Integrating Containers in the CERN Private Cloud

To cite this article: Bertrand Noel et al 2017 J. Phys.: Conf. Ser. 898 092045

View the article online for updates and enhancements.

Related content

- New directions in the CernVM file system
  Jakob Blomer, Predrag Buncic, Gerardo Ganis et al.

- Optimizing CMS build infrastructure via Apache Mesos
  David Abdurachmanov, Alessandro Degano, Peter Elmer et al.

- Docker experience at INFN-Pisa Grid Data Center
  E. Mazzoni, S. Arezzini, T. Boccali et al.
Integrating Containers in the CERN Private Cloud

Bertrand Noel, Davide Michelino, Mathieu Velten, Ricardo Rocha and Spyridon Trigazis
European Organization for Nuclear Research (CERN), CH-1211 Geneva, Switzerland

Abstract. Containers remain a hot topic in computing, with new use cases and tools appearing every day. Basic functionality such as spawning containers seems to have settled, but topics like volume support or networking are still evolving. Solutions like Docker Swarm, Kubernetes or Mesos provide similar functionality but target different use cases, exposing distinct interfaces and APIs.

The CERN private cloud is made of thousands of nodes and users, with many different use cases. A single solution for container deployment would not cover every one of them, and supporting multiple solutions involves repeating the same process multiple times for integration with authentication services, storage services or networking.

In this paper we describe OpenStack Magnum as the solution to offer container management in the CERN cloud. We will cover its main functionality and some advanced use cases using Docker Swarm and Kubernetes, highlighting some relevant differences between the two. We will describe the most common use cases in HEP and how we integrated popular services like CVMFS or AFS in the most transparent way possible, along with some limitations found. Finally we will look into ongoing work on advanced scheduling for both Swarm and Kubernetes, support for running batch like workloads and integration of container networking technologies with the CERN infrastructure.

1. Introduction

The CERN OpenStack cloud[3] entered production in 2013 and has since had a steady growth. As of the end of 2016 it hosts more than 2700 users across 3300 projects, with a total of 23000 VMs running on 7000 hypervisors. It includes a large range of OpenStack services, including the most commonly deployed: Keystone for authentication, authorization and service discovery; Glance for image storage and provision; Cinder for persistent block storage, backed by the CEPH storage system; Nova as the core of the system for virtual machine handling; and Horizon as the main user frontend dashboard. In addition we’ve more recently added: Heat as the orchestration service, for multi-node and more complex application stacks; Neutron as the new networking component, replacing the deprecated Nova Network.

The workloads being run by users vary from batch jobs doing simulation, event reconstruction and end user physics analysis and running infrastructure services on which the campus depends. For the majority of the long running services the deployment and configuration is typically managed by Puppet, but recently users started looking at containers as an alternative for improved flexibility and scalability.

Following this demand we started looking at how we could offer our users a container service that would be well integrated with the existing OpenStack infrastructure. Magnum appeared as a viable solution as it was compatible with our main goals:
Integrate well with OpenStack at CERN, reusing our identity service, resource allocation, networking and data access
Support for multiple container orchestration services, especially Docker Swarm, Kubernetes and Mesos
Fast and easy to use, allowing users to scale clusters easily

In the following sections we give an overview of the reasons why containers are an exciting addition to our range of services, the technologies we chose and the process that lead us to a production service.

2. Containers
Containers are a form of operating system level virtualization to run multiple isolated systems in the same host and kernel, avoiding a heavy virtualization layer to achieve increased performance and resource utilization. They exist in other platforms such as Solaris (called Solaris Zones) and in different flavors for Linux, including LXC (Linux containers), OpenVZ and more recently Docker.

To accomplish the required isolation between workloads and processes all of the solutions presented above rely on the following set of kernel features:

- cgroups, allowing limitation and prioritization of resources (CPU, memory, block I/O, network) without the need to start virtual machines
- namespace isolation, giving individual applications independent views of the operating environment, including distinct process trees, networking, users and mounted filesystems

Running workloads on a shared kernel presents security challenges, and only after multiple iterations these issues have been mitigated. Users can run today multi-tenant workloads on a single machine with confidence, and add or remove sets of privileges as required.

