Evaluating Ceph Deployments with Rook

by

Rubab Zahra Sarfraz

Supervised by

Dan Van Der Ster

Julien Collet

September 27, 2018

Contents

Chapter 1

Introduction

1.1 Ceph at CERN

Ceph is an open source distributed storage system that is designed for excellent reliability, performance and scalability. It was founded in 2006 and since then it has been adopted by a large number of tech giants such as RedHat, Open Suse, CERN, Cisco, SanDisk, Fujitsu etc. One of the main reason for its popularity is it provides its user with object storage, block storage and file storage all in one. It is a de facto persistent storage for block devices used by OpenStack and hence it becomes a first choice for vendors using OpenStack. At CERN, in addition to being large-scale Ceph users, people at CERN are also active contributors to Ceph components. CERN has the largest cluster of Ceph deployed in its data center. It is used for several storage use-cases such as, for:

- OpenStack Images and Volumes (RBD)
- HPC scratch spaces (CephFS)
- Private NFS-like file shares (CephFS)
- Object storage compatible with Amazon S3 (RGW)

CERN has to deal with petabytes of data so it is always on the look out for ways to simplify its cloud-based deployments. It has been actively evaluating container-based approaches that build upon its Kubernetes infrastructure. One such technology that recently caught attention was Rook; a storage orchestrator for cloud-native environments.

1.2 Motivation and Goal

Although, Ceph has been satisfying its storage use-cases for years now, it would be a cherry on the top if somehow its operational complexities could be reduced. CERN currently uses a combination of kickstart and puppet to configure and manage Ceph deployments which poses some time constraints. With the recent Rook Ceph integration being in Beta stage, it made sense to evaluate it to see if future Ceph clusters could be deployed using this tool. To be precise, the following areas were hoped to be improved by Rook:

- Reduced deployment times for new clusters.
- Simplified upgrades.
- More agile horizontal scaling.
- Better failure tolerance.
- Reduced reliance on expert Ceph operators

1.2.1 Problem Statement

In order to address above mentioned improvements, the idea is to deploy Ceph containers on Kubernetes cluster using Rook and to decide whether we should adopt this stack for our future clusters here at CERN or not. To be precise, the following steps were involved in this project:

- Deploy Kubernetes clusters in virtual environment
- Deploy Kubernetes cluster in physical environment
- Deploy Ceph clusters using Rook in virtual as well as physical environments using stable versions
- Deploy Ceph clusters using development code to access orchestrator command line interface
- Establish an evaluation criteria for deployments
- Compare results with previous deployment technique
- Explore further storage use-cases at CERN that are needed in orchestrator CLI
- Contribute open source code to meet those storage use-cases

1.2.2 Report Organization

Chapter 2 briefly covers introduction and architecture of main technologies used in this project. Chapter 3 includes the detailed deployment process followed in this project. Chapter 4 talks about evaluation metrics and results followed by chapter 5 which covers conclusion and future work. Chapter 6 gives a list of solutions to problems faced in this project.

Chapter 2

Background Theory

In order to deploy a Rook cluster, it is necessary to gain some background knowledge about containers, Ceph, Kubernetes, Rook and all the theoretical concepts related to these, especially relevant to storage. In this section, these concepts are explained one by one.

2.1 Kubernetes

Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF) [4].

The Kubernetes project started in the year 2014 with more than a decade of experience of running production workloads at Google with Google's internal container cluster managers (Borg and Omega). It has now become the de facto standard for deploying containerized applications at scale in private, public and hybrid cloud environments. The largest public cloud platforms AWS, Google Cloud, Azure, IBM Cloud and Oracle Cloud now provide managed services for Kubernetes. An important reason for its popularity is that a user does not have to change his/her application code for making it run in a cloud. A Kubernetes user gets the freedom to decide as well as define how the applications should run and communicate. The user is also allowed to scale up/down services, perform rolling updates, switch traffic between different application versions, and more. Kubernetes also offers different interfaces and platform primitives for defining/managing applications.

2.1.1 Kubernetes Architecture

On the high level, any application that runs on VMs can be deployed on Kubernetes by simply containerizing its components. This is achieved by its core features; container grouping, container orchestration, overlay networking, container-to-container routing with layer 4 virtual IP based routing system, service discovery, support for running daemons, deploying stateful application components, and most importantly the ability to extend the container orchestrator for supporting complex orchestration requirements.

Kubernetes provides a set of dynamically scalable hosts for running workloads using containers and uses a set of management hosts called masters for providing an API for managing the entire container infrastructure as it can be seen in figure 2.1. The workloads could include long-running services, batch jobs and container host specific daemons. All the container hosts are connected together using an overlay network for providing container-to-container routing. Applications deployed on Kubernetes are dynamically discoverable within the cluster network and can be exposed to the external networks using traditional load balancers. The state of the cluster manager is stored on a highly distributed key/value store **etcd** which runs within the master instances.

