster we will create three Ubuntu virtual machines (VM) for our three Ceph Nodes following next steps:

- **Step 1. Install VirtualBox on Ubuntu.**

  Oracle VirtualBox is a free software available at http://www.virtualbox.org for all OS. In our case, we download **VirtualBox 5.0.2** for Linux hosts and install it in our Ubuntu host machine. We also download the **Ubuntu 14.04 iso** image so as to install it in out three virtual machines.

  So as to save time and increase our productivity, to set our sandbox environment we will first create a single VM, install the downloaded Ubuntu iso on it and then we will clone this VM twice.

- **Step 2. To create the first virtual machine**

  After the installation of the VirtualBox software we can use the **GUI-based New Virtual Machine Wizard** so as to create our first VM, but in our case we will use the **CLI** commands to create a VM with the following characteristics:

  - 1 CPU.
  - 1024 MB memory
  - 10 GB x 4 hard disks. One drive is used for OS and the other three for Ceph OSD.
  - 2 network adapters.
  - Ubuntu 14.04 ISO

  To create our first VM we will follow next steps:

  - 1. Create the first virtual machine called **ceph-node1**:

```
# VBoxManage createvm --name ceph-node1 --ostype Linux --register
```

```
# VBoxManage modifyvm ceph-node1 --memory 1024 --nic1 nat --nic2 hostonly
  --hostonlyadapter2 vboxnet1
```

    NOTE: with the installation of VirtualBox software, a network adapter called **vboxnet1** is created, and we can use it to connect our VM.

  - 2. Create a CD-Drive and attach Ubuntu 14.04 ISO image to the first virtual machine:

```
# VBoxManage storagectl ceph-node1 --name "IDE Controller" --add ide --controller
  PIIX4 --hostiocache on --bootable on
```

```
# VBoxManage storageattach ceph-node1 --storagectl "IDE Controller" --type
  dvddrive --port 0 --device 0 --medium /Downloads/ubuntu-14.04.3-desktop-i386.iso
```

  - 3. Create SATA interface, OS hard drive and attach them to VM. We must make sure that the VirtualBox host has enough free space for creating vm disks.

```
# VBoxManage storagectl ceph-node1 --name "SATA Controller" --add sata
  --controller IntelAHCI --hostiocache on --bootable on
```

```
# VBoxManage  createhd --filename OS-ceph-node1.vdi --size 10240
```

```
# VBoxManage storageattach ceph-node1 --storagectl "SATA Controller"
  --port 0 --device 0 --type hdd --medium OS-ceph-node1.vdi
```

- 4. Create SATA interface, first ceph disk and attach them to VM:

```
# VBoxManage  createhd --filename ceph-node1-osd1.vdi --size 10240
```

```
 VBoxManage storageattach ceph-node1 --storagectl "SATA Controller"
   --port 1 --device 0 --type hdd --medium ceph-node1-osd1.vdi
```

- 5. Create SATA interface, second ceph disk and attach them to VM:

```
VBoxManage  createhd --filename ceph-node1-osd2.vdi --size 10240
```

```
VBoxManage storageattach ceph-node1 --storagectl "SATA Controller"
  --port 2 --device 0 --type hdd --medium ceph-node1-osd2.vdi
```

- 6. Create SATA interface, third ceph disk and attach them to VM:

```
VBoxManage  createhd --filename ceph-node1-osd3.vdi --size 10240
```

```
VBoxManage storageattach ceph-node1 --storagectl "SATA Controller"
  --port 3 --device 0 --type hdd --medium ceph-node1-osd3.vdi
```

- 7. Once we have our **ceph-node1** VM created, to power it on we run the following command and then we boot from the ISO image to install the **Ubuntu 14.04** OS.

```
# VboxManage startvm ceph-node1 --type gui
```

- **Step 3. Network Configuration**

  With the Ubuntu OS successfully installed, we edit the network configuration of the machine:

  - 1. Edit the machine hostname to **ceph-node1** in **/etc/hostsname**:

```
# sudo vi /etc/hostname
```

  - 2. Configure the network interfaces in **/etc/network/interfaces**:

```
# sudo vi /etc/network/interfaces:
        # interfaces(5) file used by ifup(8) and ifdown(8)
        auto lo
        iface lo inet loopback

        # Primary network interface
        auto eth0
        iface eth0 inet static
        address 192.168.1.10
        netmask 255.255.255.0
        network 192.168.1.0
        broadcast 192.168.1.255
        gateway 192.168.1.1
```

  - 3. Configure the hosts names in **/etc/hosts** like:

```
        192.168.1.10 ceph-node1
        192.168.1.20 ceph-node2
        192.168.1.30 ceph-node3
```

– 4. Once we configured the network settings, we must restart the VM and then we should try to log via SSH from our host, and also test the internet connection of our **ceph-node1** VM. The internet connectivity is required to download Ceph packages. From host:

```
# ssh root@192.168.1.10
```

If ssh connection is ok, we can continue with next steps.

- **Step 4. Creating two clones of our first machine:**

Once we configured correctly the network setup, we must shut down the ceph-node1 VM so as to can make two clones of it:

– Create clone of **ceph-node1** named **ceph-node2**:

```
# VBoxManage clonevm --name ceph-node2 ceph-node1 -register
```

– Create clone of **ceph-node1** named **ceph-node3**:

```
# VBoxManage clonevm --name ceph-node3 ceph-node1 -register
```

After performing the creation of **ceph-node2** and **ceph-node3** machines, we must make the same network configuration that we made in **ceph-node1**, but with **192.168.1.20** ip for **ceph-node2** and **192.168.1.30** ip for **ceph-node3** VM.

After the correct configuration of all three virtual machines, we will have the following diagram:

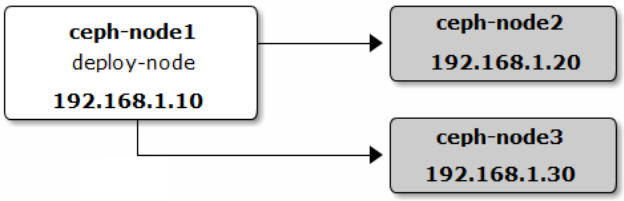


Figure 2.40: Ceph network diagram.

- **Step 5. Deploying our first Ceph Storage Cluster:**

To deploy our first Ceph cluster, we use the **ceph-deploy** tool so as to install and configure Ceph on our all three virtual machines. The **ceph-deploy** tool is a part of the Ceph software-defined storage used for the easier deployment and management of our Ceph storage cluster.

So, next we will proceed to configure our three machines as we can see in the above diagram.

– 1. We must repeat the steps explained in **Prior Software configurations** section. Once we have added the needed release packages (**Firefly (0.80)**) and created our ceph user we proceed to install packages.

– 2. Installing packages in ceph-node1:

```
# sudo apt-get update
```

```
# sudo apt-get install ceph-deploy
```

– 3. Creating directory.

Before creating the Ceph Storage Cluster we create a directory on our **ceph-node1** for maintaining the configuration files and keys that **ceph-deploy** generates for our cluster. We must ensure that we are in this directory when we execute **ceph-deploy**. We create a directory for ceph:

```
# mkdir my_cluster
# cd my_cluster
```

– 4. Trouble resolutions.

It is important to note that, if we have trouble at any of the next points, we can start over executing the following to purge configuration:

```
# ceph-deploy purgedata {ceph-node} [{ceph-node}]
```

```
# ceph-deploy forgetkeys
```

If we need we can also purge the Ceph packages too executing the next command, but then we must re-install Ceph:

```
# ceph-deploy purge {ceph-node} [{ceph-node}]
```

– 5. Create Ceph Cluster.

Next, in **ceph-node1** we will create a Ceph cluster using **ceph-deploy** by executing the next command from my_cluster directory. The **ceph-deploy** utility will use DNS to resolve hostnames to IP adresses. Will name the monitor as **ceph-node1** and it will also add the specified host names to the Ceph configuration file. We execute:

```
marius@ceph-node1:~/my_cluster$ ceph-deploy new ceph-node1
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy new ceph-node1
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  func                          : <function new at 0xb6dab5a4>
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6d3d24c>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  ssh_copykey                   : True
[ceph_deploy.cli][INFO  ]  mon                           : ['ceph-node1']
[ceph_deploy.cli][INFO  ]  public_network                : None
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  cluster_network               : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.cli][INFO  ]  fsid                          : None
[ceph_deploy.new][DEBUG ] Creating new cluster named ceph
[ceph_deploy.new][INFO  ] making sure passwordless SSH succeeds
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
[ceph-node1][DEBUG ] detect platform information from remote host
[ceph-node1][DEBUG ] detect machine type
[ceph-node1][DEBUG ] find the location of an executable
[ceph-node1][INFO  ] Running command: sudo /bin/ip link show
[ceph-node1][INFO  ] Running command: sudo /bin/ip addr show
[ceph-node1][DEBUG ] IP addresses found: ['192.168.1.10', '192.168.10.10']
[ceph_deploy.new][DEBUG ] Resolving host ceph-node1
[ceph_deploy.new][DEBUG ] Monitor ceph-node1 at 192.168.1.10
[ceph_deploy.new][DEBUG ] Monitor initial members are ['ceph-node1']
[ceph_deploy.new][DEBUG ] Monitor addrs are ['192.168.1.10']
[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...
```

The new subcommand of **ceph-deploy** deploys a new cluster with ceph as the cluster name, by default; it generates a cluster configuration and keying files. If we list the **my_cluster** directory we will find the **ceph.conf** and **ceph.mon.keyring** files.

