* This cloud model is composed of
  + Five essential characteristics
  + Three service models, and
  + Four deployment models.

NIST: Five essential characteristics of cloud computing

* C1. On-demand self-service
* C2. Broad network access
* C3. Resource pooling
* C4. Rapid elasticity
* C5. Measured ser

Enabling technologies (equal order of importance)

* Broadband networks and Internet architecture
* Data center technology
* Virtualization technology
* Web technology
* Multi-tenant technology

## What is GENI

* Global Environment for Networking Innovation
* Combining heterogeneous resource types, each virtualized along one or more suitable dimensions, to produce a single platform for network science researchers”
* Key components:
  + GENI racks: virtualized computation and storage resources
  + Software-defined networks (SDNs): virtualized, programmable network resources
  + WiMAX: virtualized cellular wireless communication

## Key experimental concepts

* Sliceability: the ability to support virtualization while maintaining some degree of isolation for simultaneous experiments
* Deep programmability: the ability to influence the behavior of computing, storage, routing, and forwarding components deep inside the network, not just at or near the network edge.

## What services does Cloud offer?

* Before we can evaluate the necessity of moving to the cloud, we need to to know what services are available.
  + SaaS: Software-as-a-Service
  + PaaS: Platform-as-a-Service
  + IaaS: Infrastructure-as-a-Service

## SaaS: Software-as-a-Service

* Vendor controlled applications that are accessed over the network by users.
* Characteristics:
  + Network-based access
  + Multi-tenancy
  + Single software release for all
* Examples:
  + Applications in the Google Suite
  + Dropbox

## SaaS: Application Design

* Net native
  + Cloud-specific design, development, and deployment
  + Multi-tenant data
  + Built-in metering and management
  + Browser-based
  + Customization via configuration
* High degree of configurability, efficiency, and scalability

## SaaS: Disadvantages

* SaaS providers are dependent on network and cloud service providers.
  + [A Dropbox story](https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/)
* Performance is dependent on individual client’s bandwidth.
* Security
  + Good: Better security than personal computers
  + Bad: SaaS vendors (and cloud providers) are in charge of the data
  + Ugly: Privacy

## PaaS: Platform-as-a-Service

* Vendors provide development environment.
  + Tools and technologies are selected by vendors.
  + Users maintain control over data (application) life-cycle.
* Examples:
  + Google App Engine
  + AWS Elastic Beanstalk
  + Heroku

## PaaS: Architectural characteristics

* Support multi-tenancy at various scale: sessions, processes, and data.
  + Isolation at: physical, virtual, and logical levels
  + [Microsoft’s offerings of isolation choices](https://docs.microsoft.com/en-us/azure/security/fundamentals/isolation-choices)
* Native scalability
  + Load balancing and fail-over (AWS Elastic Beanstalk)
* Native integrated management
  + Performance
  + Resource consumption/utilization
  + Load

## PaaS: Disadvantages

* Inherits all from SaaS
* Options on technologies and tools are limited by the PaaS vendors

## IaaS: Infrastructure-as-a-Service

* Vendors provide computing resources.
* Users provision computing resources.
  + Compute resources include processing, storage, memory, network etc.
  + Users are provided with customized virtual machines.
* Users maintain control over:
  + Operating system, memory
  + Storage,
  + Servers and deployment configurations, and
  + Some limited control over network resources via software-defined networking

## IaaS: Advantages

* Infrastructure scalability
* Native-integrated management via vendors’ utilities
  + Performance, resource consumption/utilization, load
* Economical cost
  + Hardware, IT support

## IaaS: Disadvantages

* Require more technical efforts than SaaS and PaaS.

Flexibility

Graphical user interface, application

Description automatically generated

## XaaS: Everything-as-a-Service

* Composite second level services
* [NIST Evaluation of Cloud Computing Services (2018) p. 20](https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-322.pdf)

## NIST: Four deployment models

* Private Cloud
* Community Cloud
* Public Cloud
* Hybrid Cloud

## Private cloud

* Infrastructure is organized solely for an organization
* Infrastructure is managed by the organization or by a third party

## Community cloud

* Supports a specific community
* Infrastructure is shared by several organizations
* Examples: CloudLab

## Public cloud

* Infrastructure is made available to the general public
* Infrastructure is owned by an organization selling cloud services
* Example: Azure Notebook free tier.

## Hybrid cloud

* Infrastructure is a composition of two or more clouds deployment models.
* Enables data and application portability

## Cloud Security: who is doing what

* The cloud provider is responsible for the security **OF** the Cloud.
* The cloud consumer (users) is responsible for the security **IN** the Cloud.

## Cloud consumer

* SaaS/PaaS:
  + Standard security procedure for online presences.
* IaaS:
  + Standard security procedure as any on-premise infrastructures.
  + Benefits from native administrative tools from the Cloud Provider.

## Cloud provider: SaaS security

* SaaS:
  + Web application security: [OWASP’s Top 10](https://owasp.org/www-project-top-ten/)
  + Multi-tenancy: data isolation/leakage
  + Data security: accessibility versus security trade-off

## Cloud provider: PaaS security

* Similar security concerns as SaaS
* Complex security schemes due to potential third-party relationships.
* Development Lifecycle
  + Users depend on PaaS providers to patch security issues of the individual tools.