Kubernetes scheduler that runs on master hosts will always make sure that each application component is health checked, provided high availability, when the number of replicas is set to more than one each instance is scheduled in multiple hosts, and if one of those hosts becomes unavailable all the containers which were running in that host are scheduled in any of the remaining hosts. One of the fascinating capabilities Kubernetes offers is two level autoscaling. 1) First, it provides the ability to autoscale containers using a resource called `Horizontal Pod Autoscaler` which watches the resource consumption and scale the number of containers needed accordingly. 2) Second, it can scale the container cluster itself by adding and removing hosts depending on the resource requirements. Moreover, with the introduction of the cluster federation capability, it can even manage a collection of Kubernetes clusters which may span over multiple data centers using a single API endpoint.

2.1.2 Kubernetes Components

Kubernetes is composed of various software and hardware components. It must be noted here that it is not dependent on any kind of hardware but some basic resources are needed to provision the cluster. Following is a brief introduction of the components relevant to this project.

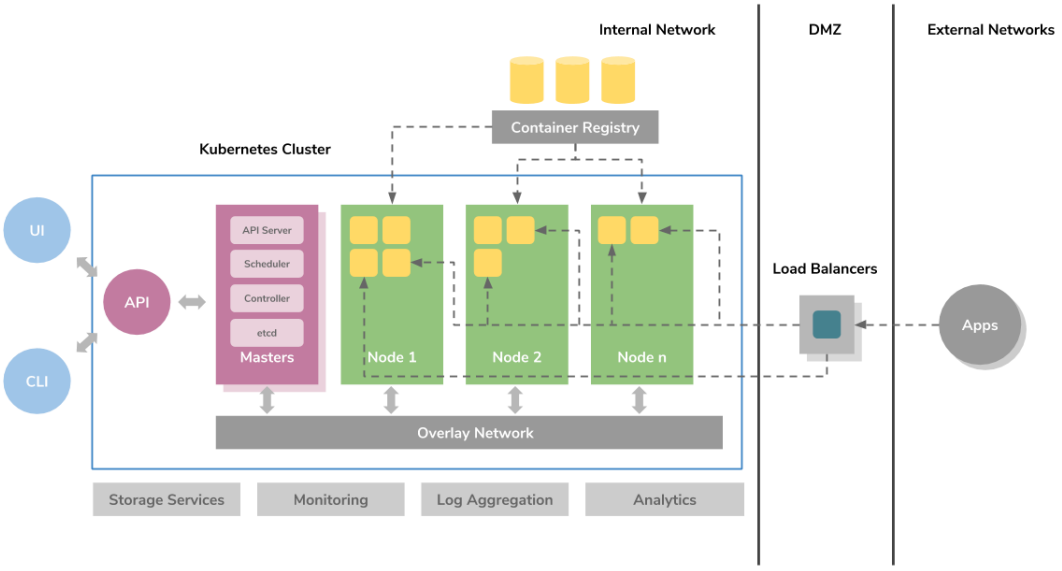

FIGURE 2.1: Kubernetes Architecture

- **Containers**

  A container is a lighter version of virtual machine that is used to deploy applications. The programs, as well as its dependencies, are packed in one single file and shared on the internet. So anyone can download the container and deploy it on their infrastructure as per the requirement. Deployment is hassle-free with just a little setup. In Kubernetes, Linux containers host the programs. These containers are globally accepted and already have pre-built images. The images can be deployed on Kubernetes.

  The containers are capable of handling multiple programs. But it is recommended to limit one process per container because it helps in troubleshooting. Updating the containers is easy and the deployment is easy if it is small. It is better to have many small containers, rather than a big one.

- **Pods**

  Kubernetes has some unique features and one of them is that it does not run the containers directly. It rather wraps up one or more containers into a pod. The concept of a `pod` is that any containers within the same pod use the same resources and the same local network. The benefit is that the containers can communicate with each other easily. They are isolated but are readily available for communication. The `pods` can replicate in Kubernetes. For example, an application becomes popular and a single `pod` is not able to sustain the load. At that moment, Kubernetes can be configured for deploying new replicas of the pod as per the requirement. But it is not necessary that replication occurs only during heavy load. A `pod` can

replicate during normal conditions as well. This helps in uniform load balancing and resisting failures.

`Pods` are capable of holding multiple containers but one should limit one or two if possible. The reason is that the `pods` scale up and down as a single unit. The containers within the pod must also scale together with the pods. Their individual needs are not important at this stage. On the other side, this leads to wastage of resources and expensive bills.

- **Deployment**
  Although, `pods` are the basic units in Kubernetes but they are not launched directly on a cluster. They are managed by more than one layer of abstraction. This overall makes `deployment`. The main purpose is to declare the number of replicas running at a time. When the `deployment` is added, it spins up the number of `pods` and monitors them. Similarly, if the `pod` does not exist anymore, `deployment` re-creates it. An interesting thing with `deployment` is that there is no need to deal with `pods`. By declaring the state of the system, everything is managed automatically.

