Containerization

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from online materials, primarily from docker.com

Learning objectives

- Understand the purpose of containerization.
- Develop a basic working knowledge of Docker.

Motivation and background

Modern software applications are complex. They can be written in multiple languages, may have many dependencies (e.g., libraries), and may be required to run in a variety of environments.

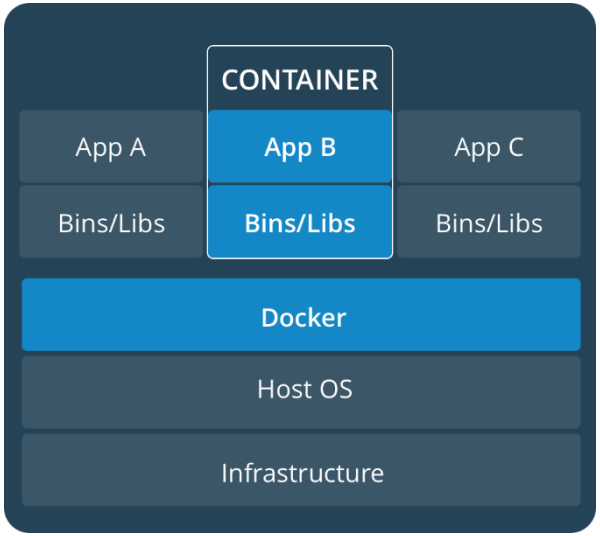
Containers isolate applications from their environment, which simplifies deployment and improves portability. In particular, an application obtains its dependencies from the container in which it is shipped, and not from the host on which it is deployed.

Two of the most popular container frameworks today are Docker and Kubernetes. Docker is a Silicon Valley company founded in 2008, whereas Kubernetes is a project developed at Google and open-sourced in 2014.

Docker and Kubernetes can be used together. Docker can containerize applications, whereas Kubernetes can automate container provisioning and scaling across multiple nodes.

Why not use VMs instead?

Virtual machines can be used to contain an application's environment, but this method is less efficient. VMs virtualize the hardware, whereas containers virtualize the software environment only. As a result, multiple containers can run efficiently as independent processes on the same host OS, but each VM requires a separate guest OS with its own file system.



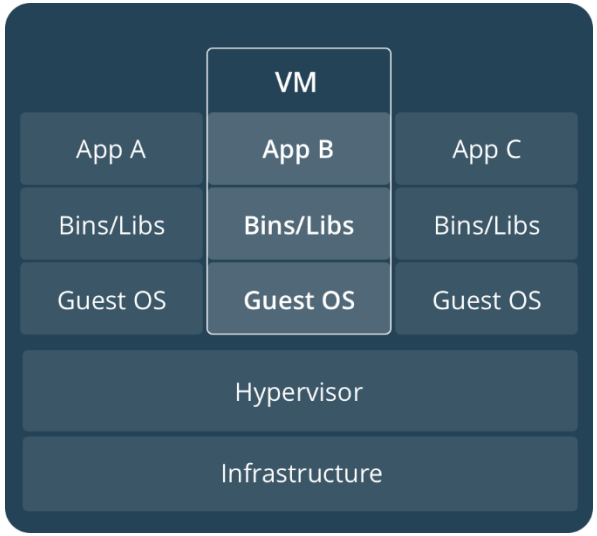


Image source: https://docs.docker.com/get-started/

Getting started with Docker

Setup instructions are provided in the Docker Quickstart document (https://docs.docker.com/get-started/). The main software component is the Docker Engine, which is used to both configure and run containers.

A Docker container executes software packaged in a single file called a Docker container image, which is similar to a VM image but generally much smaller and easier to copy between hosts.

It is possible to run both Linux and Windows applications in Docker containers. Linux is often preferred because it is free software.
On the other hand, the environment inside a Windows Docker container requires a "supplemental license," which is granted by Microsoft when the container runs on a host with a "validly licensed" copy of Windows.

It is possible to containerize a Linux app on Windows using Docker Desktop, and then deploy to a Linux host that runs the Docker engine.

Images and Dockerfiles

Once Docker is installed, the next step is to create a Docker image, which is a file that encapsulates your application and its dependencies. A Docker image is similar to a VM image, but is much smaller because it does not encapsulate an entire guest OS.

Your Docker image is constructed by combining a parent image with a human-readable Dockerfile that specifies how to assemble the private file system of a container.

The parent image is often a base image, obtained from Docker's servers, that provides the abstraction of a particular Linux distribution along with some software platform (e.g., Java, Python, or a database).

The image for your particular container is built from the parent image according to your Dockerfile, and then run as a container.