## 22. Cloud provider: IaaS security

* Standard security measures.
  + To Cloud Provider, cloud resources are on-premise.
* Concerns with virtual machines’ security
* Concerns with virtual networking security

## What is virtualization?

* Operating System concept: The abstraction of available resources
* **Virtualization technologies encompass a variety of mechanisms and techniques used to address computer system problems such as security, performance, and reliability by decoupling the architecture and user-perceived behavior of hardware and software resources from their physical implementation.**

## Virtualization

* [Formal requirements for virtualizeable third generation architectures](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.141.4815&rep=rep1&type=pdf)
* A virtual machine is taken to be an efficient, isolated duplicate of the real machine.
* These notions can be explained through the idea of a **virtual machine monitor**.
* Essential characteristics of VMM:
  + Essentially identical to the physical resource
  + Efficiency
  + Complete control of system resources (with regard to the processes running inside the VM)

## Virtualization

* Virtualization Layer: The Virtual Machine Monitor (or its modern name: **Hypervisor**) provides an interface between hardware and virtual operating systems.
* Type of hypervisors:
  + Bare-metal
  + Hosted

## Issues that virtualization can help with

* Under-utilized resources
* Complicated system management
* Limited access to shared resources
* Inefficient power consumption
* Tight coupling with underlying resource

## Types of virtualization

* Platform Virtualization
* Memory Virtualization
* Desktop Virtualization
* Application Virtualization
* Network Virtualization
* Storage Virtualization

## Platform virtualization

* Full Virtualization
* Para Virtualization
* Hardware assisted virtualization
* OS level virtualization

## Without virtualization

* x86 offers four levels of privilege (Ring 0 through 3)
* OS needs to have access to hardware and run on ring 0
* Application runs on ring 3, gain access to hardware by trapping into kernel mode for privileged instructions.
* Virtualizing x86 requires a layer under OS (which already at lowest level) to create and manage the VM
* Sensitive instructions must be executed in ring 0

## Full virtualization

* Guess OS is unaware of host OS.
  + VMM provides virtual BIOS, virtual devices, and virtual memory management.
* Non-critical instructions run directly on hardware.
* Runtime translation of critical non-virtualizable instructions happens in the hypervisor.
* Provide best isolation and security at the cost of performance.

## Para virtualization

* Thin layer interfaces between each guest OS and underlying hardware.
* Need guest kernel modification.
* No need of runtime translation for critical instructions.
* Superior in performance.
* Requires expertise to patch the kernels.

## Hardware-assisted virtualization

* Hardware provides support to run instructions independently.
  + Intel Virtualization Technology (VT-x)
  + AMD Virtualization Technology (AMD-V)
* No need to patch the kernels.
* Runtime translation not required.
* Better performance in comparison to other variants.
* Greater stability

## Virtualization at OS level

* Same OS for both host and guest machines.
* User space is completely isolated.
* High performance.
* Extremely light-weight.

## Memory virtualization

* How to share physical system memory and dynamically allocating it to virtual machines.
* Guess OS maps virtual memory space (of VM) to physical memory space (of VM).
* VMM translates physical memory space (of VM) to physical memory space (of main machine), but also enables direct mapping (shadow table) to avoid overhead.
* Summary

|  | **Full Virtualization with Binary Translation** | **Hardware Assisted Virtualization** | **OS Assisted Virtualization/Para Virtualization** |
| --- | --- | --- | --- |
| Guest modification/Compatibility | Unmodified Guest OS, excellent compatibility | Unmodified Guest OS, excellent compatibility | Guest OS codified to run Hypercall, cannot run of native hardware or other hypervisors. Poort compatibility |
| Performance | Good | Fair | Better on certain cases |
| Guest OS Hypervisor Independent | Yes | Yes | Xen Linux runs only on Xen Hypervisor. VMI-Linux is Hypervisor agnostic |

## Desktop and application virtualization

* Desktop and Applications run on servers.
* Stateless thin clients connected to servers.
* Efficient system management.
* Requires high-end servers for system stability

## Network and storage virtualization

* Similar idea of providing an abstraction layer to the physical infrastructures
* In networks, abstraction will
  + Be at the level of routers, switches, gateway, firewalls, load balancers, …
  + Enabled by software-defined networking
* In storage, single storage backends can be used for different requirements
  + Ephemeral
  + Persistent
  + Specialize storage backends

## Virtualization: concept of overcommits

* Allocating more than the available physical resources to the Guest OS
* Common types of overcommit:
  + CPU
  + Memory
  + Storage

## Virtualization: concept of overcommits

* Advantages:
  + Favorable economic model
  + Efficient resource utilization
  + Support green computing
* Disadvantages:
  + Performance loss or unstable system response
  + Complex system understanding
  + VM shutdown by the hypervisor

## Virtualization: CPU and memory overcommits

* Allows more virtual CPUs than physically available
  + Openstack KVM: overcommit-number = 16.0
* Allow more memory than physically available
  + Overstack KVM: overcommit-number = 1.5GB

## Virtualization hypervisors

* Contribution from industry and academia
* Xen: Project from Cambridge Computer Laboratory
* VMware: Commercial product
  + Also comes from academic research (see Mendel Rosenblum ACM)
* KVM: Initiated by the Open Virtualization Alliance, later dissolved and is now managed by the Linux Foundation
* Qemu: Open source machine emulator and virtualizer

## Openstack: Neutron

* Management: internal comm between OpenStack components, reachable only within the data center.
* Guest: Used for VM data communication within the Cloud Deployment.
* External: Provide VM with Internet access.
* API: Exposed all the Stack’s API to the public.

## Setup Apache webserver (from the volume-based Alpine from Challenge 8)

* You should be inside the console after log in as root and have the root password.
* Run the following commands to install Apache webserver

$ apk update

$ apk add apache2

$ rc-service apache2 start

## Cloud security basic

* In the cloud, egress means traffic that’s leaving from inside the private network out to the public internet (similar to standard network definition).
* In the cloud, ingress refers to unsolicited traffic sent from an address in public internet to the private network – it is not a response to a request initiated by an inside system. In this case, firewalls are designed to decline this request unless there are specific policy and configuration that allows ingress connections.