- **Ingress**
  Once the cluster is made, it is time to launch `deployments` of `pods` on the cluster. But how will a user allow external traffic to their application? As per the concept of Kubernetes, it offers isolation between pods and the outside world. To communicate with a service running within a `pod`, the outsider needs to open a channel. The channel is a medium of communication and is known as `ingress`. There are numerous ways for adding an ingress to the cluster. The most common being through an ingress controller or load balancer.

2.1.3 Persistent Storage in Kubernetes

Containers by design are stateless entities. They are meant to be executed like a program, that starts and eventually stops. But the applications that containers contain need persistent ("stateful") storage to store data, such as configuration data and databases. It is only natural for users to demand persistent storage in Kubernetes clusters, and that too without worrying about how it works under the hood [1].

Kubernetes solves this problem of persistence using **Kubernetes Volumes**. Applications that require persisting data on the filesystem may use volumes for mounting storage devices to ephemeral containers similar to how volumes are used with VMs. Kubernetes has properly designed this concept by loosely coupling physical storage devices with containers by introducing an intermediate resource called `persistent volume claims`

(PVCs). A PVC defines the disk size, disk type (ReadWriteOnce, ReadOnlyMany, ReadWriteMany) and dynamically links a storage device to a volume defined against a pod. The binding process can either be done in a static way using PVs or dynamically be using a persistent storage provider. In both approaches, a volume will get linked to a PV one to one and depending on the configuration given data will be preserved even if the pods get terminated. According to the disk type used multiple pods will be able to connect to the same disk and read/write.

Kubernetes provides a collection of volume plugins for supporting storage services available on public cloud platforms such as AWS EBS, GCE Persistent Disk, Azure File, Azure Disk and many other well-known storage systems such as NFS, CephFS, GlusterFS, Cinder, etc.

2.2 Rook

Rook is an open source cloud-native storage orchestrator, providing the platform, framework, and support for a diverse set of storage solutions to natively integrate with cloud-native environments.

Rook takes storage software such as Ceph, Minio, CockroachDB etc and turn them into self-managing, self-scaling, and self-healing storage services. It does this by automating deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management. Rook relies on Kubernetes to perform its duties such as container scheduling, management and orchestration.

Rook runs over top of Kubernetes cluster and, deploys and manages life cycle of Ceph services as Kubernetes pods. Rook also provides support for different storage backends other than Ceph such as Minio, CockarochDB etc, or one can also write its own plugin/driver for Rook.

2.2.1 Rook Architecture

Rook uses Kubernetes operator [2] construct as its main management resource. That means that it extends the Kubernetes API by adding additional application-specific control to create, configure, and manage complex stateful Kubernetes pods and an administrator just have to declare the desired state of their cluster. The state that will be managed by the operator includes everything necessary to get the cluster up and running and keep it healthy.

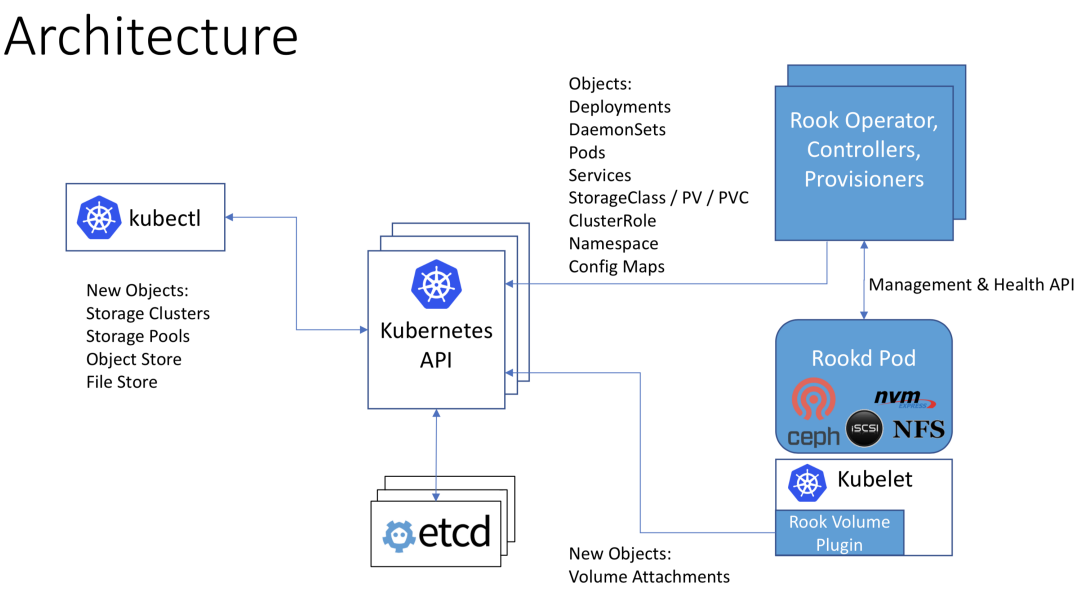

FIGURE 2.2: Rook Architecture

With Rook running in the Kubernetes cluster, Kubernetes applications can mount block devices and filesystems managed by Rook, or can use the S3/Swift API for object storage. The Rook operator automates configuration of storage components and monitors the cluster to ensure the storage remains available and healthy.