The application and its container image are often built on one computer, and then deployed to a remote computer, such a server in the cloud.

Example: containerizing your A0

The running example in the remainder of the lecture module uses Linux containers. You can reproduce it on any host that runs the Docker engine or Docker desktop.

When following the example on a Linux host, you need to be either a superuser or a member of the docker group. If needed, you can add yourself to the docker group using the following command:

```
sudo usermod -aG docker $USER
```

Remember to log out and log back in after modifying the docker group.

Step 1: Prepare your application

Create a working directory for this example on your local host.

Copy the CCServer.java file from the A0 starter code bundle to this working directory.

Just for fun, change the name of the Java class to Server, and rename the source code file to Server.java.

Step 2: Create the Dockerfile

Place the following lines in a file called Dockerfile:

```
# Set parent image.
FROM openjdk:8-alpine
# Set working directory.
WORKDIR /usr/src/app
# Copy Java file from the host to container's working directory.
COPY Server.java .
# Build the Java code.
RUN javac Server.java
# Port number the container is listening on at runtime.
EXPOSE 8080
# Command to run the server process on port 8080.
CMD [ "java", "Server", "8080" ]
```

Step 2: Create the Dockerfile

Explanation:

The chosen parent image is openjdk:8-alpine, which combines Alpine Linux with OpenJDK 8.

Alpine Linux is a “security-oriented, lightweight Linux distribution based on musl libc and busybox” (see https://alpinelinux.org/). Its small size makes it a great candidate for Docker images.

Step 3: Build the image

Run the following commands:

```
docker build --tag a0_server:1.0 .
docker image ls
```

Presuming success, the output of the second command should include the newly created a0_server image:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
a0_server	1.0	3c1aee5b7014	37 minutes ago	105MB
ubuntu	latest	1e4467b07108	12 days ago	73.9MB
hello-world	latest	bf756fb1ae65	7 months ago	13.3kB
openjdk	8-alpine	a3562aa0b991	15 months ago	105MB

Step 3: Build the image

Explanation:

The build command creates an image according to the Dockerfile in the current directory.

Docker automatically downloads the parent image, if needed, to the local host.

Your Server.java file is copied into the image, and the javac command (of the image) is executed to compile the Java code. The image then stores both the Java file and the class file.

The image is configured to expose port 8080 to the host that runs it.

The Docker image is only 105MB, much smaller than a typical VM image!

Step 4: Save the image locally

Run the following command:

```
docker save -o a0_server_image.tar  a0_server:1.0
```

Presuming success, the command will save your image in the current directory in tar format. Now you can copy the image to another host, for example a server in the cloud.

Step 5: Load the image remotely

Run the following command on the server to which you copied the Docker image:

```
docker load  < a0_server_image.tar
docker image ls
```

Presuming success, the second command should indicate that the a0_server image has been loaded into the Docker engine.

Step 6: Run the container

Run the following command on the server to which you copied the image:

```
docker run --detach --publish 8080:8080  --name A0 a0_server:1.0
docker container ls
```

Presuming success, the first command will output a container ID. The second command will confirm that the container is running:

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
09b8776ecb8c        a0_server:1.0       "java Server 8080"  21 seconds ago      Up 19 seconds       0.0.0.0:8080->8080/tcp   A0
```

Step 6: Run the container

Explanation:

Your containerized application is now running on the remote server!

The container is detached from the terminal, and runs in the background.

The process name is docker-proxy, and it runs as the root user.

The Docker engine makes port 8080 of the container available to the host OS (and outside world if there is no firewall) as port 8080. It is possible to remap ports when you run the container.

Since your application runs as root inside the container by default, it can bind to privileged ports (below 1024).

Step 7: Connect the client

Build the A0 client program on the remote host, where your container is running. Run the client program and point it to localhost:8080.

The request sent by your A0 client will be forwarded by Docker to the Java server process inside the container. The response will be forwarded by Docker back to the A0 client process.

Step 8: Cleanup

Run the following commands on the server where you ran the container:

```
docker stop 09b8776ecb8c
docker container ls
```

Presuming success, the first command will repeat the container ID provided on the command line, and the second one will show that the container is no longer running.

Step 8: Cleanup

Explanation:

Your containerized application has been stopped. The container has been removed. Any data saved by your application in the container's file system is lost, in contrast to what happens when a VM is stopped.

It is possible to pause containers temporarily, which preserves the container's file system.

If the application needs to persist data, it can connect to a database on the host or in another container. It can also mount a portion of the host's file system. Be careful, however, because the application inside the container runs as root, and can read anything that is mounted!

It is possible to modify the Dockerfile and configure an unprivileged user account to run the application. This is considered more secure.

The End