## History of KVM

* Created by Avi Kivity at Qumranet
* Become part of the Linux kernel in 2007
* Hardware-based virtualization

## History of KVM

* Virtualizing the x86 architecture (Intel and AMD)
* Hardware vendors provide the following extensions to the x86 architectures
  + A new guest operating mode: the CPU can switch into a guest mode that allows system software to trap specific privileged instructions or accesses.
  + Hardware state switch: Hardware is responsible for switching the control registers that affect the CPU operation modes and relevant segment registers.
  + Exit reason reporting: When a switch from guest to host happens, the hardware reports the reason so that the software can take appropriate responses.
* KVM has been developed to take advantage of these extensions.

## General architecture

* Virtual machines are created by opening a device node (/dev/kvm) - essentially a file.
* Operations provides by /dev/kvm includes:
  + Create a new VM
  + Allocate memory to a VM
  + Reading and writing virtual CPU registers
  + Injecting an interrupt into a virtual CPU
  + Running a virtual CPU

## Memory mapping

* Kernel allocates discontiguous pages to form the Guest address space.
* User address space (of processes inside the VM) can mmap directly to these allocation to provide direct-memory-access capability.

## CPU execution

* A new mode is added called guest mode.
* User space calls the kernel to execute guest code.
* Kernel causes hardware to enter guest mode and handles interrupts when CPU exits guest mode.
* Hardware execute guest code until it encounters instructions that needs assistance.

## Virtualizing the MMU (memory management unit)

* MMU: Translate virtual address space (from users’ perspective) to physical addresses.
  + Page table: virtual-to-physical translation, located in hardware registers
  + Page faults mechanism (missing translation)
  + On-chip cache (translation lookaside buffer) that accelerates lookup of the page table.
* Hardware support mmu virtualization:
  + Provides support for guest\_virtual to guest\_physical
  + Does not account for guest\_physical to host\_physical
* Solutions: shadow page table that performs the later task.

## Initial implementation

* Less complicated, but also sacrificed performance.
* Copy the contents changes due to guest instructions that require access to TLB’s page tables.
* Most common TLB instruction is context\_switch, which invalidate the entire TLB (impacting shadow table).
* Rebuilding the shadow table is expensive.
* Impact guest’s workload with multiple processes

## Caching virtual mmu

* Virtual MMU implementation (from hardware vendor) was enhanced to allow page tables to be cached across context switches.
* Guest memory pages (on physical host) that are shadowed by KVM need to be write-protected.
* Write access to guest page table needs to be emulated (rather than trying to use the host x86 instructions) so that we know exactly what happens to the guest memory and to the shadow page table.

## I/O Virtualization

* Software uses programmed I/O (pio) and memory mapped I/O (mmio) to communicate with hardware devices.
* KVM needs to be able to trap and emulate pio and mmio request (from guest to physical host) and simulate interrupts from virtual hardware to programs running in guest.

## Why

* Essential characteristics of VMM:
  + Essentially identical to the physical resource
  + Efficiency
  + Complete control of system resources (with regard to the processes running inside the VM)
* Support advanced complex devices in modern computers
  + Graphical devices
  + High speed storage devices
* Benefits of VM
  + Better resource pooling
    - One physical machine divided into multiple VMs
  + Easier to scale
  + VMs in the Cloud
    - Rapid elasticity
    - Pay as you go model
* Limitation of VM
  + Each VM still requires
    - CPU allocation
    - Storage
    - RAM
    - An entire guest OS
* The more VM you runs, the more overhead resources you need
* Guest OS means wasted resources
* Application portability is not guaranteed

## What is a container

* Standardized packaging for software and dependencies
* Isolate apps from each other
* Share the same OS kernel
* Work with all major Linux and Windows server

## Key benefits of Dockers

* Speed
  + No OS to boot
* Portability
  + Less dependencies between process layers
* Efficiency
  + Less OS overhead
  + Improved VM density

## Singularity

* Begin as an open source project in 2015 at Lawrence Berkeley National Laboratory.
* Released until BSD License
* Top 5 new technologies to watch in 2016-2017 (HPCWire)
* Being used in academic:
  + Standard University Research Computing Center
  + National Institute of Health
  + 60% of the Open Science Grid Consortium.
  + Ohio State, Michigan State, TACC, SDSC, and Oak Ridge National Lab
* Supports for native high performance interconnects (InfiniBand, OmniPath)
* Supports for native graphic accelerators
* Supports for Open MPI, including a hybrid mode (inside and outside of containers)
* Useful for new advanced research areas such as ML, Deep Learning, and data-intensive workloads

## Motivation: container software unit for scaled science

* Current approaches suitable for industry’s micro-service virtualization and web-enabled cloud application.
  + Wiki: Microservices are a software development technique—a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight.
* Not quite suitable for scientific world, and HPC communities.
* The reproducibility and portability aspects of containers are highly desirable.
* Security level of Docker is not good enough for a shared research environment.

## The needs of scientists

* Technological innovation of container-based environments
* The needs for scalable and reproducible products
* Preference for usability
* Necessity to operate on everything from laptops to large-scale HPC
* Before: Scientists exchange data files and source codes
* Today: Scientists exchange workflows
* These two things need to be encapsulated into a single computing environment
* Challenges with Docker:
  + Security concerns: Docker’s containers are spawned as a child of a root-owned Docker daemon
    - Potential issue: privilege leakage
  + Lack of administrative control in what’s being run/monitored (opposite of an industry environment)

## What Singularity aims to solve

* Specific accomplishments
  + Mobility of Compute
    - Distributable image format that encapsulates the entire container and software stack into a single image file.
  + Reproducibility
    - Utilize single files, which allow snapshot, archive, and lock-down for reusability purposes.
    - No external influence from the host OS.
  + User freedom
    - Can install any relevant dependencies inside the system without worrying about the host OS.
  + Support existing traditional HPC resources.