The Rook operator is a simple container that has all that is needed to bootstrap and monitor the storage cluster. The operator will start and monitor Ceph monitor pods and a daemonset for the OSDs, which provides basic RADOS storage. The operator manages CRDs for pools, object stores (S3/Swift), and file systems by initializing the pods and other artifacts necessary to run the services.

The operator will monitor the storage daemons to ensure the cluster is healthy. Ceph mons will be started or failed over when necessary, and other adjustments are made as the cluster grows or shrinks. The operator will also watch for desired state changes requested by the api service and apply the changes.

The Rook operator also creates the Rook agents. These agents are pods deployed on every Kubernetes node. Each agent configures a Flexvolume plugin that integrates with Kubernetes' volume controller framework. All storage operations required on the node are handled such as attaching network storage devices, mounting volumes, and formating the filesystem. [7]

Chapter 3

Project Details

This project involved evaluating Ceph deployments on Kubernetes clusters using Rook and this needed to be done by taking into account CERN's storage use-cases met by Ceph. On the high level, this project can be divided into two parts: Deployment and Development. This chapter covers all the deployment and implementation details starting from deploying a Kubernetes cluster to deploying a full fledged Ceph cluster with Rook.

3.1 Deployment

First thing that is needed before deploying a Rook cluster is a Kubernetes cluster. Rook supports Kubernetes version 1.7 or higher, for this project version 1.10 was used. In this project, cluster deployments were tested in both virtual as well as physical environments. Details of both of these can be found in the following.

3.1.1 Virtual Environment

The virtual environment used for Kubernetes cluster deployment is OpenStack's virtual machines because cloud infrastructure at CERN is powered by OpenStack. In order to deploy a new Kubernetes cluster on a bunch of virtual machines, an individual needs to have access to OpenStack container orchestration engine `(coe)` that is provided by OpenStack's Magnum project.

Magnum is an OpenStack API service developed by the OpenStack's containers team making container orchestration engines such as Docker Swarm, Kubernetes, and Apache Mesos available as first class resources in OpenStack. Magnum uses Heat to orchestrate

an OS image which contains Docker and Kubernetes and runs that image in either virtual machines or bare metal in a cluster configuration [5].

Following is a step by step guide on how to deploy a Kubernetes cluster using Openstack Magnum at CERN:

- The easiest and most flexible way of carrying things forward is to deploy an OpenStack virtual machine and carry out rest of the operations from there. An openstack VM can be created from dashboard that can be accessed at openstack.cern.ch. After logging into lxplus.cern.ch account, connect to that VM and install all the necessary command line interfaces required for this project.
- Create a virtual environment and install python clients required. For this project, we needed openstackclient, magnumclient, heatclient and kubectl:

```
> virtualenv /usr/test_rc/activate
> source /usr/test_rc/bin/activate
> pip install python-openstackclient
> pip install python-magnumclient
> pip install python-heatclient
> cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg \
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
> yum install -y kubectl
```

- We are ready to deploy our Kubernetes cluster now. At CERN, we have magnum templates created for cluster deployments. Supported container orchestrators are Kubernetes, Swarm, Mesos. These templates can be listed by doing a:

```
> openstack coe cluster template list
-------+--------------------+
| uuid | name               |
+------+--------------------+
| .... | swarm              |
| .... | swarm-preview      |
| .... | swarm-alpha        |
| .... | kubernetes         |
| .... | kubernetes-preview |
| .... | kubernetes-alpha   |
| .... | dcos-preview       |
+------+--------------------+
```

We choose Kubernetes template for deploying our cluster. It deploys fedora atomic hosts of medium flavor, uses CERN's gitlab registry for docker images and flannel as a networking plugin for Kubernetes containers.

```
> openstack coe cluster create kuberook --keypair mykey \
                --cluster-template kubernetes --node-count 2
```

Above command will start the creation process and after almost half an hour to one hour a Kubernetes cluster will be ready with 2 worker nodes and one master node.

- Next step is to source cluster credentials and start using it. This can be done as follows:

```
> openstack coe cluster config kuberook > env.sh
> source env.sh
```

It would be good idea to see the contents of config file to get an idea of things required to access a Kubernetes cluster. For instance, for above cluster, our master node is running at 188.184.29.166 and its API server is listening on port 6443.

