--- title: "Tools & Concepts for Cloud Deployments" author: ["Christopher B. Hauser"] institute: ["Institute of Information Resource Management, Ulm University"] subject: "Markdown" tags: [Markdown, Example] titlepage: true graphics: true mainfont: Open Sans mainfontoptions: BoldFont=Open Sans Bold mainfontoptions: ItalicFont=Open Sans Italic mainfontoptions: BoldItalicFont=Open Sans Bold Italic

date: SummerSchool 2019, Curitiba subtitle: "Solution for Exercise 5" --- # Answers to questions

Lesson 1: Concept of Containers

Question: Containers, LXC and Docker

What architectural implications are required for an application to run in containers?

* Service-oriented: Containerized applications work best when implementing a service-oriented design. Service-oriented applications break the functionality of a system into discrete components that communicate with each other over well-defined interfaces. One container ideally represents a single service only.

* Application state: The containers should be stateless and store the application state somewhere outside the containers (e.g. in mounted volumes). Having stateless containers allow to horizontally scale by design, and enable a fast error fail over by re-creating failed containers.

* Configuration: Containers should ideally get their configuration at run time. Hence, the configuration should not be part of the container image but should be mapped (e.g. via environment variables) into the container at creation time.

What are pros/cons of containers compared to virtual machines?

Containers build on top of the host operating system and reuse its kernel as a run time environment for a applications or operating systems inside the container. Virtual machines are virtualized or (partially) emulated by a hypervisor. The host operating system (or bare metal hypervisor) is fully abstracted from the guest operating system inside a virtual machine.

Since virtual machines don't share software of the host, the encapsulation of virtual machines is rather strong. Stronger encapsulation affects the security/privacy aspect. Sharing on the other side allow to reuse existing software, to reduce the run time overhead and speed up the creation process.

* Pro containers: lower overhead, faster and less disk space * Contra containers: container cannot choose operating system / kernel * Pro virtual machines: very secure due to encapsulation, flexibly choose guest operating system * Contra virtual machines: overhead at runtime (two operating systems), waste of disk/memory space

Question: LXC Containers

Where are more processes running: on the vm operating system or inside the container? And why?

The vm operating system has much more processes, since the container is technically a ”jail” for at least one process. If the micro-service approach is used with containers, only a single (parent) process should be running inside the container.

How does the networking look like? Why is it a good idea to have a private IP for each containers?

Containers typically have their own ”virtualized” network. This network can be either bridged or tunneled through the host network of the operating system. The virtualized network is necessary to address containers, and to control which services of a container are accessible from the outside.

What resources can be limited via ‘lxc config set [container] limits.‘ command?*

Lxc uses *cgroups* to limit the resources. cgroups can limit all available resources by setting a soft and a hard limit. Resources may be: cpu cores, cpu usage (percent), memory usage, disk usage, disk i/o usage, network i/o usage.

Question: Docker and Docker Hub

Docker:

What are the differences between a Dockerfile and a Docker image? Can you imagine pros/cons?

A Dockerfile is a declarative definition of how to derive a Docker image. A Docker image is a binary snapshot of the container file system. While virtual machine images contain the full file system, Docker images have layers for each command defined in the Dockerfile. Docker images hence reuse identical layers from other Docker images.

Dockerfiles are text files, and hence very simple to share and archive. Docker images can be huge BLOBs, which makes it more challenging to share and archive. While Docker images contain all necessary software already, Dockerfiles may download and install software from third party sources. Images are hence more reliably and deterministic.

How does a typical workflow for deploying a new application component look like?

A DevOps writes software for a new application service, and writes a Dockerfile for it. Since he is smart, he works with git. When he pushed to git, the automated continuous delivery pipeline automatically compiles a Docker image from the Dockerfile and his software. The Docker image is then pushed to an image registry.

Next, a container can be started, using the image from the image registry.

Another option would be to apply changes to a container and then using the ‘docker commit‘ command to create the image. Since this is manual work, the previous path should be preferred.

Docker Hub:

Do you think it was useful to create an image for Apache by ourselves?