## Performance evaluation (Singularity is not included)

* Always need to look at latest papers, as software have been improving over time.
* Felter, W., Ferreira, A., Rajamony, R. and Rubio, J., 2015, March. An updated performance comparison of virtual machines and linux containers. In 2015 IEEE international symposium on performance analysis of systems and software (ISPASS) (pp. 171-172). IEEE.
* System setting:
  + Dual 2.4-3.0 GHz Intel Sandy Bridge Xeon E5-2665 processors (16 cores total)
  + 256GB of RAM
  + Ubuntu 13.10 (Linux kernel 3.11.0)
  + Docker 1.0
  + QEMU 1.5.0
  + Libvirt 1.1.1
* Benchmark
  + CPU Benchmark: Linpack
  + Memory bandwidth: Stream
  + Random memory access: RandomAccess
  + Network bandwidth: nuttcp
  + Network latency: netperf
  + Block I/O: fio (20TB IBM Flash system SSD)
  + Storage: Redis
  + Database: MySQL
* CPU benchmark

Chart, bar chart

Description automatically generated

* Memory benchmarkChart, bar chart

  Description automatically generated
* Random memory access

Chart, bar chart

Description automatically generated

* Network bandwidth

Chart, box and whisker chart

Description automatically generated

* Network latency

Chart, bar chart, histogram

Description automatically generated

## Podman

* Container engine developed by RedHat.
* Addressing issues with Docker
  + Persistent daemon on host.
  + Root/privileged concerns
* OCI: Open Container Initiative (2015).
* Podman
  + Client-only tool, based on Docker CLI
  + No Daemon
  + Similar CLI experience as Docker CLI
  + Build and run containers as non-root
  + Simple CLI, no client-server architecture.

## Setup

* Go to your GitHub project repository (on the first day), create a new branch called docker from the main branch, and modify to add the following components from [this link](https://github.com/CSC468-WCU/csc468cloud/tree/docker):
  + Borg facilitatesThe docker\_config directory and its content (daemon.json).
  + The install\_docker.sh file.
  + The profile.py file.
* **Check and make sure all the contents are correctly copied!**
* Go to CloudLab, open your profile, switch to Edit mode and click Update. The new docker branch should show up.
* Instantiate an experiment from this branch.
* **Only login after the Startup column becomes Finished** and type the following command: sudo docker info | grep "Docker Root Dir"
* Confirm that you have something similar to the screenshot below

## A shipping container system for applications

Diagram

Description automatically generated

## Hello world

* Docker containers are instantiated from Docker images.
* You can check availability of local images and containers.

$ docker image ls

$ docker container ls

* We can issue the following to start a service that will echo hello world to the screen.
* This requires a Linux container to run the echo command.

$ docker run alpine echo hello world

* docker: invoke the container engine.
* run: subcommand to run a container.
* alpine: name of the image based on which a container will be launched.
* echo hello world: the command to be executed in the container environment.

$ docker image ls

$ docker container ls

$ docker container ls **--all**

$ docker run alpine echo hello world

$ docker container ls **--all**

## Interactive container

* We can launch a container and get into the shell of the container.

$ docker run **-it** ubuntu bash

* You are now in a new prompt: a shell inside the container
* -it: combination of -i and -t.
  + -i tells Docker to connect to the container’s stdin for interactive mode
  + -t tells Docker that we want a pseudo-terminal

## Docker images

* Image = files + metadata
* The files form the root filesystem of the container
* The metadata describes things such as:
  + The author of the image
  + The command to execute in container when starting it
  + Environment variables to be set
  + …
* Images are made of layers, conceptually stacked on top of each other.
* Each layer can add, change, and remove files and/or metadata.
* Images can share layers to optimize disk usage, transfer times, and memory use.

## Example of a Java webapp

* CentOS base layer
* Packages and configuration files added by our local IT
* JRE
* Tomcat
* Our application’s dependencies
* Our application code and assets
* Our application configuration

## Containers versus images

* An image is a read-only filesystem.
* A container is an encapsulated set of processes running in a read-write copy of that filesystem.
* To optimize container boot time, copy-on-write is used instead of regular copy.
* docker run starts a container from a given image.

Diagram

Description automatically generated

* Object-oriented analogy
  + Images are conceptually similar to classes
  + Layers are conceptually similar to inheritance

## How do we change an image?

* It is read-only, we don’t.
* We create a new container from the image
* We make changes to the container.
* When we are satisfied with the changes, we transform them into a new layer.
* A new image is created by stacking the new layer on top of the old image.

## Image namespaces

* Official images (ubuntu, busybox, …)
  + Root namespace.
  + Small, distro images to be used as bases for the building process.
  + Ready-to-use components and services (redis, postgresl …)
* User (and organizations) images: <registry\_name>/<image\_name>:[version]
  + jpetazzo/clock:latest
  + linhbngo/csc331:latest
* Self-hosted images
  + Images hosted by third party registry
  + URL/<image\_name>

## show current images

* If this is a new experiment, go ahead and run the following commands to get some images loaded.

$ docker run hello-world

$ docker run alpine echo This is alpine

$ docker run ubuntu echo This is ubuntu

$ docker image ls

## search images

* We can search for available images in the public Docker Hub

$ docker search mysql

## General steps to create an image

* Create a container using an appropriate base distro
* Inside the container, install and setup the necessary software
* Review the changes in the container
* Turn the container into a new image
* Tag the image

## create a container with a base distro

* Remember to note your container ID.

$ docker run **-it** ubuntu

## install software inside the container

*# apt-get update*

*# apt-get install -y figlet*

*# exit*

## check for differences

* Remember to note your container ID.

## commit changes into a new image

* Remember to note your container ID.

$ docker commit 16b0 ubuntu\_figlet\_$USER

$ docker image ls

$ docker history fe101

* the docker commit ... command created a new image named ubuntu\_figlet\_lngo that has the following unique id: fe101865e2ed.
* The docker image ls command shows this image.
* The docker history fe101 shows the layers making up this image, which include the layer that is the base ubuntu image 54c9d.