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /root/ca.pem
    server: https://188.184.29.166:6443
  name: kuberook
```

```
contexts:
- context:
    cluster: kuberook
    user: admin
  name: default
current-context: default
kind: Config
preferences: {}
users:
- name: admin
  user:
    client-certificate: /root/cert.pem
    client-key: /root/key.pem
```

We can start using our Kubernetes cluster with `kubectl` now. It will automatically authenticate with the API using credentials provided above. It is also good idea to play around it with a bit such as listing namespaces, nodes, pods in different namespaces etc.

Now that Kubernetes cluster is ready, we can go ahead and deploy our Rook cluster. For that, we will have to clone Rook's official repository. At the time of this project, it was recommended by Rook's community to use Rook's version 0.8.0 because it offered more features than previous versions and introduced Ceph's beta version.

```
> git clone https://github.com/rook/rook.git
> git checkout tags/v0.8.0
```

In order to start a Rook cluster, the desired state of cluster is defined in YAML files. Rook already comes with example YAML files that can be tweaked according to a user's needs. Deployments start with creating `operator` and then rest of the cluster with Ceph services. One thing that needs to be changed before deployment is FLEXVOLUME plugin path in operator.yaml because Rook uses FlexVolume to integrate with Kubernetes for performing storage operations so overriding it with desired directory is mandatory. In some operating systems where Kubernetes is deployed, the default Flexvolume plugin directory (the directory where FlexVolume drivers are installed) is read-only. In these cases, the kubelet needs to be told to use a different FlexVolume plugin directory that is accessible and read/write (rw). It is done by updating a field in operator.yaml. If it is not configured, `rook-agent` pods will crash and storage would not work.

```
> cd rook/cluster/examples/kubernetes/ceph
```

Find the following lines in operator.yaml file and update the value with the path below:

```
- name: FLEXVOLUME_DIR_PATH
  value: "/var/lib/kubelet/volumeplugins"
```

Now, we can deploy our cluster by doing:

```
> kubectl create -f operator.yaml
```

`rook-ceph-system` namespace should appear in Kubernetes cluster:

```
> kubectl get ns
NAME               STATUS    AGE
default            Active    42d
kube-public        Active    42d
kube-system        Active    42d
rook-ceph-system   Active    6d
```

and in that namespace, following `pods` should exist:

```
> kubectl get pods -n rook-ceph-system
rook-ceph-agent-cp8pm                  1/1       Running   0          6d
rook-ceph-agent-qwnx8                  1/1       Running   0          6d
rook-ceph-agent-rrpm5                  1/1       Running   0          6d
rook-ceph-operator-6757648f75-n56sk    1/1       Running   0          6d
rook-discover-2svtc                    1/1       Running   0          6d
rook-discover-7x6n8                    1/1       Running   0          6d
rook-discover-czl59                    1/1       Running   0          6d
```

Operator pod starts `rook-agent` and `rook-discover` pods on every Kubernetes node. `rook-agent` configures storage plugin with `kubelet` and `rook-discover` pods discover different disks and their partitions on every node. After these, `rook-operator` waits for `rook-ceph` namespace to be created so that it can spawn the rest of Ceph services.

```
> kubectl create -f cluster.yaml
> kubectl get pods -n rook-ceph
```

```
rook-ceph-mgr-a-bc965f98f-6f589          1/1       Running     0          6d
rook-ceph-mon0-ftkqr                     1/1       Running     0          6d
rook-ceph-mon1-w6rs7                     1/1       Running     0          6d
rook-ceph-mon2-vrk9c                     1/1       Running     0          6d
rook-ceph-osd-id-0-655b6fd69-f9tb9       1/1       Running     0          6d
rook-ceph-osd-id-1-f7989966c-259kd       1/1       Running     0          6d
rook-ceph-osd-id-2-5f5dd7c474-spdz4      1/1       Running     0          6d
rook-ceph-osd-prepare-minion-0-qf6z8     0/1       Completed   0          6d
rook-ceph-osd-prepare-minion-1-m2v5b     0/1       Completed   0          6d
rook-ceph-osd-prepare-minion-2-xl5sp     0/1       Completed   0          6d
```

Rook deploys `monitor` pods according to the specifications in operator.yaml. `rook-ceph-osd-prepare` pods get deployed on every Kubernetes node and are responsible for detecting devices and partitioning them. Their state is meant to be `Completed` because their sole job is to prepare the nodes for OSD deployment. Currently, only one `mgr` pod can be deployed with Rook but more will be supported in future releases. Rest of the example YAML files are available and can be used to deploy filesystem and object store.

3.1.2 Physical Environment

Physical environment used for this project consisted of bare metal hosts managed by OpenStack Ironic [3] because it is the primary source of provisioning and maintaining bare metal at CERN. Each node consisted of more than 50 disks with each disk having 5.5 TB of disk space. Rook deployment process on these bare metal nodes is the same as in virtual environment, except for initial configuration. The initial configuration is not required when a new cluster is being deployed on on ironic nodes, but in this project, ironic nodes were added to a running cluster i.e. cluster running in virtual environment was extended by adding more Kubernetes nodes but now these Kubernetes nodes were bare metal machines instead of virtual machines. This was done by creating a stack in Ceph Ironic project and providing Kubernetes's master node credentials to worker nodes in this new stack. Rest of the details can be found in the README file.

3.2 Orchestrator CLI

One of the major aspect of this project was to use orchestrator command line interface and explore what it had to offer. Orchestrator CLI is an extension to Ceph MGR

daemon and it brings orchestration functionality to Ceph so that Ceph does not have to be dependent on external orchestrators. It also makes it easy for people who are already familiar with Ceph CLI to perform orchestration tasks through same interface. Currently, it offers following functionality:

- It can list devices on nodes
- It can check its own status
- It can add services to running cluster e.g. it can add OSDs on specific nodes, MDS to specific filesystem and RGW to objectstore.
- It can get status for running services (OSD, MDS, RGW)

In order to start using orchestrator CLI, some additional steps are required.

- Ceph operations can be performed easily by deploying `rook-ceph-toolbox` pod and accessing CLI from there. All the basic tools are already installed there.