No. Try to reuse (at least the official) images. It was just useful for educational purposes :-) Of course, take care about the origin of the image. Everybody can upload to Docker Hub, there's no guarantee for security.

How are images in Docker Hub created and maintained?

Images can be created either by Dockerfiles or the ‘docker commit‘ command. Most if the images are generated and updated automatically from Dockerfiles. For example the official Apache image on Docker Hub [1] is stored on Github [2] and is build with Travis [3].

[1] https://hub.docker.com/_/httpd/

[2] https://github.com/docker-library/httpd

[3] https://travis-ci.org/docker-library/httpd

Lesson 2: Mediawiki with Docker

Questions: Experiences with Docker

Practically, where do you see benefits and drawbacks in the use of virtual machines versus Docker containers?

Docker containers are rather application component centric, virtual machines are operating system centric. Virtual machines are more flexible, when the purpose is unclear (e.g. not specified in a script or Dockerfile). Docker containers on the other hand are very lightweight and fast. While creating a new virtual machine can take up to some minutes, a Docker container is up within seconds.

Why are the two layers "Cloud Platform" and "Virtual Resources" still necessary although we have containers? *Or why are both layers not necessary when working with containers?*

A strong isolation between virtual machines is necessary to share the physical hardware with multiple users. While containers can limit resources as well, the encapsulation in virtualization is much stronger. Anyway Containers can be provided as a replacement of virtual machines in cloud middleware like OpenStack [1].

But: if the need for a simpler cloud stack arises, *Infrastructure as a Service* was probably the wrong choice. *Platform as a Service* may be better - where application components are deployed directly without thinking about the underlying infrastructure.

[1] https://wiki.openstack.org/wiki/HypervisorSupportMatrix

Question: Docker distributed

Can you find a solution to distribute Docker containers on multiple hosts?

So called *Orchestrators* are needed. Some examples are: *Docker Swarm*, Google's *Kubernetes*, Apache *Mesos*, Rancher

These orchestrators schedule the containers in available resources - virtual machines or bare metal servers.

Solution for practical part

Mediawiki with Docker

You should have the following Docker images:

“‘ REPOSITORY TAG IMAGE ID CREATED SIZE clouds/loadbalancer latest ... 2 hours ago 109 MB clouds/mediawiki latest ... 2 hours ago 412 MB clouds/database latest ... 3 hours ago 280 MB telegraf latest ... 3 days ago 239 MB ubuntu 14.04 ... 3 days ago 188 MB nginx latest ... 6 days ago 109 MB chronograf latest ... 2 weeks ago 112 MB influxdb latest ... 2 weeks ago 224 MB mariadb 5.5 ... 3 weeks ago 280 MB “‘

And you should have the following Docker containers:

“‘ ID IMAGE ... STATUS PORTS NAMES

“‘

Mediawiki with Docker Compose

To install docker compose, download the binary and make it executable:

“‘ sudo -s curl -L https://github.com/docker/compose/releases/download/1.13.0/docker-compose-‘uname -s‘-‘uname -m‘ > /usr/local/bin/docker-compose chmod +x /usr/local/bin/docker-compose “‘

The working ‘docker-compose.yaml‘:

“‘ version: ’2’ services: web1: build: Mediawiki image: clouds/mediawiki web2: build: Mediawiki image: clouds/mediawiki database: build: Database image: clouds/database loadbalancer: build: Loadbalancer image: clouds/loadbalancer ports: - 80:80 influxdb: image: influxdb chronograf: image: chronograf ports: - 8888:8888 telegraf: image: telegraf volumes: - /var/run/docker.sock:/var/run/docker.sock:ro - /sys:/rootfs/sys:ro - /proc:/rootfs/proc:ro - /etc:/rootfs/etc:ro - ./Monitoring/telegraf.conf:/etc/telegraf/telegraf.conf:ro environment: - HOST_PROC=/rootfs/proc - HOST_SYS=/rootfs/sys - HOST_ETC=/rootfs/etc “‘

To build the containers and start the services, run those commands:

“‘ docker-compose build docker-compose up “‘