2.1. Docker
Docker generalized container adoption by offering a greatly simplified interface to manage the concepts described above, as well easy encapsulation of applications. Developers are able to fully describe their application setup in a simple manifest (a Dockerfile), which includes a series of commands and definitions. The result is a unit of deployment called a container image that can be easily reused, deployed in heterogeneous environments and scaled during the lifetime of an application. Images can also be layered for space and resource usage optimization.

Listing 1. Dockerfile

```sh
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y redis-server
EXPOSE 6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

The resulting images are stored in registries (often shared publicly) and referenced by the application manifests used in container orchestration engines. The most popular registry is the Dockerhub, where official images for most popular services can be found. The simplicity of this process has made it popular initially in development environments and continuous integration systems, eventually making it all the way to production setups.

3. Container Orchestration Engines
Among all types of tools already relying on the basic container building blocks, orchestration engines are the most relevant. These allow users to define their whole application stack in a single manifest file, including not only the actual application code to be run but also the networking,
associated storage and relationships between the different pieces. This type of deployment is often called microservices due to its layout as a set of small coordinated services. Below we describe three of the most popular orchestration engines.

3.1. Kubernetes

Kubernetes is an open source project for managing containerized applications on top of an heterogeneous cluster of hosts. It provides basic but robust mechanisms to deploy, maintain, scale and update applications.

The base code has been contributed by Google and the project is now under the umbrella of the Cloud Native Computing Foundation, which is part of the Linux Foundation.

Some of its main abstractions include:

- Pod, a set of one or more co-scheduled containers, sharing network and local storage
- ReplicaSet, ensures a specified number of replicas of a pod is running at any given time
- Service, entrypoint and load balancer for client requests to a set of Pods
- Deployment, a higher-level construct that manages a set of any other number of abstractions

There are many other concepts available, such as Secrets for secure declaration of data blobs, StatefulSets for pods with special requirements regarding network identity or persistent storage, or Volumes for external block storage availability inside Pods.

Listings 2 and 3 present manifests for a simple guestbook application:

**Listing 2. Service**

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
labels:
  app: guestbook
tier: frontend
spec:
  ports:
  - port: 80
  selector:
    app: guestbook
tier: frontend
```

**Listing 3. Deployment**

```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
          - containerPort: 80
```

Internally Kubernetes is composed of several components:

- the Kubelet, runs on every node and interacts with the local container runtime (usually Docker) to comply with scheduler instructions
- the API server, answers clients, maintains the global state of the cluster
- the Scheduler, responsible for matching deployment requests to the current node status and availability
- the Controller-Manager, responsible for matching the desired state with the current state of the cluster
3.2. Docker Swarm

Docker Swarm provides native clustering capabilities to turn a group of Docker engines into a single, virtual Docker instance. Initially a separate component, it is now part of the main Docker tool with different instances easily communicating together to form a single, huge computer.

Swarm provides similar abstractions to the ones described in Section 3.1 for Kubernetes, including Services, Volumes and Secrets. Services act as endpoints for clients to contact applications running in individual containers, making sure the load is properly balanced along the multiple replicas. The notion of a Pod does not exist, with each container having its own local storage and network stack. Integration with persistent storage is done via a clean and simple plugin API allowing vendors to provide their own volume plugins.

For orchestration Swarm relies on Docker Compose, which has recently also been integrated into the core Docker client. Deployments are called stacks and are defined using a yaml language similar to the one for Kubernetes manifests. Listings 4 and 5 show an example deployment.

**Listing 4. Docker Compose**

```yaml
version: "3"
services:
  redis:
    image: redis:alpine
    ports:
    - "6379"
    networks:
    - frontend
    deploy:
      replicas: 2
```

**Listing 5. Docker Compose (continue)**

```yaml
db:
  image: postgres:9.4
  volumes:
  - db-data:/var/lib/postgresql/data
  networks:
  - backend
  deploy:
    placement:
      constraints: [node.role == manager]
```

3.3. Mesos/DCOS

Mesos[2] is an Apache top level project, describing itself as the *datacenter kernel*. It abstracts datacenter resources (CPU, RAM, storage) and proposes them to registered frameworks, in the form of offers which they can accept or decline. Each framework can run one or more applications isolated via containers or pure cgroups.