  ```
  > kubectl create -f toolbox.yaml
  ```

- Connect to that pod and start using Ceph CLI.

  ```
  > kubectl -n rook-ceph exec -it rook-ceph-tools bash
  > ceph -s
  ```

- Orchestrator CLI module is not enabled by default so it needs to be explicitly enabled and its backend is also needed to be set.

  ```
  > ceph mgr module enable orchestrator_cli
  > ceph mgr module enable rook
  > ceph orchestator set backend rook
  ```

- It is enabled now and ready to use. Following is an example of how can an OSD be added on a certain drive:

  ```
  > ceph orchestrator device ls
  > ceph orchestrator service add osd rook-ironic-node:sdb
  Success.
  > ceph orchestrator service status osd 0
  ```

  It must be noted here that these operations of adding services are asynchronous that is `Success` message does not guarantee that OSD pod has been created in cluster. An important thing to mention here is that OSD would not be added to cluster if `cluster` level storage is used instead of `node` level. These settings can be found in operator.yaml.

3.3 Development

If the contents of operator.yaml are observed, it can be seen that the name of docker image used for operator deployment is `rook/ceph` by default. This is the same image that is used for deployment of rest of the Ceph pods as well, meaning that this one docker image has Rook as well Ceph binaries. On the system where Rook is built, it should be outputting an image with a name like `<some hash>/ceph-amd64` (visible in `docker image ls`) – this corresponds to `rook/ceph:v0.8.0` (the friendly tag only gets applied when pushing it to a repository). Docker image naming can be confusing, because the real identifier of an image is its hash, and the names we refer to are specific to a repository (or to person's local system).

That default `rook/ceph:v0.8.0` image contains Ceph binaries from the luminous release, which are downloaded as RPMs by the Rook build process.

```
containers:
- name: rook-ceph-operator
  image: rook/ceph:v0.8.0
```

3.3.1 How to build Rook?

In order to build Rook locally, two things need to be ensured **1)** Go lang is installed **2)** Paths for Go are correctly set. After making sure these two requirements are met, checkout to Rook's branch, for this project release-0.8 was checked out with tag 0.8.0. Then do a `make` in Rook's directory. If any kind of errors are faced refer to the troubleshooting section in the end.

After successful built of Rook, following docker images should be seen:

```
> docker images
build-6d4fb/minio-amd64                latest  a124380bf386   9 days ago
build-6d4fb/cockroachdb-amd64          latest  428caee3814e   9 days ago
build-6d4fb/ceph-amd64                 latest  aaf02ee8dada   9 days ago
build-6d4fb/ceph-toolbox-amd64         latest  c3eed29fb85e   3 weeks ago
build-6d4fb/ceph-toolbox-base-amd64    latest  dec5c1d0db82   3 weeks ago
```

The image that is of our interest is `build-6d4fb/ceph-amd64`. Development workflow from now onwards might vary from environment to environment, but for this project

the development environment was distributed. What that means is Rook and Ceph were built on two separate machines and then combined together in one image. In order to use this image `build-6d4fb670/ceph-amd64` for injecting our desired Ceph binaries instead of ones released by Rook's official repository, we have to push this image to a docker repository. This project used docker hub as image hosting site. One important think to mention here is at the time of this project, the Orchestrator CLI needed some privilege from Rook's side which was not merged so this patch here [8] was applied.