– 6. The next step is to install Ceph software binaries on each node using **ceph-deploy**. We execute the following command from **ceph-node1**:

```
# marius@ceph-node1:~/my_cluster$ ceph-deploy --username molt install ceph-node1
 ceph-node2 ceph-node3
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy install ceph-node1
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  testing                       : None
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6d2f8cc>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  install_mds                   : False
[ceph_deploy.cli][INFO  ]  stable                        : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  adjust_repos                  : True
[ceph_deploy.cli][INFO  ]  func                          : <function install
 at 0xb6d743ac>
[ceph_deploy.cli][INFO  ]  install_all                   : False
[ceph_deploy.cli][INFO  ]  repo                          : False
[ceph_deploy.cli][INFO  ]  host                          : ['ceph-node1']
[ceph_deploy.cli][INFO  ]  install_rgw                   : False
[ceph_deploy.cli][INFO  ]  repo_url                      : None
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  install_osd                   : False
[ceph_deploy.cli][INFO  ]  version_kind                  : stable
[ceph_deploy.cli][INFO  ]  install_common                : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  dev                           : master
[ceph_deploy.cli][INFO  ]  local_mirror                  : None
[ceph_deploy.cli][INFO  ]  release                       : None
[ceph_deploy.cli][INFO  ]  install_mon                   : False
[ceph_deploy.cli][INFO  ]  gpg_url                       : None
[ceph_deploy.install][DEBUG ] Installing stable version hammer on cluster ceph
 hosts ceph-node1
[ceph_deploy.install][DEBUG ] Detecting platform for host ceph-node1 ...
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
...
[ceph-node1][DEBUG ] ceph-all start/running
[ceph-node1][DEBUG ] Setting up radosgw (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node1][DEBUG ] radosgw-all start/running
[ceph-node1][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node1][DEBUG ] Setting up ceph-mds (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node1][DEBUG ] ceph-mds-all start/running
[ceph-node1][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node1][INFO  ] Running command: sudo ceph --version
[ceph-node1][DEBUG ] ceph version 0.80.10 (ea6c958c38df1216bf95c927f143d8b13c4a9e70)

[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy install ceph-node2
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  testing                       : None
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6db38cc>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  install_mds                   : False
```

```
[ceph_deploy.cli][INFO  ]  stable                          : None
[ceph_deploy.cli][INFO  ]  default_release                 : False
[ceph_deploy.cli][INFO  ]  username                        : None
[ceph_deploy.cli][INFO  ]  adjust_repos                    : True
[ceph_deploy.cli][INFO  ]  func                            : <function install
 at 0xb6df83ac>
[ceph_deploy.cli][INFO  ]  install_all                     : False
[ceph_deploy.cli][INFO  ]  repo                            : False
[ceph_deploy.cli][INFO  ]  host                            : ['ceph-node2']
[ceph_deploy.cli][INFO  ]  install_rgw                     : False
[ceph_deploy.cli][INFO  ]  repo_url                        : None
[ceph_deploy.cli][INFO  ]  ceph_conf                       : None
[ceph_deploy.cli][INFO  ]  install_osd                     : False
[ceph_deploy.cli][INFO  ]  version_kind                    : stable
[ceph_deploy.cli][INFO  ]  install_common                  : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                  : False
[ceph_deploy.cli][INFO  ]  quiet                           : False
[ceph_deploy.cli][INFO  ]  dev                             : master
[ceph_deploy.cli][INFO  ]  local_mirror                    : None
[ceph_deploy.cli][INFO  ]  release                         : None
[ceph_deploy.cli][INFO  ]  install_mon                     : False
[ceph_deploy.cli][INFO  ]  gpg_url                         : None
[ceph_deploy.install][DEBUG ] Installing stable version hammer on cluster ceph
 hosts ceph-node2
[ceph_deploy.install][DEBUG ] Detecting platform for host ceph-node2 ...
[ceph-node2][DEBUG ] connection detected need for sudo
[ceph-node2][DEBUG ] connected to host: ceph-node2
...
[ceph-node2][DEBUG ] ceph-all start/running
[ceph-node2][DEBUG ] Setting up radosgw (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node2][DEBUG ] radosgw-all start/running
[ceph-node2][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node2][DEBUG ] Setting up ceph-mds (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node2][DEBUG ] ceph-mds-all start/running
[ceph-node2][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node2][INFO  ] Running command: sudo ceph --version
[ceph-node2][DEBUG ] ceph version 0.80.10 (ea6c958c38df1216bf95c927f143d8b13c4a9e70)

[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy install ceph-node3
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  verbose                         : False
[ceph_deploy.cli][INFO  ]  testing                         : None
[ceph_deploy.cli][INFO  ]  cd_conf                         : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6e088cc>
[ceph_deploy.cli][INFO  ]  cluster                         : ceph
[ceph_deploy.cli][INFO  ]  install_mds                     : False
[ceph_deploy.cli][INFO  ]  stable                          : None
[ceph_deploy.cli][INFO  ]  default_release                 : False
[ceph_deploy.cli][INFO  ]  username                        : None
[ceph_deploy.cli][INFO  ]  adjust_repos                    : True
[ceph_deploy.cli][INFO  ]  func                            : <function install
 at 0xb6e4d3ac>
[ceph_deploy.cli][INFO  ]  install_all                     : False
[ceph_deploy.cli][INFO  ]  repo                            : False
[ceph_deploy.cli][INFO  ]  host                            : ['ceph-node3']
[ceph_deploy.cli][INFO  ]  install_rgw                     : False
[ceph_deploy.cli][INFO  ]  repo_url                        : None
[ceph_deploy.cli][INFO  ]  ceph_conf                       : None
[ceph_deploy.cli][INFO  ]  install_osd                     : False
[ceph_deploy.cli][INFO  ]  version_kind                    : stable
[ceph_deploy.cli][INFO  ]  install_common                  : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                  : False
[ceph_deploy.cli][INFO  ]  quiet                           : False
[ceph_deploy.cli][INFO  ]  dev                             : master
```

```
[ceph_deploy.cli][INFO  ]  local_mirror                   : None
[ceph_deploy.cli][INFO  ]  release                        : None
[ceph_deploy.cli][INFO  ]  install_mon                    : False
[ceph_deploy.cli][INFO  ]  gpg_url                        : None
[ceph_deploy.install][DEBUG ] Installing stable version hammer on cluster ceph hosts
 ceph-node3
[ceph_deploy.install][DEBUG ] Detecting platform for host ceph-node3 ...
[ceph-node3][DEBUG ] connection detected need for sudo
[ceph-node3][DEBUG ] connected to host: ceph-node3
...
[ceph-node3][DEBUG ] ceph-all start/running
[ceph-node3][DEBUG ] Setting up radosgw (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node3][DEBUG ] radosgw-all start/running
[ceph-node3][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node3][DEBUG ] Setting up ceph-mds (0.80.10-0ubuntu0.14.04.1) ...
[ceph-node3][DEBUG ] ceph-mds-all start/running
[ceph-node3][DEBUG ] Processing triggers for ureadahead (0.100.0-16) ...
[ceph-node3][INFO  ] Running command: sudo ceph --version
[ceph-node3][DEBUG ] ceph version 0.80.10 (ea6c958c38df1216bf95c927f143d8b13c4a9e70)
```

As we can see, the **ceph-deploy** install tool first install all the dependencies followed by the Ceph binaries. Once the installation is completed, we can check the Ceph version and Ceph health on all the nodes, as follows:

```
# ceph -v
ceph version 0.80.10 (ea6c958c38df1216bf95c927f143d8b13c4a9e70)
```

In this case we have the same output in all nodes. We can see that in the three nodes we have the Ceph **Firefly (0.80)** release installed.

– 7. Now we can create our first Ceph Monitor on **ceph-node1** using the following command:

```
# ceph-deploy mon create {ceph-node}
```

