Kubernetes Essentials

Student guide

Release 1 rev46

Component Soft Ltd.

January 12, 2018

Contents

Preface

Formatting notes

Here we present some examples of a the formatting applied in this document.

Note:
Here we describe the formatting rules of this book. Take the following commands as examples only. These commands not necessary execute correctly or produce the same output for your actual setup.

Example 1

```
root@controller1 (admin) $> nova list --mininal
+--------------------------------------+--------+
| ID                                   | Name   |
+--------------------------------------+--------+
| 58012abf-1b73-40ec-989c-96f4592cd277 | test_1 |
+--------------------------------------+--------+
```

In this case

- The command runs on the `controller` node, as user `root`.
- The keystone credentials for tenant `admin` are loaded into the shell environment.

In the above case it is required to load the admin credentials in order to make the certain command to work properly. In case of the `admin` user (and tenant), it can be done by sourcing the file `/root/keystonerc_admin`

```
root@controller1 $> source /root/keystonerc_admin
root@controller1 (admin) $> env |grep OS_
OS_REGION_NAME=RegionOne
OS_PASSWORD=makeitso
OS_AUTH_URL=http://10.10.10.51:5000/v2.0/
OS_USERNAME=admin
OS_TENANT_NAME=admin
```

Commands not related to OpenStack (like `ls`) does not take care of the OpenStack credentials at all, so for those, it is irrelevant whether you can `OS_` environment variables or not.

Example 2

```
root@controller1 (admin) $> nova interface-list vm1

// Port ID                              | Net ID                                >>
//-------------------------------------+---------------------------------------->>
// 7912e922-e871-4e7a-a943-34017ee29160 | 41f61be7-40b9-4a67-aeca-feae3a5986ac>>
// 93925c7f-0112-493c-8a39-261449128f5f | e3ee2d62-e216-4e76-b438-733e706f1500>>

 IP addresses //
--------------//
 10.40.40.100 //
 10.30.30.102 //
```

In this one, the output is too long, so the first and last columns are cut off (`//` symbols) and the third column is presented separately (`>>` symbol)

Example 3

```
root@controller1 (admin)$> nova host-describe compute2.openstack.local
+--------------------------+------------------+-----+-----------+------+
| HOST                     | PROJECT          | cpu | memory_mb | disk |
+--------------------------+------------------+-----+-----------+------+
| compute2.openstack.local | (total)          | 8   | 15948     | 4    |
| compute2.openstack.local | (used_now)       | 4   | 2560      | 4    |
| compute2.openstack.local | (used_max)       | 4   | 2048      | 4    |
| compute2.openstack.local | 0cb8d/--/c274e67 | 4   | 2048      | 4    |
+--------------------------+------------------+-----+-----------+------+
```

In this case the second column `PROJECT` is truncated to 20 characters, so UUID `0cb8d6ab778546bbadc69488dc274e67` is shortened to `0cb8d/--/c274e67`.

Module 1: Introduction

Introduction

- Cloud computing in general
- Cloud native computing
- Application containers
- Kubernetes overview
- Kubernetes architecture

Cloud computing in general

- a model for enabling ubiquitous network access to a shared pool of configurable computing resources*
  - resources (compute, storage) as services
    - resources are allocated on demand
      - scaling and removal also happens rapidly (seconds-minutes)
    - multi-tenancy
      - share resources among thousands of users
      - resource quotas
  - cost effective IT
    - Pay-As-You-Go model
      - pay per hour/gigabyte instead of flat rate
    - maximized effectiveness of the shared resources
      - maybe over-provisioning
    - lower barriers to entry (nice for startups)
      - focus on your business instead of your infrastructure

*definition by NIST

Cloud computing, in general, provides resources, such as compute instances, storage objects and virtual networks to its customers. These resources can be allocated/resized/dropped any time, and their usage is payed on a per minute/per hour basis (for a public cloud). This unparalleled flexibility allows small companies, like start-ups, to focus rather on their new business, instead of building up a local infrastructure.