## Automatic image construction: Dockerfile

* A build recipe for a container image.
* Contains a series of instructions telling Docker/Podman how an image is to be constructed.
* The docker build command builds an image from a Dockerfile.

## writing the first Dockerfile

* The following commands are done in the terminal (Ubuntu WSL on Windows/Mac Terminal).

$ cd

$ mkdir myimage

$ cd myimage

$ nano Dockerfile

* Type the following contents into the nano editor

|  |  |
| --- | --- |
|  | FROM ubuntu |
|  | RUN apt-get update |
|  | RUN apt-get install -y figlet |

* FROM: the base image for the build
* RUN: represents one layer of execution.
* RUN commands must be non-interactive.
* Save and quit after you are done.
* To build the image

## build the image

* The following commands are done in the terminal (Ubuntu WSL on Windows/Mac Terminal).
* Check that you are still inside myimage

$ pwd

$ docker build **-t** figlet\_$USER .

* -t indicates a tag named figlet will be applied to the image.
* . indicates that the Dockerfile file is in the current directory.
* The build context is the Dockerfile file in the current directory (.) and is sent to the container engine. This context allows constructions of images with additional resources from local files inside the build context.
* The base image is Ubuntu.
* For each RUN statement, a container is created from the base image for the execution of the
* commands. Afterward, the resulting container is committed into an image that becomes the base for the next RUN.
* Use docker image ls and docker history ... to check which layer is reused for this image.
* Test run the new ubuntu\_figlet image by launching an interactive container using this image, then immediately run figlet hello world.

## CMD

* Edit your Dockerfile so that it has the following content

|  |  |
| --- | --- |
|  | FROM ubuntu |
|  | RUN apt-get update |
|  | RUN apt-get install -y figlet |
|  | CMD figlet -f script hello |

[**view raw**](https://gist.github.com/linhbngo/b9f794bed306562f2eb85da310ae7b5e/raw/a4782365f85b7b635b3211a782f2b67f5e67f1aa/Dockerfile.2)[**Dockerfile.2**](https://gist.github.com/linhbngo/b9f794bed306562f2eb85da310ae7b5e#file-dockerfile-2)hosted with ❤ by [**GitHub**](https://github.com/)

* CMD: The command to be run if the container is invoked without any command.
* Rebuild the image with the tag figlet\_cmd\_$USER.
* Run the following command

$ docker run figlet\_cmd\_$USER

## Overriding CMD

* With CMD, the -it flag does not behave as expected without a parameter.
* To override CMD, we can provide a command

$ docker run **-it** figlet\_cmd\_$USER

$ docker run **-it** figlet\_cmd\_$USER bash

## ENTRYPOINT

-ENTRYPOINT defines a base command (and its parameters) for the container.

* The command line arguments are appended to those parameters.
* Edit Dockerfile as follows:

|  |  |
| --- | --- |
|  | FROM ubuntu |
|  | RUN apt-get update |
|  | RUN apt-get install -y figlet |
|  | ENTRYPOINT ["figlet","-f", "script"] |

* Rebuild the image with the tag figlet\_entry\_$USER.
* Run the followings:

$ docker run figlet\_entry\_$USER golden rams

## Why not both

* ENTRYPOINT and CMD can be used together.
* The command line arguments are appended to those parameters.
* Edit Dockerfile as follows:

|  |  |
| --- | --- |
|  | FROM ubuntu |
|  | RUN apt-get update |
|  | RUN apt-get install -y figlet |
|  | ENTRYPOINT ["figlet","-f", "script"] |
|  | CMD ["hello rammies"] |

* Rebuild the image with the tag figlet\_both\_$USER.
* Run the followings:

$ docker run figlet\_both\_$USER golden rams

$ docker run figlet\_both\_$USER

## Caveat

* /bin/bash does not work as expected.

$ docker run **-it** figlet\_both\_$USER bash

$ docker run **-it** **--entrypoint** bash figlet\_both\_$USER

*# exit*

## Importing and building external code

* Create the following file called hello.c:

|  |  |
| --- | --- |
|  | #include <stdio.h> |
|  |  |
|  | int main() { |
|  | printf("Hello, world!\n"); |
|  | return 0; |
|  | } |

* Create the following Dockerfile called Dockerfile.hello:

|  |  |
| --- | --- |
|  | FROM ubuntu |
|  | RUN apt-get update |
|  | RUN apt-get install -y build-essential |
|  | COPY hello.c / |
|  | RUN gcc -o hello hello.c |
|  | CMD /hello |

* You can build an image with a specific Dockerfile

$ docker build **-t** hello\_$USER **-f** Dockerfile.hello .

$ docker run hello\_$USER

## a simple web server

$ docker run **-d** **-P** nginx

$ docker ps

* -P: make this service reachable from other computers (--publish-all)
* -d : run in background

## How does the container engine know which port to map?

* This is described in the Dockerfile and can be inspected.
* The keyword for this action is EXPOSE.
* Why do we have to map ports?
  + Containers cannot have public IPv4 addresses.
  + We are running low on IPv4 addresses anyway.
  + Internally to host, containers have their own private addresses
    - Services have to be exposed port by port.
    - These have to be mapped to avoid conflicts.

## manual allocation of port numbers

$ docker run **-d** **-p** 8000:80 nginx

$ docker run **-d** **-p** 8080:80 **-p** 8888:80 nginx

* Convention: port-on-host:port-on-container
* Check out the web servers at all of these ports

## integrating containers into your infrastructure

* Manually add the containers to the infrastructure via container-generated public port.
* Predetermine a port on the infrastructure, then set the corresponding port mapping when run the containers.
* Use a network plugin to connect the containers with network tunnels/VLANS …
* Deploy containers across a physical cluster using Kubernetes.