In our case:

```
marius@ceph-node1:~/my_cluster$ ceph-deploy mon create ceph-node1
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy mon create ceph-node1
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                       : None
[ceph_deploy.cli][INFO  ]  verbose                        : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                 : False
[ceph_deploy.cli][INFO  ]  subcommand                     : create
[ceph_deploy.cli][INFO  ]  quiet                          : False
[ceph_deploy.cli][INFO  ]  cd_conf                        : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6d4f60c>
[ceph_deploy.cli][INFO  ]  cluster                        : ceph
[ceph_deploy.cli][INFO  ]  mon                            : ['ceph-node1']
[ceph_deploy.cli][INFO  ]  func                           : <function mon at 0xb6d403e4>
[ceph_deploy.cli][INFO  ]  ceph_conf                      : None
[ceph_deploy.cli][INFO  ]  default_release                 : False
[ceph_deploy.cli][INFO  ]  keyrings                       : None
[ceph_deploy.mon][DEBUG ] Deploying mon, cluster ceph hosts ceph-node1
[ceph_deploy.mon][DEBUG ] detecting platform for host ceph-node1 ...
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
[ceph-node1][DEBUG ] detect platform information from remote host
[ceph-node1][DEBUG ] detect machine type
[ceph-node1][DEBUG ] find the location of an executable
[ceph_deploy.mon][INFO  ] distro info: Ubuntu 14.04 trusty
[ceph-node1][DEBUG ] determining if provided host has same hostname in remote
[ceph-node1][DEBUG ] get remote short hostname
[ceph-node1][DEBUG ] deploying mon to ceph-node1
```

```
[ceph-node1][DEBUG ] get remote short hostname
[ceph-node1][DEBUG ] remote hostname: ceph-node1
[ceph-node1][DEBUG ] write cluster configuration to /etc/ceph/{cluster}.conf
[ceph-node1][DEBUG ] create the mon path if it does not exist
[ceph-node1][DEBUG ] checking for done path: /var/lib/ceph/mon/ceph-ceph-node1/done
[ceph-node1][DEBUG ] done path does not exist: /var/lib/ceph/mon/ceph-ceph-node1/done
[ceph-node1][INFO  ] creating keyring file: /var/lib/ceph/tmp/ceph-ceph-node1.mon.keyring
[ceph-node1][DEBUG ] create the monitor keyring file
[ceph-node1][INFO  ] Running command: sudo ceph-mon --cluster ceph --mkfs -i ceph-node1
 --keyring /var/lib/ceph/tmp/ceph-ceph-node1.mon.keyring
[ceph-node1][DEBUG ] ceph-mon: mon.noname-a 192.168.1.10:6789/0 is local, renaming to
 mon.ceph-node1
[ceph-node1][DEBUG ] ceph-mon: set fsid to 6fac025f-1bfe-4a6c-b09a-5f7863cc447b
[ceph-node1][DEBUG ] ceph-mon: created monfs at /var/lib/ceph/mon/ceph-ceph-node1 for
 mon.ceph-node1
[ceph-node1][INFO  ] unlinking keyring file /var/lib/ceph/tmp/ceph-ceph-node1.mon.keyring
[ceph-node1][DEBUG ] create a done file to avoid re-doing the mon deployment
[ceph-node1][DEBUG ] create the init path if it does not exist
[ceph-node1][DEBUG ] locating the `service` executable...
[ceph-node1][INFO  ] Running command: sudo initctl emit ceph-mon cluster=ceph
 id=ceph-node1
[ceph-node1][INFO  ] Running command: sudo ceph --cluster=ceph --admin-daemon
/var/run/ceph/ceph-mon.ceph-node1.asok mon_status
[ceph-node1][DEBUG ] ********************************************************************
[ceph-node1][DEBUG ] status for monitor: mon.ceph-node1
[ceph-node1][DEBUG ] {
[ceph-node1][DEBUG ]   "election_epoch": 2,
[ceph-node1][DEBUG ]   "extra_probe_peers": [],
[ceph-node1][DEBUG ]   "monmap": {
[ceph-node1][DEBUG ]     "created": "0.000000",
[ceph-node1][DEBUG ]     "epoch": 1,
[ceph-node1][DEBUG ]     "fsid": "6fac025f-1bfe-4a6c-b09a-5f7863cc447b",
[ceph-node1][DEBUG ]     "modified": "0.000000",
[ceph-node1][DEBUG ]     "mons": [
[ceph-node1][DEBUG ]       {
[ceph-node1][DEBUG ]         "addr": "192.168.1.10:6789/0",
[ceph-node1][DEBUG ]         "name": "ceph-node1",
[ceph-node1][DEBUG ]         "rank": 0
[ceph-node1][DEBUG ]       }
[ceph-node1][DEBUG ]     ]
[ceph-node1][DEBUG ]   },
[ceph-node1][DEBUG ]   "name": "ceph-node1",
[ceph-node1][DEBUG ]   "outside_quorum": [],
[ceph-node1][DEBUG ]   "quorum": [
[ceph-node1][DEBUG ]     0
[ceph-node1][DEBUG ]   ],
[ceph-node1][DEBUG ]   "rank": 0,
[ceph-node1][DEBUG ]   "state": "leader",
[ceph-node1][DEBUG ]   "sync_provider": []
[ceph-node1][DEBUG ] }
[ceph-node1][DEBUG ] ********************************************************************
[ceph-node1][INFO  ] monitor: mon.ceph-node1 is running
[ceph-node1][INFO  ] Running command: sudo ceph --cluster=ceph --admin-daemon
/var/run/ceph/ceph-mon.ceph-node1.asok mon_status
```

After these steps, in our my_cluster directory we can see some new files:

```
~/my_cluster$ ceph-deploy mon create ceph-node1
ceph.conf ceph.log ceph.mon.keyring
```

In the **ceph.conf** file we can find the first configuration of our cluster. We can see that we only have the **ceph-node1** added, but the other two nodes will be added when we deploy the Ceph Nodes Monitors.

```
~/my_cluster$ cat ceph.conf
[global]
```

```
fsid = 6fac025f-1bfe-4a6c-b09a-5f7863cc447b
mon_initial_members = ceph-node1
mon_host = 192.168.1.10
auth_cluster_required = cephx
auth_service_required = cephx
auth_client_required = cephx
filestore_xattr_use_omap = true
```

- 8. The next step is to gather the keys. These keys will be used by all the nodes of the cluster for authenticate all the services:

```
~/my_cluster$ ceph-deploy gatherkeys ceph-node1
...
```

The result of the previous command is the creation of the next files in our **mycluster** directory:

```
~/my_cluster$ ls
ceph.bootstrap-mds.keyring  ceph.client.admin.keyring  ceph.log release.asc
ceph.bootstrap-osd.keyring  ceph.conf   ceph.mon.keyring
```

- 9. To check our cluster status we can use the next command:

```
~/my_cluster$ ceph status
2015-09-02 12:41:19.419609 b616eb40 -1 monclient(hunting): ERROR: missing keyring, cannot
 use cephx for authentication
2015-09-02 12:41:19.419623 b616eb40  0 librados: client.admin initialization error (2)
 No such file or directory
Error connecting to cluster: ObjectNotFound
```

As we can see, at this stage our Ceph Cluster it is not ready yet because we also need to create an Object Storage Device (OSD) in a few more nodes so as to set up a distributed, replicated object storage.

- **Step 6. Installing our first OSD**

Following the next steps we will create and add our first OSD to our Ceph Cluster:

- 1. List partitions on the VM disk in order to find the storage partitions:

```
~/my_cluster$ ceph-deploy disk list ceph-node1
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy disk list ceph-node1
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  subcommand                    : list
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6e1938c>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  func                          : <function disk at 0xb6e0780c>
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.cli][INFO  ]  disk                          : [('ceph-node1', None, None)]
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
[ceph-node1][DEBUG ] detect platform information from remote host
[ceph-node1][DEBUG ] detect machine type
[ceph-node1][DEBUG ] find the location of an executable
[ceph_deploy.osd][INFO  ] Distro info: Ubuntu 14.04 trusty
[ceph_deploy.osd][DEBUG ] Listing disks on ceph-node1...
[ceph-node1][DEBUG ] find the location of an executable
[ceph-node1][INFO  ] Running command: sudo /usr/sbin/ceph-disk list
```

```
[ceph-node1][DEBUG ] /dev/sda :
[ceph-node1][DEBUG ]  /dev/sda1 other, ext4, mounted on /
[ceph-node1][DEBUG ]  /dev/sda2 other, 0x5
[ceph-node1][DEBUG ]  /dev/sda5 swap, swap
[ceph-node1][DEBUG ] /dev/sdb other, unknown
[ceph-node1][DEBUG ] /dev/sdc other, unknown
[ceph-node1][DEBUG ] /dev/sdd other, unknown
[ceph-node1][DEBUG ] /dev/sr0 other, iso9660, mounted on
 /media/marius/VBOXADDITIONS_5.0.2_102096
```

From the command output we must identify the disks on which we should create the Ceph OSD. We must be careful to not select the disk where we have the OS-partition. So, from the previous output we can select **sdb**, **sdc** and **sdd**.

– 2. Now, with the **disk zap** subcommand we will destroy the existing partition table and content from the selected disk, so we must be sure that we use the correct disk device name:

```
~/my_cluster$ ceph-deploy disk zap ceph-node1:sdb ceph-node1:sdc ceph-node1:sdd
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy disk zap ceph-node1:sdb
 ceph-node1:sdc ceph-node1:sdd
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  subcommand                    : zap
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
  Conf instance at 0xb6d7234c>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  func                          : <function disk at 0xb6d6080c>
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.cli][INFO  ]  disk                          : [('ceph-node1', '/dev/sdb',
  None), ('ceph-node1', '/dev/sdc', None), ('ceph-node1', '/dev/sdd', None)]
[ceph_deploy.osd][DEBUG ] zapping /dev/sdb on ceph-node1
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
[ceph-node1][DEBUG ] detect platform information from remote host
[ceph-node1][DEBUG ] detect machine type
[ceph-node1][DEBUG ] find the location of an executable
[ceph_deploy.osd][INFO  ] Distro info: Ubuntu 14.04 trusty
[ceph-node1][DEBUG ] zeroing last few blocks of device
[ceph-node1][DEBUG ] find the location of an executable
[ceph-node1][INFO  ] Running command: sudo /usr/sbin/ceph-disk zap /dev/sdb
...
[ceph_deploy.osd][INFO  ] Distro info: Ubuntu 14.04 trusty
[ceph-node1][DEBUG ] zeroing last few blocks of device
[ceph-node1][DEBUG ] find the location of an executable
[ceph-node1][INFO  ] Running command: sudo /usr/sbin/ceph-disk zap /dev/sdd
[ceph-node1][DEBUG ] Creating new GPT entries.
[ceph-node1][DEBUG ] GPT data structures destroyed! You may now partition the disk using
fdisk or [ceph-node1][DEBUG ] other utilities.
[ceph-node1][DEBUG ] The operation has completed successfully.
[ceph_deploy.osd][DEBUG ] Calling partprobe on zapped device /dev/sdd
[ceph-node1][INFO  ] Running command: sudo partprobe /dev/sdd
```