Cloud computing also means virtualization of resources, which makes an effective use, or full utilization of hardware resources. In case of resource demanding applications (high CPU utilization, a lot of IO operations), this approach could easy lead to over-utilization.

Cloud types

- By service model
  - Infrastructure as a Service (IaaS)
    - Virtual or physical machines (MaaS)
      block storage, virtual networking (FW,LB,VPN), object store
    - Examples: AWS, OpenStack, Azure, VmWare VCenter
  - Platform as a Service (PaaS)
    - provides a middleware (OS, DB,etc maintained by the provider)
    - Examples: OpenShift, Heroku, Google App Engine
  - Software as a Service (SaaS)
    - shared access to a software (like ERP, DB or even desktop)
    - Examples: Gmail, Instagram, Adobe
- By location
  - Public cloud
    - Multi-region, shared deployment of services
  - Private cloud
    - On premise deployment, mainly for security
  - Hybrid cloud

Cloud computing offers different service models depending on the capabilities a consumer may require.

- IaaS (Infrastructure-as-a-Service)

  It provides infrastructure such as computer instances, network connections, and storage so that people can run any software or operating system. IaaS systems usually provide a set of prepared operating system images, from which end-users can start new virtual machine instances with a single click. The networking between these VMs, and additional file/object storage space is also provided by the IaaS infrastructure.

  Several IaaS providers can also provision bare-metal servers (Metal as a Service or MaaS), allowing their customers direct access to a physical hardware.

- PaaS (Platform-as-a-Service)

  With PaaS, the consumer has the ability to deploy applications through a programming language or tools supported by the cloud platform provider. An example of Platform-as-a-Service is OpenShift by RedHat. Built on top of an IaaS (Amazon Elastic Compute), it provides JBoss/FireFly as service where Java developers can deploy their applications

without taking care of the underlying operating system, database or Java runtime.

- SaaS (Software-as-a-Service)

  The consumer uses a software in a cloud environment, without needing to install anything to the local computer. The simplest example of SaaS is a web-based mail service, but recently more resource demanding applications (like Adobe Photoshop) are available as a cloud service.

By the location of the cloud servers, we can also distinguish between

- Public clouds

  Cloud resources are available from everywhere to everyone.

- Private clouds

  In this case the cloud infrastructure is installed on the site of the company, and resources can be allocated by employees only. Such design allows full control over sensitive set of data. In some cases companies required by law to store their data on a server located in the same country.

- Hybrid clouds

  This solution aims to combine the security of private clouds with the flexibility of public clouds.

Cloud native computing

- a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes.
- Main properties:
  - Container packaged – containers represents an isolated unit of application deployment.
  - Dynamically managed - actively scheduled and actively managed by a central orchestrating process.
  - Micro-services oriented - loosely coupled with dependencies explicitly described (e.g. through service endpoints).

According to the CNCF Cloud Native Computing means a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes. This will enable a constantly evolving software architectures that will allow for continuous innovation and easy deployment of new software components.

Cloud native systems will have some common properties:

- Container packaged - processes will run in containers. Containers will provide the unit of application deployment, and a mechanism for environment and resource isolation.
- Dynamically managed - components are scheduled and actively managed by a central orchestrating process. This will allow for better resource utilization and reduced cost for operation and maintenance.
- Micro-services oriented - systems are composed of several loosely coupled components that are communicating over service endpoints. This will increase the overall agility and maintainability of applications.

Application containers

- OS level virtualization – OS partitioning (virtual OS vs virtual HW)
- Allows us to run multiple isolated user-space application instances in parallel.
- Instances will have:
  - Application code
  - Required libraries
  - Runtime
- Self sufficient – no external dependencies
- Portable
- Lightweight
- Immutable images

Application containers are using operating system partitioning to create isolated, self sufficient environments for application components.

Applications running in containers are separated from each other using the services of the underlying kernel.

All containers from a host are running on the same kernel - the one that runs on the host. This way the overhead of running containerized applications is much smaller compared with virtual host based deployment.

Operating system partitioning will provide applications a virtual OS environment. The applications running in the containers will have their own view of file system, processes, users, networking, and the developers can populate/configure these resources according to their needs.