## Container network model

* Provide the notion of a network to connect containers
* Provide top level command to manipulate and observe these networks:
  + docker network

$ docker network

$ docker network ls

* What’s in a container network?
  + Conceptually, it is a virtual switch
  + It can be local to a single Engine (on a single host) or global (spanning multiple hosts).
  + It has an associated IP subnet.
  + The container engine will allocate IP addresses to the containers connected to a network.
  + Containers can be connected to multiple networks.
  + Containers can be given per-network names and aliases.
  + The name and aliases can be resolved via an embedded DNS server.

## create a network

$ docker network create ramnet

$ docker network ls

## placing containers on a network

$ docker run **-d** **--name** es **--net** ramnet elasticsearch:2

$ docker run **-it** **--net** ramnet alpine sh

*# ping es*

*# exit*

## Docker compose

* Dockerfiles are great for building container images.
* Dockerfiles are not quite satisfactory if you have to link multiple containers into a complex infrastructure.
* We want the ability to write custom scripts (program everything!) to automatically build, run, and connect containers together.
* This is possible via Docker Compose.
* For Podman, it is called Buildah.

## In a nutshell

* External, Python-based tool.
* Open source.
* Simple deployment workflow
  + Checkout code
  + Run docker-compose up
  + Everything is up and running

## Overview of compose

* Design of a container stack is described in a YAML file called docker-compose.yml.
* Run docker-compose up.
* Compose automatically pulls images, builds containers, and starts them.
* Compose can
  + Set up links, volumes, and other Docker options for the containe

## Docker compose demonstration

* Run the following commands:

$ cd

$ git clone https://github.com/CSC468-WCU/ram\_coin.git

$ cd ram\_coin

$ docker-compose up

* Visit YOUR\_CLOUDLAB\_HEADNODE:8000 to see the deployed webserver.
* Does it work?
* Open another terminal, connect to your CloudLab headnode and run docker ps to see how many containers were deployed by the docker-compose.
* Press Ctrl-C to stop the containers.

## Sections of a compose file

* Use cat or nano to view docker-compose.yaml file.
* version is mandatory (“2” or later).
* services is mandatory. A service is one or more replicas of the same image running as containers.
* networks is optional and indicates to which networks containers should be connected. By default, containers will be connected on a private, per-compose-file network.
* volumes is optional and can define volumes to be used and/or shared by the containers.

## Compose file versions

* Version 1 is legacy.
* Version 2 has support for networks and volumes.
* Version 3 has support for deployment options.

## Containers in docker-compose.yaml

* Each service in the YAML file must container either build or image.
* build indicates a path containing a Dockerfile.
* image indicates an image name (local or on registry).
* If both are specified, an image will be built from the build directory and named image
* Other parameters are optional and typically what you would add to docker run
  + command = CMD
  + ports = -p
  + volumes = -v

## Rerun ram\_coin in background

$ docker-compose **-d** up

$ docker-compose ps

## cleanup

$ docker-compose kill

$ docker-compose rm

* Once the experiment is fully deployed, **and all Startup Finished running**:
  + SSH into the head node and run the followings

$ cd

$ bash /local/repository/launch\_network.sh

$ kubectl get nodes

## Automated Kubernetes Deployment

* This is done via Kubernetes Objects, described through YAML files.
* **Kubernetes objects** are persistent entities in the Kubernetes system, which represent the state of your cluster.
  + What containerized applications are running (and on which nodes)
  + The resources available to those applications
  + The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance
* A Kubernetes object is a “record of intent”–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you’re effectively telling the Kubernetes system what you want your cluster’s workload to look like; this is your cluster’s desired state.

## Sequence of commands to launch ram\_coin on Kubernetes

* First, we deploy a registry service. This is equivalent to a local version of Docker Hub.

$ cd

$ kubectl create deployment registry **--image**=registry

$ kubectl expose deploy/registry **--port**=5000 **--type**=NodePort

$ kubectl get svc

* We can patch configurations of deployed services

$ kubectl patch service registry **--type**='json' **--patch**='[{"op": "replace", "path": "/spec/ports/0/nodePort", "value":30000}]'

$ kubectl get svc

## Building and pushing images for ramcoin

* We test our local registry by pulling busybox from Docker Hub and then tag/push it to our local registry.

$ docker pull busybox

$ docker tag busybox 127.0.0.1:30000/busybox

$ docker push 127.0.0.1:30000/busybox

$ curl 127.0.0.1:30000/v2/\_catalog

* Next, we clone ramcoin repository

$ git clone https://github.com/CSC468-WCU/ram\_coin.git

$ cd ~/ram\_coin

$ docker-compose **-f** docker-compose.images.yml build

$ docker-compose **-f** docker-compose.images.yml push

$ curl 127.0.0.1:30000/v2/\_catalog

$ kubectl create deployment redis **--image**=redis

$ **for** SERVICE **in** hasher rng webui worker; **do** kubectl create deployment $SERVICE **--image**=127.0.0.1:30000/$SERVICE:v0.1; **done**

$ kubectl expose deployment redis **--port** 6379

$ kubectl expose deployment rng **--port** 80

$ kubectl expose deployment hasher **--port** 80

$ kubectl expose deploy/webui **--type**=NodePort **--port**=80

$ kubectl get svc

* svc is abbreviation for services.
* You can see the difference between services and pods

$ kubectl get services

$ kubectl get pods

## Setup a Kubernetes Dashboard

* Run the following commands from inside the ram\_coin directory

$ kubectl apply **-f** dashboard-insecure.yaml

$ kubectl apply **-f** socat.yaml

$ kubectl get namespace

$ kubectl get svc **--namespace**=kubernetes-dashboard

$ kubectl patch service kubernetes-dashboard **-n** kubernetes-dashboard **--type**='json' **--patch**='[{"op": "replace", "path": "/spec/ports/0/nodePort", "value":30082}]'

* Go to the head node URL at port 30082 for kubernetes-dashboard
* Hit skip to omit security (**don’t do that at your job!**).