– 3. Now we use the **osd create** subcommand that it will first prepare the disk, erasing it with a file-system, which is **xfc** by default. Then, it will activate the disk's first partition as data partition and second partition as journal:

```
~/my_cluster$ ceph-deploy osd create ceph-node1:sdb
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy osd create
```

```
  ceph-node1:sdb
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  disk                          : [('ceph-node1', '/dev/sdb',
  None)]
[ceph_deploy.cli][INFO  ]  dmcrypt                       : False
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  subcommand                    : create
[ceph_deploy.cli][INFO  ]  dmcrypt_key_dir               : /etc/ceph/dmcrypt-keys
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
  Conf instance at 0xb6dff82c>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  fs_type                       : xfs
[ceph_deploy.cli][INFO  ]  func                          : <function osd at 0xb6de97d4>
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.cli][INFO  ]  zap_disk                      : False
[ceph_deploy.osd][DEBUG ] Preparing cluster ceph disks ceph-node1:/dev/sdb:
[ceph-node1][DEBUG ] connection detected need for sudo
[ceph-node1][DEBUG ] connected to host: ceph-node1
[ceph-node1][DEBUG ] detect platform information from remote host
[ceph-node1][DEBUG ] detect machine type
[ceph-node1][DEBUG ] find the location of an executable
[ceph_deploy.osd][INFO  ] Distro info: Ubuntu 14.04 trusty
[ceph_deploy.osd][DEBUG ] Deploying osd to ceph-node1
[ceph-node1][DEBUG ] write cluster configuration to /etc/ceph/{cluster}.conf
...
[ceph-node1][WARNIN] INFO:ceph-disk:Running command: /sbin/sgdisk
--typecode=1:4fbd7e29-9d25-41b8-afd0-062c0ceff05d -- /dev/sdb
[ceph-node1][DEBUG ] The operation has completed successfully.
[ceph-node1][WARNIN] DEBUG:ceph-disk:Calling partprobe on prepared device /dev/sdb
[ceph-node1][WARNIN] INFO:ceph-disk:Running command: /sbin/partprobe /dev/sdb
[ceph-node1][INFO  ] Running command: sudo udevadm trigger --subsystem-match=block
  --action=add
[ceph-node1][INFO  ] checking OSD status...
[ceph-node1][INFO  ] Running command: sudo ceph --cluster=ceph osd stat --format=json
[ceph-node1][WARNIN] there is 1 OSD down
[ceph-node1][WARNIN] there is 1 OSD out
[ceph_deploy.osd][DEBUG ] Host ceph-node1 is now ready for osd use.
```

- **Step 7. Adding more Ceph Monitors**

  Now we have a single-node Ceph Cluster but, for high availability, a Ceph storage cluster relies on an odd number of monitors that's more than one, for example 3 or 5. In our case, since we already have one monitor running on **ceph-node1**, we will create two more monitors so as to form a **quorum**. Ceph uses the **Paxos** algorithm to maintain quorum majority. So, let's create the other monitors for our Ceph Cluster to make it a distributed and reliable Storage Cluster.

  – 1. As the Ceph monitor nodes need to communicate each other, we must adjust the firewall rules in order to let monitors form a quorum. In our case, we will disable firewall on all three nodes running the following commands from **ceph-node1**:

```
# service iptables stop
# chkconfig iptables off
# ssh ceph-node2 service iptables stop
# ssh ceph-node2 chkconfig iptables off
# ssh ceph-node3 service iptables stop
# ssh ceph-node3 chkconfig iptables off
```

  – 2. Now, also from **ceph-node1**, we deploy a monitor on **ceph-node2** and **ceph-node3**:

```
# ~/my_cluster$ ceph-deploy mon create ceph-node2
...
```

```
# ~/my_cluster$ ceph-deploy mon create ceph-node3
...
```

– 3. Next, we will add the recent created monitors to the Cluster. From **ceph-node1** we insert the next command for **ceph-node2** and for **ceph-node3**. The following example is for **ceph-node3**:

```
~/my_cluster$ ceph-deploy mon add ceph-node3
[ceph_deploy.conf][DEBUG ] found configuration file at: /home/marius/.cephdeploy.conf
[ceph_deploy.cli][INFO  ] Invoked (1.5.28): /usr/bin/ceph-deploy mon add ceph-node3
[ceph_deploy.cli][INFO  ] ceph-deploy options:
[ceph_deploy.cli][INFO  ]  username                      : None
[ceph_deploy.cli][INFO  ]  verbose                       : False
[ceph_deploy.cli][INFO  ]  overwrite_conf                : False
[ceph_deploy.cli][INFO  ]  subcommand                    : add
[ceph_deploy.cli][INFO  ]  quiet                         : False
[ceph_deploy.cli][INFO  ]  cd_conf                       : <ceph_deploy.conf.cephdeploy.
 Conf instance at 0xb6dc16ac>
[ceph_deploy.cli][INFO  ]  cluster                       : ceph
[ceph_deploy.cli][INFO  ]  mon                           : ['ceph-node3']
[ceph_deploy.cli][INFO  ]  func                          : <function mon at 0xb6db33ac>
[ceph_deploy.cli][INFO  ]  address                       : None
[ceph_deploy.cli][INFO  ]  ceph_conf                     : None
[ceph_deploy.cli][INFO  ]  default_release               : False
[ceph_deploy.mon][INFO  ] ensuring configuration of new mon host: ceph-node3
[ceph_deploy.admin][DEBUG ] Pushing admin keys and conf to ceph-node3
[ceph-node3][DEBUG ] connection detected need for sudo
[ceph-node3][DEBUG ] connected to host: ceph-node3
[ceph-node3][DEBUG ] detect platform information from remote host
[ceph-node3][DEBUG ] detect machine type
[ceph-node3][DEBUG ] write cluster configuration to /etc/ceph/{cluster}.conf
[ceph_deploy.mon][DEBUG ] Adding mon to cluster ceph, host ceph-node3
[ceph_deploy.mon][DEBUG ] using mon address via configuration: 192.168.1.30:6789
[ceph_deploy.mon][DEBUG ] detecting platform for host ceph-node3 ...
[ceph-node3][DEBUG ] connection detected need for sudo
[ceph-node3][DEBUG ] connected to host: ceph-node3
...
[ceph-node3][INFO  ] Running command: sudo ceph --cluster=ceph --admin-daemon
/var/run/ceph/ceph-mon.ceph-node3.asok mon_status
[ceph-node3][DEBUG ] ************************************************************
[ceph-node3][DEBUG ] status for monitor: mon.ceph-node3
[ceph-node3][DEBUG ] {
[ceph-node3][DEBUG ]   "election_epoch": 0,
[ceph-node3][DEBUG ]   "extra_probe_peers": [
[ceph-node3][DEBUG ]     "192.168.1.10:6789/0"
[ceph-node3][DEBUG ]   ],
[ceph-node3][DEBUG ]   "monmap": {
[ceph-node3][DEBUG ]     "created": "0.000000",
[ceph-node3][DEBUG ]     "epoch": 2,
[ceph-node3][DEBUG ]     "fsid": "6fac025f-1bfe-4a6c-b09a-5f7863cc447b",
[ceph-node3][DEBUG ]     "modified": "2015-09-02 15:34:44.963884",
[ceph-node3][DEBUG ]     "mons": [
[ceph-node3][DEBUG ]       {
[ceph-node3][DEBUG ]         "addr": "192.168.1.10:6789/0",
[ceph-node3][DEBUG ]         "name": "ceph-node1",
[ceph-node3][DEBUG ]         "rank": 0
[ceph-node3][DEBUG ]       },
[ceph-node3][DEBUG ]       {
[ceph-node3][DEBUG ]         "addr": "192.168.1.20:6789/0",
[ceph-node3][DEBUG ]         "name": "ceph-node2",
[ceph-node3][DEBUG ]         "rank": 1
```

```
[ceph-node3][DEBUG ]       }
[ceph-node3][DEBUG ]     ]
[ceph-node3][DEBUG ]   },
[ceph-node3][DEBUG ]   "name": "ceph-node3",
[ceph-node3][DEBUG ]   "outside_quorum": [],
[ceph-node3][DEBUG ]   "quorum": [],
[ceph-node3][DEBUG ]   "rank": -1,
[ceph-node3][DEBUG ]   "state": "probing",
[ceph-node3][DEBUG ]   "sync_provider": []
[ceph-node3][DEBUG ] }
[ceph-node3][DEBUG ] ********************************************************************
[ceph-node3][INFO  ] monitor: mon.ceph-node3 is currently at the state of probing
```

At this point we can check our newly added monitors using the Ceph **status** command:

```
ceph-node1:~$ ceph status
    cluster 6fac025f-1bfe-4a6c-b09a-5f7863cc447b
    health HEALTH_WARN 192 pgs degraded; 192 pgs stuck unclean; clock skew detected on
    mon.ceph-node2, mon.ceph-node3
    monmap e3: 3 mons at {ceph-node1=192.168.1.10:6789/0,ceph-node2=192.168.1.20:6789/0,
    ceph-node3=192.168.1.30:6789/0}, election epoch 24, quorum 0,1,2 ceph-node1,ceph-node2,
    ceph-node3
    osdmap e21: 3 osds: 3 up, 3 in
    pgmap v48: 192 pgs, 3 pools, 0 bytes data, 0 objects
            101724 kB used, 15227 MB / 15326 MB avail
                 192 active+degraded
```

As we can see in the **ceph status** output, at this point we have a running Ceph Cluster with three monitors OSDs. Next we will scale up our Cluster adding more OSDs from the **ceph-node1**.

