Introduction to Cloud Computing and Containerisation

Niels Olof Bouvin

Overview

- **What is Cloud Computing**
- **Virtualisation and Containers**
- **Docker**

The Cloud?

- **Not *just* marketing-speak for someone else's computer**
  - (though it is that *too*)

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction."

[NIST Definition]

On-demand self-service

- **The user of the cloud service can add additional resources (computing, storage, network) as they wish, and when they wish directly through an interface**
- **It might even be possible for the system to add additional resources automatically, depending on the configuration and service plan**
  - e.g., add more servers, if there is a spike in traffic

Broad access

- The cloud service is accessible through standard networking protocols

Resource pooling

- **The cloud provider can pool their resources, and provide access to their users dynamically**
- **Access is independent of location of the provided machine**
  - though it can be specified, e.g., “give me a server within EU”

Measured service

- The user is billed according to the use of resources
- This can be continually and dynamically monitored by the cloud provider and user

Overview

- What is Cloud Computing
- **Virtualisation and Containers**
- **Docker**

Other people's computers?

- **Cloud providers (Amazon, Microsoft, Google, etc) have *vast* hosting centres**
  - they have economy of scale to their advantage, so they can be quite cost-effective
- **You *might* rent or buy an entire server (typically at a colocation host), but usually you rent *instances,* i.e., virtual machines that abstract the hardware**
  - no capital cost for the hardware, no hardware to lose value, no staff to maintain it
  - very cheap getting started—at scale it of course becomes more expensive

The Cloud requires shared hosting

- **For the cloud providers to efficiently utilise their hardware, it is essential that multiple jobs can run simultaneously on the same hardware**
  - it reduces costs and is *much* better use of modern, powerful blade servers
- **However, what about data security? A company might unbeknownst be running their server jobs on the same (physical) machine as their unscrupulous competitors…**
  - they cannot all be permitted to be `root` on the same machine
  - so, a partitioning of some sort is necessary

Virtualisation

- **A VM appears to an application to be a genuine computer, but is in reality itself just a program abstracting the hardware as a virtual machine isolated from other VMs running on the same machine**
- **This eases hosting considerably, as it enables**
  - multiple VMs on a single machine ensures better utilisation of hardware
  - moving VMs from one machine to another should be transparent to the user
  - packaging a VM, and creating *n* number of copies
  - strict configuration control, where new VMs can be rolled out (or rolled back!)
  - flexible deployment—from the developer's machine to various hosting providers
  - eases scalability—if your system takes off, it is easy (assuming your architecture is sound) to add more instances dynamically to handle the increased load

Virtual machines

- **A virtual machine emulates a real, physical computer**
- **VMs are controlled by a *hypervisor* that governs the actual computer's resources**
- **A blank VM is a computer with a blank harddrive—to work, an operating system must be installed, along with supporting programs, etc.**
  - this makes VMs fairly large, and starting a VM is comparable to booting a computer
- **So, flexible, but heavy with baggage**
  - each carrying its own OS, etc.

Containerisation

- **An lightweight alternative to VMs**
- **A container has access to the host's operating system, but resides in 'user space', and be restricted/isolated**
- **A container 'image' need only specify the difference between itself and its host (or another container)**
  - rather than having an entire OS itself
  - the modularity is similar to building with LEGOs…
- **Containers are often highly specialised**
  - a container for the Web server, a container for the load balancer, a container for the database, …

Building a system for a purpose

- Any given application will have its share of other, required software components
- As software grows in complexity, so does the work of handling and maintaining these configurations
- If the configuration is not exactly right, the system might not behave as expected
- So, given a new server and a specific purpose, all the required components have to be precisely aligned
  - on the developers' machines
  - on the testers' machines
  - on the production servers

Security?