An application instance running in a container will have the application code, the required libraries, and the runtime components. This will provide a self sufficient environment for running the applications. By eliminating the external dependencies for a container packaged application, it is possible to easily deploy the application on a new host.

Container images can be created from multiple layers, and one layer can be used by several images. New images are built and tested when there are changes in the application, and by deploying the new image we are *replacing* the container. There are no changes performed on a deployed container, thus application are becoming immutable. This immutability reduces the potential of configuration drifts. In case disruptions due to unexpected events applications are redeployed instead of bring restored from different configurations and versions.

Containers on linux

- Linux native functionality.
- Not new (chroot concept dates back to Version 7 Unix in 1979)
- Linux namespaces
  - Mount
  - Process ID – independent set of PIDs
  - Network – independent network stack. Each interface is present in exactly 1 namespace (can be moved)
  - User – provide both privilege isolation and user identification segregation across multiple sets of processes.
  - IPC – isolate processes from SysV style inter-process communication.
  - UTS – isolate host and domain names.
- Control groups – limits, accounts for, and isolates the resource usage of a collection of processes.
- Linux capabilities – privileges traditionally associated with superuser

Containers are not new in the operating systems' world. These can be tracked back to the chroot system call introduced during the development of Version 7 Unix in 1979. The different functionalities that make possible the current application containers are present in the current Linux kernels. These are *Linux namespaces*, *Control groups*, and *Capabilities*.

The *Linux namespaces* are a feature of the Linux kernel that isolate and virtualize system resources for a collection of processes. Current Linux kernels defines 6 namespaces:

- the **mount** namespace allows us to create dedicated file system hierarchies for each container where mount/umount operations can be performed independently of the other containers.

- the **network** namespace virtualizes the network stack. Each namespace will have its own interfaces (each network interface is present in exactly 1 namespace and can be moved between namespaces), IP addresses, routing table, sockets, firewall, contrack table. Different network namespaces are behaving as different hosts from networking point of view.

- the **pid** namespace provides an independent set of process IDs. The first process that runs in a new pid namespace will have the PID=1, and orphaned processes within the namespace are attached to it. PID namespaces are nested, meaning when a new process is created it will have a PID for

each namespace from its current namespace up to the initial PID namespace.

- the **user** namespace provide both privilege isolation and user identification segregation across multiple sets of processes. User namespaces are also nested. A user namespace contains a mapping table converting user IDs from the container's point of view to the system's point of view.

- the **IPC** namespace will provide isolation for SysV style IPC mechanisms (message queue, semaphore sets and shared memory segments). Processes in different IPC namespaces can use SysV IPC objects with the same ID without interfering with each other.

- the **UTS** namespaces allow a single system to appear to have different host and domain names to different processes.

The *control groups* functionality of the Linux kernel allows us to limit, account for and isolate the resources used by a collection of processes. A control group (abbreviated as cgroup) is a collection of processes that are bound by the same criteria and associated with a set of parameters or limits. The kernel maintains different cgroup hierarchies for the different subsystems like: blkio, cpuacct, cpuset, devices, memory, pids, net_prio, ns, freezer, etc. A process cannot be member of different cgroups in the same hierarchy.

Linux capabilities divides the privileges traditionally associated with superuser into distinct units that can be independently enabled or disabled for a process (thread). This allow us to create fine grained privilege sets for different processes, which is different from the traditional *privileged/unprivileged* model.

Container runtime

- enables users to make effective use of the OS level virtualization mechanisms by providing APIs and tooling that abstract the low level technical details.
- will run the container images
- Open Container Initiative
  - Runtime specification – how to run
  - Image Specification
- docker
- rkt
- runC

Container runtime is at the lowest layers of software on the Kubernetes node that enables users to make effective use of the OS level virtualization mechanisms by providing APIs and tooling that abstract the low level technical details. The container runtime will provide means to download the images, start/stop them, interact with them.

Kubernetes started out with *Docker* as the container runtime, but it is also possible to integrate *rkt* as a runtime. In version 1.5 the Container Runtime Interface (CRI) has been introduced to improve the extensibility of Kubernetes by allowing easier integration of other container runtime impementations.