- **Step 8. Adding more Ceph OSDs**

  To accomplish this, we will follow the same method for OSD addition that we used earlier:

```
~/my_cluster$ ceph-deploy disk list ceph-node2 ceph-node3
~/my_cluster$ ceph-deploy disk zap ceph-node2:sdb ceph-node2:sdc ceph-node2:sdd
~/my_cluster$ ceph-deploy disk zap ceph-node3:sdb ceph-node3:sdc ceph-node3:sdd
~/my_cluster$ ceph-deploy osd create ceph-node2:sdb ceph-node2:sdc ceph-node2:sdd
...
[ceph_deploy.osd][DEBUG ] Host ceph-node2 is now ready for osd use.
~/my_cluster$ ceph-deploy osd create ceph-node3:sdb ceph-node3:sdc ceph-node3:sdd
...
[ceph_deploy.osd][DEBUG ] Host ceph-node3 is now ready for osd use.
```

  Once added the OSDs, now we should have a **healthy** Cluster with nine OSDs in and up. To check this:

```
ceph-node1:~$ ceph status
ceph-node1:~/my_cluster$ ceph status
    cluster 6fac025f-1bfe-4a6c-b09a-5f7863cc447b
     health HEALTH_OK
     monmap e3: 3 mons at {ceph-node1=192.168.1.10:6789/0,ceph-node2=192.168.1.20:6789/0,
     ceph-node3=192.168.1.30:6789/0}, election epoch 30, quorum 0,1,2 ceph-node1,ceph-node2,
     ceph-node3
     osdmap e39: 9 osds: 9 up, 9 in
      pgmap v103: 192 pgs, 3 pools, 0 bytes data, 0 objects
            298 MB used, 45681 MB / 45980 MB avail
                 192 active+clean
```

  As we can see, the healh of our Cluster is **OK** and we have nine OSDs in and up.

- **Step 9. Monitoring a Cluster**

Now, once we have created our first Ceph Storage Cluster and we have it running, as we have just seen with the above command, we can use **ceph** tool to monitor our Cluster. Monitoring a Cluster typically involves checking OSD status, Placement Group status, Monitor status and Metadata server status:

- Monitoring OSDs An OSD's status can be either in the Cluster **in** or **out** of the Cluster, and also it can be either **up** and running, or it is **down** and not running. Also, if an OSD is **up**, it may be either **in** the Cluster so we can read and write data, or it can be **out** of the Cluster. If it was **in** the Cluster and recently moved **out** of the Cluster, Ceph will migrate **Placement Groups** to other OSDs. If an OSD is **out** of the Cluster, **CRUSH** will not assign **Placement Groups** to the OSD. If an OSD is **down**, it should also be **out** so our Cluster will not have a healthy state.

  Next figure shows the possible states of OSD daemons:

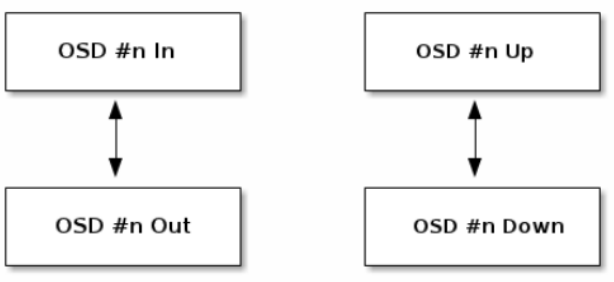


Figure 2.41: OSD stauts.

The command to monitor our OSDs is:

```
ceph osd stat
```

In our case we can see that the map epoch is **e39**, the total number of OSDs is **9** and that all are **up** and **in**

If with the above command we see that the number of OSDs that are in the Cluster is more than the number of OSDs that are **up**, we must execute the following command to identify the ceph-osd daemons that aren't running:

```
ceph osd tree
```

And the output should be, for example something like this:

```
dumped osdmap tree epoch 1
# id    weight  type name       up/down reweight
-1      2       pool openstack
-3      2               rack dell-2950-rack-A
-2      2                       host dell-2950-A1
0       1                               osd.0   up      1
1       1                               osd.1   down    1
```

If one of ours OSDs is down, like we can see in the above example, to start it we must use next command:

```
sudo /etc/init.d/ceph -a start osd.1
```

- Monitoring Placement Groups

  Sometimes, when we execute ceph health, ceph -s or ceph -w commands, we can notice that our cluster is not completely healthy, and this, if our OSDs are running, it can be caused by possible Placement Groups peering-related problems. These problems could be:

  - 1. We have just created a pool and Placement Groups haven't peered yet.

- 2. The Placement Groups are recovering.
- 3. We have just added an OSD to or removed an OSD from the cluster.
- 4. We have just modified our CRUSH map and our Placement Groups are migrating.
- 5. There is inconsistent data in different replicas of a placement group.
- 6. Ceph is scrubbing a Placement Group's replicas.

So as to see our PG status, we must should use the following command:

```
ceph pg stat
```

In our case, the result of the above command is:

```
v103: 192 pgs y active+clean; 3 pools, 0 bytes data, 0 objects 298 MB used, 45681 MB /
 45980 MB avail
```

As we can see, our state is **active + clean**, we have 3 pools but we used no objects.

- Monitor Status

When our Cluster has multiple monitors, like our case, we should check the monitor quorum status after we start the Cluster before reading and/or writing data. A quorum must be present when multiple monitors are running. We should also check monitor status periodically to ensure that they are running.

The possible commands to check monitor status are:

```
ceph mon stat
```

or

```
ceph mon dump
```

Also, to check the quorum status for the monitor cluster, we must execute the following:

```
ceph quorum_status
```

The output of our three monitors Cluster is:

```
{ "election_epoch": 30,
  "quorum": [
        0,
        1,
        2],
  "monmap": { "epoch": 3,
      "fsid": "6fac025f-1bfe-4a6c-b09a-5f7863cc447b",
      "modified": "2015-09-02 02:09:38:27.505520",
      "created": "2015-09-02 02:09:38:27.505520",
      "mons": [
            { "rank": 0,
              "name": "ceph-node1",
              "addr": "192.168.1.10:6789/0"},
            { "rank": 1,
              "name": "ceph-node2",
              "addr": "192.168.1.20:6789/0"},
            { "rank": 2,
              "name": "ceph-node3",
              "addr": "192.168.1.30:6789/0"}
           ]
    }
}
```

– Monitoring Metadata Server

Metadata servers provide Metadata services for Ceph File System, so in our case, as we don't have any Metadata server daemon running, we will not be able to monitorize it. But if we had one, we should now that the Metadata servers have two sets of states: **up | down** and **active | inactive**. To ensure our Metadata servers are **up** and **active**, we execute the following:

```
ceph mds stat
```

To display details of the metadata cluster, execute the following:

```
ceph mds dump
```

- **Summary of Ceph Cluster building using VirtualBox**

The software-defined nature of Ceph provides a great deal of flexibility to its adopters. Unlike other proprietary storage systems, which are hardware dependent, Ceph can be easily deployed and tested on almost any computer system available today. Moreover, if getting physical machines is a challenge, we can use like in this example, virtual machines using VirtualBox to install Ceph, but keeping in mint that such a setup should only be used for testing purposes.

In this chapter we learned how to create a set of VirtualBox Ubuntu machines, followed by Ceph deployment as a three-node cluster using the **ceph-deploy** tool. We also added a **nine OSDs** and **three monitor containers** to our cluster in order to demonstrate its dynamic scalability.

In next chapter we will see how we can create the same cluster using LXC containers virtualization technology.

2.10.1.6.5.2 Ceph Cluster with LXC

As explained above, in this chapter we will see how to create a set of containers using LXC, followed by Ceph deployment as a three-node Cluster using the **ceph-deploy** tool. To do it we must follow next steps:

- **Step 1. Install LXC on Ubuntu.**

To install LXC and everything needed by LXC we use the next command:

```
# sudo apt-get install lxc lxctl lxc-templates
```

To check if everything is OK:

```
# sudo lxc-checkconfig

Kernel configuration not found at /proc/config.gz; searching...
Kernel configuration found at /boot/config-3.16.0-41-generic
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: enabled
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled
```

```
--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled

Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /usr/bin/lxc-checkconfig
```

Now, we can create a new linux container or we can use one from the LXC templates. To list all available templates, enter the following command:

```
#sudo ls /usr/share/lxc/templates/

lxc-alpine   lxc-busybox   lxc-debian        lxc-gentoo lxc-oracle    lxc-ubuntu lxc-altlinux
lxc-centos   lxc-download  lxc-openmandriva  lxc-plamo  lxc-ubuntu-cloud        lxc-archlinux
lxc-cirros   lxc-fedora    lxc-opensuse      lxc-sshd
```

In our case we will use the **lxc-ubuntu** template.