## Kubernetes namespace

* Provides a mechanism for isolating groups of resources within a single cluster.
* Uniqueness is enforced only within a single namespace for namespaced objects (Deployment and Services)
* Uniquess of other cluster-wide objects (StorageClass, Nodes, PersistentVolumes, etc) is enforced across namespaces.
* Run the following commands from inside the ram\_coin directory
* namespaces, namespace or ns
* $ kubectl get namespaces
* $ kubectl get ns

$ kubectl get namespace

* Using --namespace or -n let you specify a namespace and look at objects within that namespace.
* Without any specification, it is the default namespace (default)

$ kubectl get ns

$ kubectl get pods **-n** kubernetes-dashboard

$ kubectl get pods

$ kubectl get services **--namespace** kubernetes-dashboard

$ kubectl get services

## Remove pods and services

* Removing pods is equivalent to removing deployment
* Removing pods and services separately

$ kubectl get pods

$ kubectl get deploy

$ kubectl delete deploy redis

$ kubectl get services

$ kubectl delete services redis

$ kubectl get services

$ kubectl get deploy

## Automated Kubernetes Deployment

$ kubectl create namespace ramcoin

$ kubectl create **-f** ramcoin.yaml **--namespace** ramcoin

$ kubectl get pods **-n** ramcoin

$ kubectl create **-f** ramcoin-service.yaml **--namespace** ramcoin

$ kubectl get services **--namespace** ramcoin

## Automated recovery

* Check status and deployment locations of all pods on the head node

$ kubectl get pods **-n** ramcoin **-o** wide

* SSH into worker-1 and reset the Kubelet. Enter y when asked.

$ sudo kubeadm reset

* Run the following commands on head to observe the events
  + After a few minutes, worker-1 becomes NotReady via kubectl get nodes
  + After five minutes, kubectl get pods -n ramcoin -o wide will show that pods on worker-1 being terminated and replications are launched on worker-2 to recover the **desired state** of ramcoins.
  + The five-minute duration can be set by the --pod-eviction-timeout parameter.

$ kubectl get nodes

$ kubectl get pods **-n** ramcoin **-o** wide

## Borg, a cluster management system

* Google’s Cluster Management System
  + First developed in 2003.
* Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale cluster management at Google with Borg.” In Proceedings of the Tenth European Conference on Computer Systems, p. 18. ACM, 2015.
* Manages hundreds of thousands of jobs, from many thousands of different applications, across clusters up to tens of thousands machines.

Diagram

Description automatically generated

## Why Borg and Kubernetes

* Borg is the predecessor of Kubernetes. Understand Borg helps understand the design decision in creating Kubernetes.
* Kubernetes is perhaps the most popular open-source container orchestration system today, for both academic and industry.
* Other container orchestration systems are either
  + Deprecating (Docker Swarm)
  + Integrates container management as part of the existing framework rather than developing a new management system (UC Berkeley’s Mesos and Twitter’s Aurora)
  + We will briefly discuss them at the end of this episode.

## Benefits of Borg

* Hides the details of resource management and failure handling so its users can focus on application development.
* Operates with very high reliability and availability, and supports applications that have similar requirements.
* Runs workloads across tens of thousands of machines efficiently.
* Is not the first system that can do these, but is one of the very few that can do it at such scale.

## User’s perspective

* Work is submitted to Borg as jobs, which can have one or more tasks (binary).
* Each job runs in one Borg cell, consisting of multiple machines that are managed as a single unit.
* Job types:
  + Long running services that should never goes down and have short-lived latency-sensitive requests: Gmail, Google Docs, Web Search …
  + Batch jobs that take a few seconds to a few days to complete.
* Borg cells allow for not just applications, but applications frameworks
  + One master job and one or more worker jobs.
  + The framework can execute parallel applications itself.
  + Examples of frameworks running on top of Borg:
    - MapReduce
    - FlumeJava: Data-Parallel Pipelines
    - Millwheel: Fault-tolerant Stream Processing at Internet Scale
    - Pregel: Large-scale graph processing

## Clusters and cells in Borg

* Machines in cells belong to a single cluster, defined by the high-performance datacenter-scale network fabric connecting them.
  + How is this different that the traditional cluster model?
* A Borg’s alloc defines a reserved set of resources on a machine in which one or more tasks can be run.

## Jobs and tasks

* A job consists of multiple tasks
* Jobs have constraints that allow them to map to machines with satisfactory attributes
* Tasks:
  + **Each task maps to a set of Linux processes**.
  + Authors’ notes: Borg was not designed for virtualization (2003).
  + Also has resource requirements (CPU cores, RAM, disk space, port available …)
* All Borgs’ programs are **statically linke**

## Borg’s architecture

* Borg Master
* Borglet

## Borg Master

* Consists of two process:
  + The main Borgmaster process
  + The scheduler
* Borgmaster:
  + Replicated five times
  + Contains in-memory copy of most of the state of the cell
  + Handles client RPCs that either mutate state (create jobs) or provide read-only access to data.
  + Manages state machines for all the objects in the system (machines, tasks, allocs …)
* Scheduler:
  + Perform feasibility check to map tasks’ constraints to available resources.
  + Picks one of the feasible machines to run the tasks.

## Borglet

* Local Borg agent that is present on every machine in a cell.
* Starts and stops tasks, restarts if failed.
* Manages local resources through OS kernel manipulations
* Reports state of the machine to the Borgmaster.

