Introduction to containers

Nabil Abdennadher
nabil.abdennadher@hesge.ch
• Introduction
• Details: chroot, control groups, namespaces
• My first container
• Deploying a distributed application using containers
• How to get software to run reliably when moved from one computing environment to another:
  • From a developer's laptop to a test environment,
  • From a physical machine to a virtual machine in a cloud.
Containers are the solution

- A container consists of an entire runtime environment: an application, all its dependencies, libraries, binaries, configuration files … bundled into one package.
- Containers encapsulate components of application logic. These components are provisioned only with the minimal resources needed to do their job.
- Unlike virtual machines (VM), containers have no need for embedded operating systems (OS); calls are made for OS resources via an application programming interface (API).
Virtualization vs. “containerization”

traditional virtualization

containers

app
lib
os

app
lib
os

hypervisor
OS

x86 hardware

app
lib
os

app
lib
os

OS

x86 hardware
Virtualization vs. “containerization”

- A virtual machine includes an entire operating system + application.
- A physical server running three virtual machines would have:
  - a hypervisor,
  - three separate operating systems running on top of it.
- By contrast a server running three containerized applications runs a single operating system
- Each container shares the operating system kernel with the other containers.
- That means the containers are much more lightweight and use far fewer resources than virtual machines.
Virtualization vs. “containerization”

- A container may be only **tens of megabytes** in size.
- A virtual machine with its own entire operating system may be **several gigabytes** in size.
- A single server can host far more containers than virtual machines.
- Virtual machines may take **several minutes** to boot up their operating systems and begin running the applications they host.
- Containerized applications can be started **almost instantly**.
What about security?

- Container is a method that isolates processes and resources … But ...

- Containers are not as secure as virtual machines
  - if there's a vulnerability in the kernel, it could provide a way into the containers that are sharing it
  - That's also true with a hypervisor, but since a hypervisor provides far less functionality than the whole OS, it presents a much smaller attack surface.
Will containers eventually replace virtualization?

- That's unlikely in the short term … for security reasons.
- The management tools that orchestrate large numbers of containers are nowhere near as comprehensive as software like Vmware.
- Virtualization and containers may come to be seen as complementary technologies rather than competing ones.
• Introduction
• Details: namespaces, control groups, chroot
• My first container
• Deploying a distributed application using containers
traditional virtualization

containers

namespace
cgroups
chroot

hypervisor

app
lib
os

app
lib
os

x86 hardware

x86 hardware

app
lib
os

app
lib
os

OS
chroot

- Exists since 1979
  - #include <unistd.h>
  - int chroot(const char *path);
- It’s UNIX system call for changing the root directory of a process and it's children to a new location in the filesystem which is only visible to a given process.
- A chroot is a way to isolate a process and its children from the rest of the system
  - create a directory tree
  - copy all the system files needed for a process to run.
  - Use the chroot() system call to change the root directory to be at the base of this new tree.
- The idea of chroot is to provide an isolated disk space for each process.
Control groups (cgroups)

- Provides a mechanism for easily managing and monitoring system resources, by partitioning them into groups, then assigning applications to those groups.
- Resources are:
  - cpu time,
  - system memory,
  - disk
  - network bandwidth
- Application knows nothing about these limits, this is happening outside of the application.
Control groups (cgroups)

- Once the group is created, applications are added to the group. This can happen on the fly, without system reboots, limits can be adjusted on the fly.

- Applications can consume outside the resource limit. However, if there is resource contention, the resources applications will be limited to the cgroup policy.
Namespace

- A Linux installation:
  - maintains a single process tree.
  - shares a single set of network interfaces and routing table entries.
- With namespaces, you can have different and separate network interfaces and routing tables that operate independent of each other: **network namespace**
- With namespaces, it became possible to have multiple “nested” process trees. A process running within a namespace can only see processes in the same namespace: **process namespace**
Process namespace

https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces
Three system calls are used for namespaces:

- **clone()**: creates a new process and a new namespace; the process is attached to the new namespace.
- **unshare()**: does not create a new process; creates a new namespace and attaches the current process to it.
- **setns()**: join an existing namespace.
Process namespace (summary)

- The process namespace allows one to spin off a new tree, with its own PID 1 process.
- The process that does this, remains in the parent namespace (original tree), but makes the child the root of its own process tree.
- With process namespace isolation, processes in the child namespace have no way of knowing of the parent process’s existence.
- Processes in the parent namespace have a complete view of processes in the child namespace.
Network namespace
Network namespace