Popular frameworks include Marathon for service deployment and Chronos to run scheduled tasks, with support for a very wide range of applications such as Spark, Hadoop or Elastic Search. One of its main features is the two level scheduling, allowing easy integration of containerized and cloud native with legacy applications via custom scheduling policies.

DCOS is a distribution built on top of Mesos initially released as a commercial product but recently made open source. It adds more advanced features such as L7 load balancing and advanced management tools, including a framework marketplace and web and command line interfaces.

4. OpenStack Magnum

Magnum is an API service developed by the OpenStack Containers team handling container cluster management. The three essential concepts in Magnum are:

- Cluster Template, a reusable layout for cluster deployments
- Cluster, a specific deployment of a container cluster
- Node, which can be either a virtual machine or a baremetal server

Cluster templates contain the full description of how a cluster should look like, most importantly the container orchestration engine (COE), the type of node (VM or baremetal),
the flavors to be used for both the master and the nodes, the server image (typically Fedora
Atomic or CoreOS), and any advanced features like load balancing.

Currently there is support for Kubernetes, Docker Swarm, Apache Mesos and DC/OS, leaving
users with a good set of options to match their use cases.

4.1. Architecture

Figure 4.1 introduces the different interactions between Magnum, clients and other OpenStack
components. Following the usual OpenStack service layout it consists of two daemons,
the magnum-api and magnum-conductor. The first handles user requests for cluster
templates, clusters and certificates, while the second handles all other asynchronous operations.
Communication between the two is done via RPC queues over a RabbitMQ server.

To handle the heavy work of deploying and managing the multiple cluster components
Magnum relies on OpenStack Heat, and each cluster is represented by a set of Heat stacks.
Once a cluster is available the client can request a set of TLS certificates to communicate
directly with the orchestration engine via its native API. Each cluster has a distinct certificate
authority for credential isolation.

4.2. Usage

In addition to guaranteeing proper integration with the remaining OpenStack services, one of
the main goals of Magnum is to simplify the cluster deployment process and minimize the
complexity of user interactions. Listing 6 summarizes all the interaction required for a user to
get a running cluster in only a couple minutes. Listing 7 shows how users can easily scale their
existing clusters.

Figure 1. Magnum Architecture
Listing 6. Magnum Usage

```bash
$ magnum cluster-create --name mycluster --cluster-template swarm --node-count 100
$ magnum cluster-list
```

<table>
<thead>
<tr>
<th>uuid</th>
<th>name</th>
<th>node_count</th>
<th>master_count</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>mycluster</td>
<td>100</td>
<td>1</td>
<td>CREATE_COMPLETE</td>
</tr>
</tbody>
</table>

```bash
$ (magnum cluster-config mycluster --dir magnum/mycluster)
$ docker info / ...
$ docker run --volume-driver cvmfs -v atlas.cern.ch:/cvmfs/atlas -it centos /bin/bash
```

Listing 7. Magnum Scaling

```bash
$ magnum cluster-update mycluster replace node_count=200
$ magnum cluster-update mycluster replace node_count=5
```

4.3. Deployment at CERN

Reaching a production container service at CERN involved multiple phases, spread over a year. During the first phase we performed several investigations of the existing container technologies, and tested the possibility of integration of essential CERN services with containers - mainly AFS storage at this point. The first results showed it was possible, although requiring additional privileges that would prevent running containers in a multi-tenant environment - which we worked around later.