- **Step 2. Create new Ubuntu Containers**

  Next we will create three containers with the next names:

  - 1. **ceph-node1**: node from which we will deploy Ceph to the other nodes, and also will monitor the cluster and will have a OSD module. IP 10.0.0.2
  - 2. **ceph-node2**: node with the OSD and Monitor module. IP 10.0.0.3
  - 3. **ceph-node3**: node with the OSD and Monitor module. IP 10.0.0.4

  Next, we will see an example of how to create the **ceph-node1** container. The command should be as below but changing **<container-name>** and **<template>** for the correct names.

```
sudo lxc-create -n <container-name> -t <template>
```

Sample of **ceph-node1** container creating:

```
# sudo lxc-create -n ceph-node1 -t ubuntu
Checking cache download in /var/cache/lxc/trusty/rootfs-i386 ...
Copy /var/cache/lxc/trusty/rootfs-i386 to /var/lib/lxc/ceph-node1/rootfs ...
Copying rootfs to /var/lib/lxc/ceph-node1/rootfs ...
Generating locales...
  en_US.UTF-8... up-to-date
Generation complete.
.
.
.
perl: warning: Falling back to the standard locale ("C").
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
        LANGUAGE = "en_US",
        LC_ALL = (unset),
        LC_PAPER = "es_ES.UTF-8",
        LC_ADDRESS = "es_ES.UTF-8",
        LC_MONETARY = "es_ES.UTF-8",
        LC_NUMERIC = "es_ES.UTF-8",
        LC_TELEPHONE = "es_ES.UTF-8",
        LC_IDENTIFICATION = "es_ES.UTF-8",
        LC_MEASUREMENT = "es_ES.UTF-8",
        LC_TIME = "es_ES.UTF-8",
        LC_NAME = "es_ES.UTF-8",
```

```
        LANG = "en_US.UTF-8"
    are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
locale: Cannot set LC_ALL to default locale: No such file or directory
Current default time zone: 'Europe/Madrid'
Local time is now:      Tue Jul 14 21:55:12 CEST 2015.
Universal Time is now:  Tue Jul 14 19:55:12 UTC 2015.

##
# The default user is 'ubuntu' with password 'ubuntu'!
# Use the 'sudo' command to run tasks as root in the container.
##
```

To create the other two containers we must repeat the above command changing the container name.

- **Step 3. Containers list**

  After creating the three ubuntu containers we can view the our containers list with:

```
# sudo lxc-ls
ceph-node1       ceph-node2       ceph-node3
```

- **Step 4. Starting and configuring containers network**

  All the following commands are used with **ceph-node1** container. To configure the other two containers we must use the same commands but changing the container name.

  To start a container we use the following command.

```
sudo lxc-start -n ceph-node1 -d
```

  Now we log in to the container with next command:

```
sudo lxc-console -n ceph-node1

ceph-node1 login: ubuntu
Password:
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-41-generic i686)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

  In this case we must use the default **username** and **password** that were generated while creating the new containers. In our case, the default **username** is **ubuntu**, and password is **ubuntu**.

  To view the complete details of **ceph-node1** container:

```
sudo lxc-info -n ceph-node1
```

  To **exit** from the container's console, and return back to our original host computer terminal, we must press **ctrl+a** followed by the letter **q** from our keyboard.

  Next we will configure the ip of the lxc containers.

  From our host terminal, with the **ifconfig -a** command we will see the rang of Ip's that our containers will have after creating them with the **lxc-ubuntu** container template:

```
# sudo ifconfig -a
eth0      Link encap:Ethernet  HWaddr 90:e6:ba:71:0f:b4
          inet addr:192.168.1.37  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::92e6:baff:fe71:fb4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:113217 errors:0 dropped:0 overruns:0 frame:0
          TX packets:100767 errors:0 dropped:0 overruns:0 carrier:1
          collisions:0 txqueuelen:1000
          RX bytes:126542781 (126.5 MB)  TX bytes:8315717 (8.3 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:5571 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5571 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:540397 (540.3 KB)  TX bytes:540397 (540.3 KB)

lxcbr0    Link encap:Ethernet  HWaddr 56:56:3c:65:07:ba
          inet addr:10.0.3.1  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::5456:3cff:fe65:7ba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:47 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:7639 (7.6 KB)
```

From the output we can see a new NIC called **lxcbr0**. This is the the default LXC bridge created when we install LXC package.

As we can see, the **lxcbr0** interface has the **IP 10.0.3.1**, so our created containers will have IPs in the same range (**10.0.3.2 – 10.0.3.254**). To change this range so as to set the IPs that we explained before we must change **/etc/default/lxc-net** and **/etc/init/lxc-net.conf** files.

But first, we must stop the **lxc-net** service with the next command:

```
sudo service lxc-net stop
```

Then, using the **vi** text editor we change the **/etc/default/lxc-net** file as follows:

```
# sudo vi /etc/default/lxc-net
[sudo] password for marius:
# This file is auto-generated by lxc.postinst if it does not
# exist.  Customizations will not be overridden.
# Leave USE_LXC_BRIDGE as "true" if you want to use lxcbr0 for your
# containers.  Set to "false" if you'll use virbr0 or another existing
# bridge, or mavlan to your host's NIC.
USE_LXC_BRIDGE="true"

# If you change the LXC_BRIDGE to something other than lxcbr0, then
# you will also need to update your /etc/lxc/default.conf as well as the
# configuration (/var/lib/lxc/<container>/config) for any containers
# already created using the default config to reflect the new bridge
# name.
# If you have the dnsmasq daemon installed, you'll also have to update
# /etc/dnsmasq.d/lxc and restart the system wide dnsmasq daemon.
LXC_BRIDGE="lxcbr0"
LXC_ADDR="10.0.0.1"
LXC_NETMASK="255.255.255.0"
LXC_NETWORK="10.0.0.0/24"
LXC_DHCP_RANGE="10.0.0.2,10.0.0.254"
LXC_DHCP_MAX="253"
```

```
# Uncomment the next line if you'd like to use a conf-file for the lxcbr0
# dnsmasq.  For instance, you can use 'dhcp-host=mail1,10.0.0.100' to have
# container 'mail1' always get ip address 10.0.0.100.
#LXC_DHCP_CONFILE=/etc/lxc/dnsmasq.conf

# Uncomment the next line if you want lxcbr0's dnsmasq to resolve the .lxc
# domain.  You can then add "server=/lxc/10.0.0.1' (or your actual )
# to /etc/dnsmasq.conf, after which 'container1.lxc' will resolve on your
# host.
#LXC_DOMAIN="lxc"
```

Next we do the same with the **/etc/init/lxc-net.conf** file.

Once we changed both files, we restart the **lxc-net** service as follows:

```
sudo service lxc-net restart
```

Now, using again the **ifconfig -a** command we will see that our **lxcbr0** interface has the desired IP **10.0.0.1**:

```
# sudo ifconfig -a
eth0      Link encap:Ethernet  HWaddr 90:e6:ba:71:0f:b4
          inet addr:192.168.1.37  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::92e6:baff:fe71:fb4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:113217 errors:0 dropped:0 overruns:0 frame:0
          TX packets:100767 errors:0 dropped:0 overruns:0 carrier:1
          collisions:0 txqueuelen:1000
          RX bytes:126542781 (126.5 MB)  TX bytes:8315717 (8.3 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:5571 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5571 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:540397 (540.3 KB)  TX bytes:540397 (540.3 KB)

lxcbr0    Link encap:Ethernet  HWaddr 56:56:3c:65:07:ba
          inet addr:10.0.0.1  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::5456:3cff:fe65:7ba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:47 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:7639 (7.6 KB)
```

Next we must change IPs in each container so as to set the one we explained before.

To do so, we must follow the next steps. For example in **ceph-node1** container:

- 1. We set down the **eth0**:

```
sudo ifdown eth0
```

- 2. We change the the **eth0** interface in **/etc/network/interfaces** file from **dhcp** to a **static** ip:

```
# sudo cat /etc/network/interfaces
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
```

```
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 10.0.0.2
        network 10.0.0.0
        netmask 255.255.255.0
        broadcast 10.0.0.255
        gateway 10.0.0.1

        dns-nameservers 8.8.8.8 8.8.4.4
```

– 3. We turn up the **eth0** and restart the networking service:

```
sudo ifup eth0
sudo /etc/init.d/networking restart
```

Now we have our desired ip for the **ceph-node1** container:

```
# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:16:3e:a5:95:36
          inet addr:10.0.0.2  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::216:3eff:fea5:9536/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:91 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10258 (10.2 KB)  TX bytes:5284 (5.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Next, so as to set the correct IPs for the other containers that we have created, we must follow the same steps as explained for the **ceph-node1** container.