- Creating your namespace: `ip netns add mynamespace`
- Checking: `ip netns list`
- Creating virtual Ethernet interfaces:
  - `ip link add veth0 type veth peer name veth1`
  - Checking: `ip link list`
- Connect the global namespace to `mynamespace`
  - `ip link set veth1 netns mynamespace`
  - Checking:
    - `ip netns exec mynamespace ip link list`
    - `ip link list`
Network namespace

- Configuring Interfaces in Network Namespaces:
  - `ip netns exec mynamespace ifconfig veth1 10.1.1.1/24 up`

- Finding the list of pids in a specified network namespace
  - `ip netns pids mynamespace`

- Finding the network namespace of a specified pid:
  - `ip netns identify #pid`
• Introduction
• Details: namespaces, control groups, chroot
• **My first container (Docker)**
• Deploying a distributed application using containers
• Docker container is an open source software development platform. Its main benefit is to package applications in containers: Container as a Service (CaaS)
Why Docker?

- Portable deployment across machines
- Docker images can run unchanged on any platform supporting docker
- Automatic build: Create containers from build files
- Component re-use: Any container can be used as a base, specialized and saved
- Sharing: Support for public/private repositories of containers
- Tools: CLI / REST API for interacting with docker
Creating your first container with Docker

- `sudo apt-get update`
- `sudo apt-get install docker.io` # docker installation
- `sudo addgroup ubuntu docker` # add Ubuntu user to the docker group in order to avoid sudo
- `reboot`
Creating your first container with Docker: server.py

```python
#!/bin/python
from flask import Flask
app = Flask(__name__)
@app.route("/")
def helloworld():
    return "Hello world !\n"
app.run(host="0.0.0.0",debug=False)
```
Creating your first container with Docker

- Create a folder (example: dockertest)
- In dockertest folder, create Dockefile file

- FROM ubuntu:latest
- RUN apt-get update && apt-get install -y python-pip
- RUN pip install flask
- COPY "server.py" "/tmp/server.py"
- EXPOSE 5000
- CMD ["python","/tmp/server.py"]
Creating your first container with Docker

- `docker build -t IMAGE_NAME` .
- `docker images`  
- `docker run -p 10000:5000 -d IMAGE_NAME`  
- `docker ps`  
- `docker inspect ID`  
- `docker stop ID`  
- `docker rm ID`
RUN, CMD, ENTRYPOINT

- When Docker runs a container, it runs an *image* inside it. This image is usually built by executing Docker instructions, which add *layers* on top of existing image or OS *distribution*. OS *distribution* is the initial image and every added layer creates a new image.
- Each *RUN* adds a new layer
- **RUN** instruction allows you to install your application and packages required for it.
- **RUN** executes any commands on top of the current image and creates a new layer by committing the results.
RUN

RUN    apt-get update && apt-get install -y \\cvs  \\git  \\subversion
• CMD instruction allows you to set a *default* command, which will be executed only when you run container without specifying a command.

• If Docker container runs with a command, the default command will be ignored.

• If *Dockerfile* has more than one CMD instruction, all but last CMD instructions are ignored.
ENTRYPOINT

- Similar to CMD
- Used to transfer parameters (arguments) to the container
• Introduction
• Details: namespaces, control groups, chroot
• My first container (Docker)
• Deploying a distributed application using containers
Docker installation

- `sudo apt-get update`
- `sudo apt-get install docker.io`
- `sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker`
- `sudo groupadd docker`
- `sudo usermod -aG docker $USER`
- `sudo reboot`
restclient’s Dockerfile

- FROM ubuntu:latest
- RUN apt-get update && apt-get install -y python-pip
- RUN pip install pymongo
- RUN pip install requests
- COPY "restclient.py" "/tmp/restclient.py"
- ENTRYPOINT ["python","/tmp/restclient.py"]
restserver’s Dockerfile

- FROM ubuntu:latest
- RUN apt-get update && apt-get install -y python-pip
- RUN pip install flask
- RUN pip install pymongoCOPY
  "restserver.py" "/tmp/restserver.py"
- EXPOSE 18000
- ENTRYPOINT ["python","/tmp/restserver.py"]
Deploying a distributed applications with docker: Creating images

- cd client
- docker build -t restclient .
- cd ../server
- docker build -t restserver .
- docker run --name mongodb -d mongodb