The second step, once Magnum appeared as the most viable solution for container integration, involved an initial deployment of this service in the CERN environment. A few additional tasks were required:

- Deploy Neutron, which we did not have in production at the time. This was done in an isolated Nova cell which allowed us to immediately integrate the service into our production environment without disturbing the rest of the cloud
- Disable floating IPs, security groups, load balancers and network creation. At CERN we expose to users a single flat network (no tenant networks) using the LinuxBridge plugin (no overlay networks). These were mandatory at the time for Magnum so we run a series of patches locally, which eventually made it upstream later on

These changes gave us a working service which we exposed to a set of early pilot users. The initial use cases included continuous integration and builds, infrastructure services (including a prototype for the WLCG File Transfer Service in containers [11]), physics analysis using Jupyter/IPython notebooks (Swan[10], MyBinder[9]) and a few others. The main goal was to identify the missing pieces and work on them before opening the service to all users. We describe the most relevant in Section 4.4.
During much of 2016 we worked with our pilot users to debug and adapt the service to their needs, and worked closely with the upstream Magnum team contributing our changes and helping steer the project. The result was a stable service which we opened to all CERN users in October 2016, just a year after the initial investigations.

4.4. Storage Integration

Storage integration was the top request from all our initial users, which at CERN means CVMFS[4] and EOS[7].

CVMFS is a read-only distributed filesystem used mainly to scale the distribution of software among the several hundred sites collaborating with CERN experiments. It relies on a large network of cache layers making sure each of the thousands of nodes gets its copy in a timely manner and with minimal load on the network infrastructure. The integration of CVMFS with Magnum at CERN is done via a docker volume plugin [5], which makes sure each repository is made available on the host running the container when requested. If multiple containers in the same host access the same repository they share the mount point and cache layers, optimizing data access. As it’s a read-only filesystem, no special user or service credentials are needed and the implementation is much simplified.

EOS handles all physics data coming out of the CERN experiments, already storing hundreds of petabytes and preparing to store even more as luminosity increases and the laboratory prepares itself for a big upgrade in a few years. Data is protected using common POSIX semantics, and access requires an authentication/authorization token, usually Kerberos or X509 certificates. The implementation is again done via a docker volume plugin [6], but this time significantly more complex due to the need of sharing user certificates in a secure manner for containers running in shared hosts.

In both cases the integration with Kubernetes relies on a wrapper via the FlexVolume plugin [8], which underneath calls the Docker native API so that the same code is reused.

5. Performance Evaluation

Before opening the service to all users, we wanted to answer a series of questions. How fast is cluster deployment? How much concurrent load can Magnum handle? How large can clusters be? How do services deployed in a Magnum cluster scale?

To answer the questions above, we decided to perform a series of benchmark tests against our deployment. For the service benchmarks we used Rally, a tool designed to benchmark OpenStack services which we already used extensively for other services. To benchmark the resources created by Magnum, and evaluate how a service running in a Magnum cluster scales, we used a demo application [1] published by the Google Cloud Compute team. This demo deploys in a large cluster a web application (running in a replicated http nginx server) serving a static file, and a number of load bots to stress test the service.

The results of the first evaluation are presented in Table 1. Numbers for clusters up to 128 nodes are very good, with an almost flat deployment time. Above that there seems to be a close to linear increase following the cluster size, even if 23 minutes is a perfectly acceptable value for deploying a 1000 node cluster. We’re working together with the OpenStack Heat team to understand why this linear increase is happening.

Figure 3 shows the result of the Kubernetes scale test, where we reached 7 million requests per second using a 1000 node cluster. This was under the 10 million requests achieved by the Google team, with the issue being understood and pointing to the CERN internal network setup used for this test. We plan to redo the test to confirm this.
Cluster Size | Time (min)
---|---
2  | 2.5
32 | 4
128 | 5.5
512 | 14
1000 | 23

Table 1. Deployment Time

6. Conclusion and Future Work
In just over a year and starting with a set of early investigations on existing container technologies, the OpenStack Magnum service has been deployed in production at CERN and made available to thousands of users. It allows users to easily deploy Docker Swarm, Kubernetes and Mesos clusters of hundreds of nodes in minutes and integrates seamlessly with the internal identity, resource provisioning and storage services. Our performance evaluation proved it scales to all required use cases, including physics data analysis, high load web services and batch systems.

Additional work will involve improved support for cluster upgrades, which are currently very intrusive on running clusters; node groups, allowing a cluster to be composed of nodes with different flavors, in different AZs or with network access permissions; and cluster healing, with improved recover procedures for broken nodes and services.

We will continue doing this work together with the upstream Magnum development team.

References