```
> docker image tag build-6d4fb/ceph-amd64 <username>/ceph-amd64:<version>
> docker push image <username>/ceph-amd64:<version>
```

It needs to be ensured that our system can push to a certain repository on docker hub and also, that repository should exist in docker hub. For instance, in above example, `<username>` should exist on docker hub and it should be public. If it is not public, some extra steps are needed for setting up the credentials. Alternatively, any image hosting site could have been used e.g. CERN has its own private registry as well hosted on gitlab or if Rook and Ceph are built on the same system then using a local docker registry would be convenient. This whole process of Rook build was carried out on the same virtual machine that was created for accessing and deploying cluster.

3.3.2 How to build Ceph?

After our Rook and Ceph image becomes available for use, we turn to Ceph side. This step was needed because we wanted to use current Ceph `master` branch, not the one released by official Rook. As mentioned earlier, one vital aspect of this project was to use orchestrator command line interface and it was available in Ceph's master branch. It should be also noted here that when this project was initiated, orchestrator CLI code was still in development phase and not in master branch so some additional steps were taken to make it stable but these steps are not required now as code has been merged into master branch of Ceph. Initial step to generate Ceph binaries of our interest is to build Ceph and that process is same as traditional Ceph build process i.e.

```
> git clone https://github.com/ceph/ceph.git
> ./install_deps
> ./do_cmake.sh
> make -j(NO_OF_PROCESSORS)
```

After getting a successful build of `master` branch, we have our Ceph binaries to replace in `build-6d4fb/ceph-amd64` docker image. This process has been simplified by

`kubejacker.sh` script found in `ceph/src/script/kubejacker`. It will take that same image and overwrite the Ceph binaries with those from a local built tree. The only thing that needs to be added are REPO and BASEIMAGE environment variables in that script according to the development environment being used. For our case, it was like that:

```
REPO=<username>
BASEIMAGE=REPO/ceph-amd64:<version>
IMAGE=ceph
```

and the last three lines where pods are being killed needed to be commented out because cluster is not accessible from this machine. TAG can be set according to your desire and this IMAGE:TAG is the same name as used in operator.yaml. We run our `kubejacker.sh` script from `ceph/build directory` and the respective docker image will be pushed to docker hub. The host used for this purpose was cephbuild.cern.ch.

3.4 Teardown

Tearing down cluster is a simple process but the order of commands needs to ensured otherwise pods can get stuck in TERMINATING state for long. Following is a step by step guide on how to do that:

- First of all delete all the file and block artifacts deployed in the cluster:

  ```
  kubectl delete -n rook-ceph pool replicapool
  kubectl delete storageclass rook-ceph-block
  kubectl delete -f kube-registry.yaml
  ```

- After that, delete rest of the cluster by removing cluster CRD:

  ```
  kubectl -n rook-ceph delete cluster.ceph.rook.io rook-ceph
  ```

  This will clean up the cluster if `rook/ceph:v0.8.0` is being used but since our project is using custom docker image, we will need to do a:

  ```
  kubectl delete -f cluster.yaml
  ```

  to delete all the resources in our cluster.

- In the end, rook operator and agent need to be cleaned up.

```
kubectl delete -f operator.yaml
```

- One important step in teardown process is to delete config files from Kubernetes nodes manually because config files and data on nodes remain persistent. For that, log into Kubernetes nodes and delete `/var/lib/rook` or whatever hostDir was configured in operator.yaml. Also, if OSDs were deployed then partitions of disks need to be deleted manually as well. If these two steps are not taken, next deployment would find these directories to be non-empty and fail. Partitions and data on disks can be deleted in multiple ways, following is just one approach:

  ```
  > sudo rm -rf /var/lib/rook
  > lsblk
  > sudo fdisk /dev/<disk-name>
  ```

  then follow the process for partition removal.

Chapter 4

Evaluation and Results

This chapter covers two things: 1) evaluation criteria established to determine if deployments with Kubernetes and Rook address our pain points or not 2) results of evaluations.

4.1 Evaluation Criteria

Our main concerns are latency of Ceph operations, automation of Ceph upgrades and autoscaling. Formalizing these requirements into evaluation metrics, we have:

- Time to deploy whole Ceph cluster
- Time to add new OSDs in a running cluster
- Autoscaling: adding/removing S3/CephFS daemons (MDS, RGW, OSD)
- Ceph upgrades: how much is it automated?

After establishing our Ceph cluster with Rook and Kubernetes, S3 workloads were deployed to test the deployments.

4.2 Results

Currently, Ceph is being deployed at CERN using puppet which takes time and its configuration requires expertise. Following table shows comparison between current time taken by puppet and then time taken by Ceph deployment using Rook.

Evaluation Metric	**with Puppet**	**with Rook and k8s**
Time to deploy whole cluster	$>$ 3 hours	$<$ 15 minutes
Time to add new OSDs in a cluster	$>$ 1 hour	$<$ 2 minutes
Autoscaling S3/CephFS daemons	$>$ 1 hour	$<$ 2 minutes
Ceph upgrades	Manual	Manual (WIP)

It is evident from above table that Rook with Kubernetes improves latency of operations significantly. Previous Ceph operations that were taking hours can now be achieved in minutes. An important point to note here is that although upgrades are still manual in Rook, since Ceph services are running in containers, the process is less involved as compared to the previous version.

In addition to being time efficient, it is failure tolerant i.e. when a pod crashes `rook-operator` respawns it with the same configuration. It is also elastic in a sense that a user just have add more resources in the cluster and `rook-operator` will acknowledge those resources and start using them.

Chapter 5

Conclusion and Future Work

In summary, this project aimed to explore a new approach to improve Ceph operations at CERN. Owing to the huge infrastructure of CERN, making operations faster and easier have always been a priority. Rook looks like a promising open source project that targets to make storage better for cloud. Although CERN does not have production servers running Kubernetes yet, but container based approaches are being actively explored here and soon we would have Kubernetes clusters in production as well. The results of this project verify the future for container based solutions. One important thing to mention here is that although Ceph operations are improved in terms of latency and ease of use, Ceph managers will need to learn how to operate a Kubernetes cluster and since Rook is comparatively a newer technology, all the storage use cases are exhaustively not covered as yet.

This project also contributed open source code to Ceph and added RGW support in orchestrator CLI but there are still many features that need to be added, for instance, following two are next priority:

- Making RGW support a two way thing in which it waits for completion
- Currently, a user cannot remove services from orchestrator CLI so adding the functionality to remove services will come in handy.

All in all, it will be a good idea to spawn the next Ceph cluster with Rook and Kubernetes, ideally when features like automated upgrades and support of decoupled versions are achieved.

Chapter 6

Appendix

6.1 Troubleshooting

- Namespace stuck in `Terminating` state:
  If a namespace is still stuck in terminating stage after cluster was brought down then make sure all the resources in that namespace have been deleted. If a pod is still running, do a:

  ```
  > kubectl -n NAMESPACE get pods
  > kubectl delete pod POD -n NAMESPACE --force --grace-period=0
  ```

  If the issue still persists then look for cluster CRD, if it is there go ahead and delete it:

  ```
  > kubectl -n rook-ceph get cluster.ceph.rook.io
  > kubectl -n rook-ceph edit cluster.ceph.rook.io rook-ceph
  ```

  Delete the line `- cluster.ceph.rook.io` from the file. After few seconds, namespace should be deleted [6].

- Docker containers are unable to access internet
  When `kubejacker.sh` script is being used to build docker images on cephbuild.cern.ch host, containers were unable to access internet. Add nameservers to `/etc/resolv.conf` and restart docker daemon.

  ```
  > cat /etc/resolv.conf
  # Generated by NetworkManager
  search cern.ch
  ```

```
nameserver 137.138.17.5
nameserver 137.138.16.5
nameserver 2001:1458:201:1100::5
> service docker restart
```

- OSD pods are not getting launched in virtual environment:
  Check for disk space of volumes that you are attaching to virtual machines. It should be greater that 10 GB.

- Inter node pod connectivity issue:
  If inter node connectivity is lost it would most probably because of misconfiguration of kube-proxy. Do this on each Kubernetes node:

  ```
  sudo su
  cat > /etc/kubernetes/proxy <<EOF KUBE_PROXY_ARGS="
  --kubeconfig=/etc/kubernetes/proxy-config.yaml
  --cluster-cidr=10.100.0.0/16" EOF
  systemctl restart kube-proxy.service
  ```

  The cidr value comes from here: openstack stack show rook-ironic-nodes –no-resolve-outputs — grep pods_network_cidr. If there is a pod that still is not accepting traffic, execute this command to allow traffic to that node:

  ```
  > sudo iptables FORWARD ACCEPT
  ```

Bibliography

[1] Gokul. C. *Rook.io : Ceph Persistent Storage Made Easy on Kubernetes*. URL: https://www.linkedin.com/pulse/rookio-ceph-persistent-storage-made-easy-kubernetes-gokul-chandra/. (accessed: 29.08.2018).

[2] The CoreOS. *Operators*. URL: https://coreos.com/operators/. (accessed: 28.08.2018).

[3] OpenStack Ironic. *Magnum*. URL: https://wiki.openstack.org/wiki/Ironic. (accessed: 29.08.2018).

[4] Kubernetes. *Kubernetes Concepts*. URL: https://kubernetes.io/docs/. (accessed: 28.08.2018).

[5] OpenStack Magnum. *Magnum*. URL: https://wiki.openstack.org/wiki/Magnum. (accessed: 29.08.2018).

[6] Rook. *Common Issues*. URL: https://rook.io/docs/rook/v0.8/ceph-teardown.html#removing-the-cluster-crd-finalizer. (accessed: 29.08.2018).

[7] Rook. *Rook Docs*. URL: https://rook.io/docs/rook/v0.8/. (accessed: 28.08.2018).

[8] John Spray. *Set service account on ceph-mgr pod*. URL: https://github.com/rook/rook/pull/2001. (accessed: 29.08.2018).