- **The containers are, in principle and *largely* in reality, isolated from each other**
  - modulus Meltdown, Spectre, . . .
- **But, you should not just run *any* image off a repository**
  - there have been cases, where backdoors have been present
  - stick to official and well-regarded builds

Overview

- What is Cloud Computing
- Virtualisation and Containers
- **Docker**

Docker

- **The most widespread system for containerisation**
  - first open source release in 2013
  - originally developed internally at a french Cloud hosting company
- **Began on Linux**
  - where it runs on everything from giant servers to small devices such as Raspberry Pi
  - and is even now available on macOS & Windows

Terminology

- **Image**
  - a template for an application containing all required components, quite likely built on top of a pre-existing image (the equivalent of an application stored on disk)
- **Container**
  - a running instance of an image (the equivalent of an application running in memory)
- **Docker Daemon**
  - manages containers and images
- **Registry**
  - a repository for images. hub.docker.com is a commonly used repository, but you can also use your own

Getting started with Docker

- **The mental model of running Docker containers is that they are each computers in their own right, doing their own thing**
- **We can programmatically specify how they should be configured and what they should be doing**
  - this is done through a 'Dockerfile' that builds the image according to specification
- **or, we can launch the container interactively, modify it as we see fit, and `commit` the changes we made into a new image**
  - less easy to reproduce, but handy nonetheless, especially for debugging

Hello World

- There are thousands of available Docker images, including the ever so useful Hello World program

```
ad0f38092cf2: Pull complete
Digest: sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299cac44aca35a85c90c5e3c7afacdc
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm32v7)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Building an image with commit

pi@seabreeze: ~ — ssh seabreeze | pi@seabreeze: ~ — ssh seabreeze

```
pi@seabreeze:~ $ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
mosquitto           arm                 e5dfc7d32cad        4 weeks ago         4.39MB
arm32v6/alpine      latest              4407218c8304        2 months ago        4.01MB
arm32v6/node        alpine              ed5e5747b6af        6 months ago        64MB
pi@seabreeze:~ $ docker help commit

Usage:  docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

Create a new image from a container's changes

Options:
  -a, --author string    Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
  -c, --change list      Apply Dockerfile instruction to the created image
  -m, --message string   Commit message
  -p, --pause            Pause container during commit (default true)
pi@seabreeze:~ $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORT
S                   NAMES
affe56e0e683        arm32v6/alpine      "/bin/sh"           34 seconds ago      Up 25 seconds
                    inspiring_wescoff
pi@seabreeze:~ $ docker commit inspiring_wescoff bouvin/my_file
```

Containers stick around

- **Using the command '`docker ps -a`', you can see all containers (running or stopped)**
  - you can delete all stopped containers with '`docker container prune`'
- **In the previous example, I could have exited, and used '`docker restart inspiring_wescoff`' to restart the container (still with the /my_file in it)**
  - and executed commands in the running container with '`docker exec`'
  - and stopped it using '`docker stop inspiring_wescoff`'

Building an image through a Dockerfile

```
FROM arm32v6/alpine:latest

RUN touch /my_file
```

```
pi@seabreeze:~/Development/Courses/dBIoTP2PC/docker-01 $ ls -l
total 4
-rw-r--r-- 1 pi pi 47 Sep  6 16:37 Dockerfile
pi@seabreeze:~/Development/Courses/dBIoTP2PC/docker-01 $ cat Dockerfile
FROM arm32v6/alpine:latest

RUN touch /my_file
pi@seabreeze:~/Development/Courses/dBIoTP2PC/docker-01 $ docker build -t bouvin/my_file .
```

Building with Dockerfiles

- Files can be copied into the image
- Commands can be executed as part of the image build
- Network ports specified for communications
- Environment variables set
- Commands to be run, when the container starts
- Etc.

Delving into Docker

- Docker builds images from Dockerfiles, creates containers from images, and can create images from containers
- All saved by Docker as layers of LEGO
- If you need a fresh container, run the original image
- If you need the old data, restart a stopped container

Containers are isolated by default

- **Docker containers are little computers running inside a bigger computer**
- **They are by default cut off the world**
  - they cannot access the file system of the computer running the container
  - they have no network access: all ports are closed to them unless specified otherwise
- **Use '`docker run -p outside:inside`' to access the inside port at the outside port on your computer**
  - i.e., '`docker run -p 80:8080 fancy-pants/iot-assignment`' makes the port 8080 (that Michal's program is listening to inside its Docker container) accessible from outside at port 80

Summary

- **Containerisation in general, and Docker specifically, grew out of DevOps: Development and Operations**
  - too often, subtle differences between developers' and production machines would lead to trouble
- **With Docker, you can neatly encapsulate an application with all its requirements and dependencies**
  - images builds on other images
  - containers can be started, stopped, and restarted