The Container Runtime Interface consists of a protocol buffers and gRPC API, and libraries. With CRI the kubelet communicates with the container runtime (or a CRI shim for the runtime) over Unix sockets using the gRPC framework, where kubelet acts as a client and the CRI shim as the server.

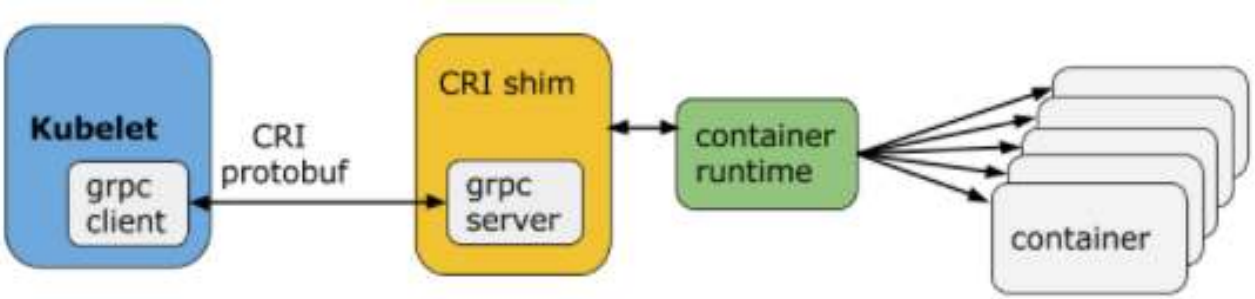

Figure 1: Kubelet interaction with CRI runtime (Source: blog.kubernetes.io)

In 2015 the Open Container Initiative (OCI) has been established with the purpose of creating open industry standards on container format and runtime. The OCI currently contains two specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime filesystem Bundle would be run by an OCI Runtime.

To help with the integration of OCI compatible container runtimes the CRI-O project has been established. This is an implementation of the Kubernetes CRI for OCI compatible runtimes. The kubelet will send requests to the CRI-O daemon using the CRI. Currently the CRI-O is tested with runC and Clear Containers, but any OCI compatible runtime is supported.

Container orchestration

- tools that are providing an enterprise-level framework for integrating and managing containers at scale.
- aim to simplify container management
  - a framework for defining initial container deployment
  - availability
  - scaling
  - networking
- Docker Swarm
- Mesosphere Marathon
- Kubernetes

Running containers on a single host allows us to abstract our application environments from what is underneath. But running containers on multiple hosts (tens, hundreds, thousands) is where the real benefits are appearing. Container orchestration systems allow us to deploy and manage containers at scale.

In general, container orchestration frameworks will handle the following tasks:

- Provisioning hosts
- Instantiating a set of containers
- Rescheduling failed containers
- Linking containers together through agreed interfaces
- Exposing services to machines outside of the cluster
- Scaling out or down the cluster by adding or removing containers

Currently there are different container orchestration tools like Docker swarm, Apache Mesos, Kubernetes, etc.

Kubernetes

kubernetes

- Kubernetes – ancient Greek word for helmsman or pilot of the ship
- Initially developed by google
- Has its origins in Borg cluster manager
- “Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.”
- Places containers on nodes
- Recovers from failure
- Basic monitoring, logging, health checking
- Enables containers to find each other

Kubernetes (commonly referred to as “K8s”) is an open-source system for automating deployment, scaling and management of containerized applications that was originally designed by Google and released in June, 2014.

With the release of version 1.0, Kubernetes was donated to the Cloud Native Computing Foundation (CNCF) as a seed technology.

The design of kubernetes is heavily influenced by Borg, Google’s own cluster manager that runs on google’s own infrastructure.

The main features of kubernetes includes:

- Container scheduling - automatically places containers based on their resource requirements and other constraints.
- Scaling - easily scale up and down applications using simple commands, user interface, or automatically based on CPU usage.
- Self healing - restarts containers that fail, replaces and reschedules containers when nodes die,