Once we have correctly configured the network of each container, from our host terminal we can see the final results using the next command:

```
# sudo lxc-ls --fancy ceph-node1
NAME        STATE    IPV4      IPV6  AUTOSTART
----------------------------------------------
ceph-node1  RUNNING  10.0.0.2  -     NO

# sudo lxc-ls --fancy node2
NAME   STATE    IPV4  IPV6  AUTOSTART
-----------------------------------
ceph-node2  RUNNING  10.0.0.3     -     NO

# sudo lxc-ls --fancy node3
NAME   STATE    IPV4      IPV6  AUTOSTART
-----------------------------------------
ceph-node3  RUNNING  10.0.0.4  -     NO
```

Next step is changing the **/etc/hosts** file in all containers so as to include the hosts of each container. For example, in **ceph-node1** container:

```
# sudo cat /etc/hosts
127.0.0.1   localhost
127.0.1.1   ceph-node1

10.0.0.2 ceph-node1
10.0.0.3 ceph-node2
10.0.0.4 ceph-node3

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Once we have completed the network configuration in all three nodes, the next step is to accomplish the software prerequisites explained in **Prior Software configurations** section.

- **Step 5. Set the Prior Software Configurations**

  Now, we must repeat the steps explained in **Prior Software configurations** section. Once we have added the needed release packages (**Firefly (0.80)**) and created our ceph user we proceed to install packages.

- **Step 6. Create the cluster**

  Now, in our **ceph-node1** container, from the folder that we have created in the above section for holding our configuration details, we will perform the following steps using **ceph-deploy** in order to create the cluster:

```
ceph-deploy new ceph-node1}
```

  Now, in our **my-cluster** directory, with the **ls** command, we should see a Ceph configuration file, a monitor keyring and a log file for the new cluster.

  In this example we will use only two Ceph OSDs, one installed in **ceph-node2** and the other in **ceph-node3**, and for this reason, first of all we must change the default number of replicas in the Ceph configuration file from **3** to **2** so that Ceph can achieve an **active + clean** state with just two Ceph OSDs. In the **ceph-node1**, we must add the following line under the **[global]** section in the Ceph configuration file in our **my-cluster** directory:

```
osd pool default size = 2
```

  The next step is installing Ceph in our three nodes:

```
ceph-deploy install ceph-node1 ceph-node2 ceph-node3
```

- **Step 7. Adding our first Monitor**

  We will add the Monitor module in **ceph-node1** node with the following command:

```
ceph-deploy mon create-initial
```

  Now, once we have the cluster initiated, in our directory we should see 4 keyrings like these ones:

  - **cluster-name.client.admin.keyring**
  - **cluster-name**.bootstrap-osd.keyring
  - **cluster-name**.bootstrap-mds.keyring
  - **cluster-name**.bootstrap-rgw.keyring

- **Step 8. Installing our OSD modules**

  Once we have the cluster up, in order to install our OSD modules we will use a quick start way for fast setup. We will use a directory rather than an entire disk per Ceph OSD Daemon.

  From the **ceph-node1** node or directly in the **ceph-node2** and **ceph-node3** we will create a new directory for the Ceph OSD Daemon. The following example is from **ceph-node1** container:

```
ssh ceph-node2
sudo mkdir /var/local/osd0
exit

ssh ceph-node3
sudo mkdir /var/local/osd1
exit
```

  Then, from the **ceph-node1** node we will use **ceph-deploy** so as to prepare the OSDs: from **ceph-node1** node:

```
ceph-deploy osd prepare ceph-node2:/var/local/osd0 ceph-node3:/var/local/osd1
```

  Then, to activate the OSDs:

```
ceph-deploy osd activate ceph-node2:/var/local/osd0 ceph-node3:/var/local/osd1
```

  Next, so that we can use the ceph CLI without having to specify the monitor address and **ceph.client.admin.keyring** each time we execute a command, from our **ceph-node1** we must following command:

```
ceph-deploy admin ceph-node1 ceph-node2 ceph-node3
```

  And to ensure that we have the correct permissions for the **ceph.client.admin.keyring**:

```
sudo chmod +r /etc/ceph/ceph.client.admin.keyring
```

  Now, if everything was correct, checking our cluster's health using the following command we should get an active + clean cluster state.

```
ceph health
```

2.10.1.6.5.3 Ceph Storage Cluster with Docker

As explained in Docker section, the **Dockerfiles** are used to form a new docker image and replace the process of doing everything manually and repeatedly. So, as there already exists Dockerfiles with the Ceph Storage Cluster deployed, the purpose of this section is to see which are our tools and what can we do if we want to play with our Ceph Storage Cluster using Docker virtualization technology.

As both Ceph and Docker are new technologies, some of their components are still in development stage, and because of this they are not running at full performance. One of the problems that we will see next is that our Ceph Monitor can only be reached by the host itself. So this problem makes our Cluster useless.

Next we will detail the existing Ceph-related Dockerfiles and a short example of their usage:

CORE COMPONENTS:

- **ceph/base**: This is the Ceph base container image, a fresh install of the latest Ceph (firefly 0.80) on Ubuntu LTS (14.04).

  Usage example:

```
$ docker run -i -t ceph/b
```

- **ceph/mds**: This Dockerfile help us to create a Ceph Metadata Server (MDS).

  Usage example:

  In this case we will need to use the environment variable **MDS_NAME**, that describes the name of the the MDS.

  For example:

```
docker run -e MDS_NAME=mymds ceph/mds
```

  It will look for either **/etc/ceph/ceph.client.admin.keyring** or **/etc/ceph/ceph.mds.keyring** with which to authenticate. We can get **ceph.client.admin.keyring** from another ceph node.

  Commonly, we will want to bind-mount our host's **/etc/ceph** into the container.

  For example:

```
docker run -e MDS_NAME=mymds -v /etc/ceph:/etc/ceph ceph/mds
```

  CephFS:

  We must consider that, by default, the MDS does NOT create a ceph filesystem. So, if we want to have a ceph filesystem created with MDS, we must set **CEPHFS_CREATE1** in the **ceph/mds** Dockerfile.

  There are also other variable to consider if we wish to customize the data and metadata pools in which our CephFS is stored:

  - **CEPHFS_CREATE**: Whether to create the ceph filesystem (0 = no 1 = yes), if it doesn't exist. Default is 0 (no).
  - **CEPHFS_NAME**: The name of the new ceph filesystem and the basis on which the later variables are created.
  - **CEPHFS_DATA_POOL**: The name of the data pool for the ceph filesystem. If it does not exist, it will be created. Defaults to ${CEPHFS_NAME}_data.
  - **CEPHFS_DATA_POOL_PG**: The number of placement groups for the data pool. Defaults to 8.
  - **CEPHFS_METADATA_POOL**: The name of the metadata pool for the ceph filesystem. If it does not exist, it will be created. Defaults to ${CEPHFS_NAME}_metadata.
  - **CEPHFS_METADATA_POOL_PG**: The number of placement groups for the metadata pool. Defaults to 8.

  NOTE:

  There is a problem with **ceph/mds** because it seems to die most of the time when trying to start it. Trying repeatedly will eventually get it to run. If run manually within the container, it fails less often, but still frequently.

- **ceph/mon**:

  This Dockerfile may be used to bootstrap a Ceph cluster or add a Monitor to an existing cluster.

  Usage

  We have the following environment variables:

  - **MON_NAME**: is the name of the monitor (defaults to hostname -s).
  - **MON_IP**: is the IP address of the monitor (public). Is required if we do not use autodetection.

– **MON_IP_AUTO_DETECT**: Whether and how to attempt IP autodetection. 0 D̄o not detect (default) 1 D̄etect IPv6, fallback to IPv4 (if no globally-routable IPv6 address detected) 4 D̄etect IPv4 only 6 D̄etect IPv6 only

For example:

```
docker run -e MON_IP=192.168.101.50 -e MON_NAME=mymon ceph/mon
```

If we have an existing Ceph cluster and we are only looking to add a monitor to it, we will need at least four files in the **/etc/ceph**:

– **ceph.conf**: The main ceph configuration file, which may be obtained from an existing ceph monitor.
– **ceph.client.admin.keyring**: The administrator key of the cluster, which may be obtained from an existing ceph monitor by **ceph auth get client.admin o /tmp/ceph.client.admin.keyring**
– **ceph.mon.keyring**: The monitor key, which may be obtained from an existing ceph monitor by **ceph auth get mon. o /tmp/ceph.mon.keyring**.
– **monmap** The present monitor map of the cluster, which may be obtained from an existing ceph monitor by **ceph mon getmap o /tmp/monmap**.

Otherwise, if we are bootstrapping a new cluster, these will be generated for us.

Commonly, we will want to bind-mount our host's **/etc/ceph** into the container.

For example:

```
docker run -e MON_IP=192.168.101.50 -e MON_NAME=mymon -v /etc/ceph:/etc/ceph ceph/mon
```

NOTE:

For now, running Ceph Monitor in Docker it's not usable since nothing can reach the Monitor except the host itself. Thus, other Ceph components will only work if they share the same network namespace as the Monitor. Sharing all the containers namespace into one could quite difficult as well. But it makes no sense to have a Ceph Cluster stuck within some namespaces, without any clients accessing it.

- **ceph/osd**: It helps us to run the Ceph OSD in Docker.

  Usage:

  In order to configure the execution of the OSD, we must know the environment variables that will help us to do it:

  – **CLUSTER**: is the name of the ceph cluster (defaults to ceph).

  If the OSD is not already created (key, configuration, OSD data), the following environment variables will control its creation:

  – **WEIGHT**: is the of the OSD when it is added to the CRUSH map (default is 1.0)
  – **JOURNAL**: is the location of the journal (default is the journal file inside the OSD data directory)
  – **HOSTNAME**: is the name of the host; it is used as a flag when adding the OSD to the CRUSH map

  To create our OSDs we must simply run the following command:

```
docker exec <mon-container-id> ceph osd create.
```

NOTE: There is a problem when attempting run run multiple OSD containers on a single docker host. There are two workarounds:

- Run each OSD with the pid_host option
- Run multiple OSDs within the same container (this is the default, if the OSD directories are present)

To run multiple OSDs within the same container, simply bind-mount each OSD datastore directory:

```
docker run -v /osds/1:/var/lib/ceph/osd/ceph-1 -v /osds/2:/var/lib/ceph/osd/ceph-2
```

Shared and/or separate journal

The default journal location for each OSD in this Docker container is **/var/lib/ceph/osd/journal/journal-<OSD_ID>/**. This means that if we like to have our journals (optionally shared) in a separate disk, we must mount that separate disk to the container's **/var/lib/ceph/osd/journal/** directory.

An easy way to have this all handled properly is to mount our journal and each OSD to their respective locations in our host's **/var/lib/ceph/osd** tree and make that entire tree available to this container by passing **v /var/lib/ceph/osd:/var/lib/ceph/osd** to the **docker run** execution.

BTRFS and journal

If our OSD is BTRFS and we want to use PARALLEL journal mode, we will need to run this Docker container with **-privileged** set to true. Otherwise, **ceph-osd** will have insufficient permissions and it will revert to the slower **WRITEAHEAD** mode.

NOTE

As the OSDs use a great many ports during periods of high traffic, it is possible that OSD craches frequently due to Docker running out of sufficient open file handles. To solve this, it is recommended to increase the number of open file handles available to Docker.

UTILITIES:

- **ceph/config**: Initializes and distributes cluster configuration.

  This Dockerfile may be used to bootstrap a cluster or add the cluster configuration to a new host. It uses etcd to store the cluster config. It is especially suitable to setup ceph on CoreOS.

  The following strategy is applied:

  - An /etc/ceph/ceph.conf is found, do nothing.
  - If a cluster configuration is available, it will be written to **/etc/ceph**
  - If no cluster configuration is available, it will be bootstrapped. A lock mechanism is used to allow concurrent deployment of multiple hosts.

  Usage

  To bootstrap a new cluster run:

```
docker run -e ETCDCTL_PEERS=http://192.168.101.50:4001 -e MON_IP=192.168.101.50
 -e MON_NAME=mymon -e CLUSTER=testing -v /etc/ceph:/etc/ceph ceph/config
```

  The above commando will generate the following files:

  - **ceph.conf**
  - **ceph.client.admin.keyring**
  - **ceph.mon.keyring**
  - **monmap**

Except the **monmap** the config will be stored in etcd under **/ceph-config/$CLUSTER**.

In case a configuration for the cluster is found, the configuration will be pulled from etcd and written to **/etc/-ceph**.

Multiple concurrent invocations will block until the first host finished to generate the configuration.

- **ceph/docker-registry**: Rados backed docker-registry images repository.

  Sample output:

```
docker run -d \
         --name ceph-docker-registry
         -e AWS_BUCKET=mybucket \
         -e AWS_KEY=myawskey \
         -e AWS_SECRET=myawssecret \
         -e AWS_HOST=myowns3.com \
         -e AWS_SECURE=true \
         -e AWS_PORT=80 \
         -e AWS_CALLING_FORMAT=boto.s3.connection.OrdinaryCallingFormat \
         -p 5000:5000 \
         ceph/docker-registry
```

- **ceph/rados**: Convenience wrapper to execute the rados CLI tool.

  This is a convenience container to execute ceph rados for object manipulation.

  We must make sure to pass our **/etc/ceph** path as a volume/bind-mount.

  Example:

```
docker run -v /etc/ceph:/etc/ceph ceph/rados lspools
```

- **ceph/rbd**: Convenience wrapper to execute the rbd CLI tool.

  Example:

```
docker run -v /etc/ceph:/etc/ceph ceph/rbd -p vms ls
```

- **ceph/rbd-lock**: Convenience wrapper to block waiting for an rbd lock.

  Example:

```
docker run -v /etc/ceph:/etc/ceph ceph/rbd -p vms ls
```

- **ceph/rbd-unlock**: Convenience wrapper to release an rbd lock.

  Example:

```
docker run --rm -v /etc/ceph:/etc/ceph ceph/rbd-unlock myPool/myImage myLockName lockId
```

- **ceph/rbd-volume**: Convenience wrapper to mount an rbd volume. This Docker container will mount the requested RBD image to a volume. We can then use link to that volume from other containers with the **–volumes-from** Docker **run** option.

  Example usage

```
docker run --name myData -e RBD_IMAGE=myData -e RBD_POOL=myPool ceph/rbd-volume
```

```
docker run --volumes-from myData myOrg/myApp
```

2.10.1.6.6 Operating the Cluster

In this section we will see the available commands so as to operate our Ceph Storage Cluster.

2.10.1.6.6.1 Running Ceph with Upstart

When deploying the new Ceph versions (like firefly) using the **ceph-deploy** tool on Ubuntu, we may start and stop Ceph daemons on a Ceph Node using the event-based Upstart. Upstart is an event-based replacement for the **/sbin/init** daemon which handles starting of tasks and services during boot, stopping them during shutdown and supervising them while the system is running. Upstart does not require to define daemon instances in the Ceph configuration file.

To list the Ceph Upstart jobs and instances on a node, we execute:

```
sudo initctl list | grep ceph
```

Starting all Daemons

To start all daemons on a Ceph Node, we execute the following:

```
sudo start ceph-all
```

Stopping all Daemons

To stop all daemons on a Ceph Node, we execute the following:

```
sudo stop ceph-all
```

Starting all Daemons by Type

To start all daemons of a particular type on a Ceph Node, execute must execute one of the following:

```
sudo start ceph-osd-all
sudo start ceph-mon-all
sudo start ceph-mds-all
```

Stopping all Daemons by Type

To stop all daemons of a particular type on a Ceph Node, execute one of the following:

```
sudo stop ceph-osd-all
sudo stop ceph-mon-all
sudo stop ceph-mds-all
```

Starting a Daemon

To start a specific daemon instance on a Ceph Node, execute one of the following:

```
sudo start ceph-osd id={id}
sudo start ceph-mon id={hostname}
sudo start ceph-mds id={hostname}
```

For example:

```
sudo start ceph-osd id=1
sudo start ceph-mon id=ceph-server
sudo start ceph-mds id=ceph-server
```

Stopping a Daemon

To stop a specific daemon instance on a Ceph Node, execute one of the following:

```
sudo stop ceph-osd id={id}
sudo stop ceph-mon id={hostname}
sudo stop ceph-mds id={hostname}
```

For example:

```
sudo stop ceph-osd id=1
sudo start ceph-mon id=ceph-server
sudo start ceph-mds id=ceph-server
```

2.10.1.6.6.2 Running Ceph

Each time we start, restart, and stop Ceph daemons (or the entire cluster) we must specify at least one option and one command. We may also specify a daemon type or a daemon instance:

```
{commandline} [options] [commands] [daemons]
```

The ceph options include:

```
Option  Shortcut        Description
--verbose       -v      Use verbose logging.
--valgrind      N/A     (Dev and QA only) Use Valgrind debugging.
--allhosts      -a      Execute on all nodes in ceph.conf. Otherwise, it only executes on localhost.
--restart       N/A     Automatically restart daemon if it core dumps.
--norestart     N/A     Don't restart a daemon if it core dumps.
--conf  -c      Use an alternate configuration file.
```

The ceph commands include:

```
Command         Description
start   Start the daemon(s).
stop    Stop the daemon(s).
forcestop       Force the daemon(s) to stop. Same as kill -9
killall         Kill all daemons of a particular type.
cleanlogs       Cleans out the log directory.
cleanalllogs    Cleans out everything in the log directory.
```

For subsystem operations, the ceph service can target specific daemon types by adding a particular daemon type for the [daemons] option. Daemon types include **mon**, **osd** and **mds**.

Running Ceph with sysvinit

Using traditional **sysvinit** is the recommended way to run Ceph with CentOS, Red Hat, Fedora, SLES, Debian, and also with older Ubuntu distributions.

Starting all Daemons

To start our Ceph cluster, we execute ceph with the start command. Use the following syntax:

```
sudo /etc/init.d/ceph [options] [start|restart] [daemonType|daemonID]
```

The following examples illustrates a typical use case:

```
sudo /etc/init.d/ceph -a start
```

Once we execute with **-a** (i.e., execute on all nodes), Ceph should begin operating.

Stopping all Daemons

To stop our Ceph cluster, we execute ceph with the stop command. Use the following syntax:

```
sudo /etc/init.d/ceph [options] stop [daemonType|daemonID]
```

The following examples illustrates a typical use case:

```
sudo /etc/init.d/ceph -a stop
```

Once we execute with **-a** (i.e., execute on all nodes), Ceph should stop operating. Starting all Daemons by Type
To start all Ceph daemons of a particular type on the local Ceph Node, we use the following syntax:

```
sudo /etc/init.d/ceph start {daemon-type}
sudo /etc/init.d/ceph start osd
```

To start all Ceph daemons of a particular type on another node, we use the following syntax:

```
sudo /etc/init.d/ceph -a start {daemon-type}
sudo /etc/init.d/ceph -a start osd
```

Stopping all Daemons by Type
To stop all Ceph daemons of a particular type on the local Ceph Node, we use the following syntax:

```
sudo /etc/init.d/ceph stop {daemon-type}
sudo /etc/init.d/ceph stop osd
```

To stop all Ceph daemons of a particular type on another node, we use the following syntax:

```
sudo /etc/init.d/ceph -a stop {daemon-type}
sudo /etc/init.d/ceph -a stop osd
```

Starting a Daemon
To start a Ceph daemon on the local Ceph Node, we use the following syntax:

```
sudo /etc/init.d/ceph start {daemon-type}.{instance}
sudo /etc/init.d/ceph start osd.0
```

To start a Ceph daemon on another node, we use the following syntax:

```
sudo /etc/init.d/ceph -a start {daemon-type}.{instance}
sudo /etc/init.d/ceph -a start osd.0
```

Stopping a Daemon
To stop a Ceph daemon on the local Ceph Node, we use the following syntax:

```
sudo /etc/init.d/ceph stop {daemon-type}.{instance}
sudo /etc/init.d/ceph stop osd.0
```

To stop a Ceph daemon on another node, we use the following syntax:

```
sudo /etc/init.d/ceph -a stop {daemon-type}.{instance}
sudo /etc/init.d/ceph -a stop osd.0
```