## Scalability of Borg Master

* Reported in the 2015 paper:
  + Unsure of the ultimate scalability limit (flex anyone?)
  + A single master can
    - manage many thousands machines in a cell
    - several cells have arrival rates of more than 10,000 tasks per minute.
* 2020 Borg analysis report:
  + (Muhamad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: the next generation.” In Proceedings of the fifteenth European conference on computer systems)[https://dl.acm.org/doi/pdf/10.1145/3342195.3387517]
  + 2011 log data: 1 cell, 12000 machines (40 GB compressed)
  + 2020 log data: 8 cells, 96000 machines (350 GB compressed)
* The below graph show fraction of CPU and memory **allocation** of each category of priority queue \*\*relative to cell’s capacity”.

## Isolation

* Sharing machines between tasks help improving utilization
* Security Isolation:
  + Need good security isolation mechanism among multiple tasks on the same machine.
  + chroot to jail processes. SSH-connection is used for communication.
  + VMs are utilized to sandbox external software (Google App Engine and Google Compute Engine). A VM is run as a single task.
* Performance isolation
  + Application’s class: latency-sensitive and batch (batch can be allowed to starved)
  + Resources:
    - Compressible: rate-based and can be reclaimed without killing the tasks (CPU cycles, I/O bandwidth)
    - Incompressible: cannot be reclaimed (memory, disk space)

## Kubernetes: where does it come from

* Developed from lessons learned via Borg
* Become available with the initial release of Docker in March 2013

## Kubernetes: applications versus services

* A service is a process that:
  + is designed to do a small number of things (often just one).
  + has no user interface and is invoked solely via some kind of API.
* An application is a process that:
  + has a user interface (even if it’s just a command line) and
  + often performs lots of different tasks. It can also expose an API,
* It is common for applications to call several service behind the scenes

## Kubernetes: what does it have?

* Kubelet: a special background process responsible for create, destroy, and monitor containers on a host.
* Proxy: a simple network proxy used to separate IP address of the container from the service it provides.
* cAdvisor: collects, aggregates, processes, and exports information about running containers.

Pods

* A collection of containers and volumes that are bundled and scheduled together because they share a common resource (same file system or IP address).
* Docker: Each container gets its own IP
* Kubernetes: Containers of a pod share the same address.
* A pod emulates a logical host (like a VM) to the containe

Important:

* + Kubernetes schedules and orchestrates things at the pod level, not at the container level.
  + Containers running in the same pod have to be managed together (shared fate).
  + Management transparency: You don’t have to micromanage processes within a pod.

## What Kubernetes learned from Borg

* Rejection of the job concept and organize around the concept of pods.
  + labels are used to described the objects (jobs, services, …) and their desired states.
* IP addresses are mapped to pods and services and not physical computers.
* Optimizations for high-demand jobs.
* The perception of Kubernetes’ kernel as an operation system kernel for a distributed system.

## Borg, Oemga, and Kubernetes

* Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. “Borg, Omega, and Kubernetes.” Queue 14, no. 1 (2016): 10.
* Borg:
  + Isolation through the root file system (chroot, cgroups).
  + A modern container is more than just an isolation mechanism: It is also an image, files that make up the applications that runs inside the container.
* Application-oriented infrastructure
  + Containerization transform the data center from being machine-oriented to being application-oriented.
  + Containers encapsulate the application environment, abstracting away many details of machines and OS from the application developer and the deployment infrastructure.
  + Managing containers means managing applications rather than machines.
* Application environment
  + Decoupling of image and OS.
  + **Hermetic** image:
    - What is hermetic?
    - Encapsulation almost all dependencies except Linux kernel system-call interface.
* Containers as the unit of management
  + Relieves application developers and operations teams from worrying about specific details of machines and OS.
  + Provides the infrastructure team flexibility to roll out new hardware and upgrade the OS with minimal impact on running applications and their developers.
  + Ties telemetry collected by the management system to applications rather than to machines.

## ther container management system

* Recalling Hadoop YARN (Yet Another Resource Negotiator)
  + Second generation scheduler for Hadoop (Open-source implementation of Google File System)
  + Deployment of software frameworks as jobs
* Apache Mesos is more similar to YARN and Borg than Kubernetes
  + Cluster management system
  + Containers are executed as jobs.
* Twitter’s Aurora is a scheduler running on top of Mesos.
  + Configurations are more complex, but is still a cluster management system.

**v**

|  |
| --- |
| Data center technology |
|  |

Resource pooling - Storage, processing power, memory, network bandwidth, virtual machines, and other computing resources are pooled together from different locations to server multiple users.

On-demand self-service - Users can automatically and unilaterally provision computing capabilities from the cloud provider

Measured service - Capability of service or resource abstractions are metered.

Broad network access - Computing capabilities from geographically distributed locations are accessible over the network using standard mechanisms

Rapid elasticity - Users can provision computing capabilities rapidly and i

|  |
| --- |
| Data center technology |
|  |

n an elastic manner.

High-speed fiber-optic connections - Broadband networks and Internet architecture

Web applications - Web technology

Modular Linux-based software architecture - Data center technology

Multi-tenant technology - Multiple customers sharing the same application and database instance but have the illusion that they are running on dedicated resources.

Software-defined networking - Virtualization technology

SaaS – Dropbox

Paas – Google app engine

Iaas – amazon elastic compute cloud

Cloud Consumer – security in the cloud

Cloud Provider – security of the cloud

Disadvantages of PaaS from the list below.

|  |
| --- |
| * dependent on individual client's bandwidth |
| * oefptions on technologies and tools are limited |
| * required significant technical efforts |
| * dependent on network and cloud service provider |

SDN Service Node - The task of setting up the network infrastructure for a cloud provision after the cloud provider received instructions from the customer is done by

KVM supports the creation of virtual machines by opening a device node called (/dev/kvm). This node is a special hardware component that helps VMs to communicate with the underlying host physical components. – True

KVM helps the host kernel to allocate a continuous segment of physical memory equivalent to what the guest VM requested. – False

Processes running inside the VM (guest) are executed on the CPU under a special mode called guest mode. – True

Which of the followings is supported in MMU virtualization? - guest\_virtual to guest\_physical