Politecnico di Torino

Introduction to Lightweight Virtualization: Namespaces, cgroups, Docker

Marco Iorio

November 2, 2021

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free:

- **to Share**: to copy, distribute and transmit the work
- **to Remix**: to adapt the work

Under the following conditions:

- **Attribution**: you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial**: you may not use this work for commercial purposes.
- **Share Alike**: if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

More information on the Creative Commons website.

Acknowledgments

The author would like to thank all the people who contributed to this document.

Contents

1 Introduction **5**
1.1 Physical setup . . . 5

2 Namespaces and cgroups **7**
2.1 Introduction . . . 7
2.2 Traditional filesystem isolation: chroot . . . 7
2.3 Linux namespaces . . . 9
2.3.1 Process ID namespaces . . . 9
2.3.2 Mount namespaces . . . 10
2.3.3 Network namespaces . . . 11
2.4 Linux cgroups . . . 15
2.4.1 Running multiple processes without cgroups . . . 16
2.4.2 Running multiple processes with cgroups . . . 16
2.4.3 Cleaning up the lab . . . 17

3 Docker **18**
3.1 Introduction . . . 18
3.1.1 Docker in brief . . . 18
3.1.2 Docker setup . . . 18
3.1.3 Ajusting MTU of containers . . . 19
3.2 Familiarizing with Docker . . . 19
3.2.1 Hello-world: run, show and delete container and images . . . 20
3.2.2 Starting an Alpine Linux . . . 21
3.2.3 Verify the isolation of running containers . . . 21
3.2.4 Running a shell in an already running container . . . 22
3.2.5 Docker networking brief . . . 23
3.3 Running a real service (LAMP stack) . . . 23
3.3.1 Running the web server . . . 23
3.3.2 Populate the apache container with (embedded) HTML pages . . . 24
3.3.3 Populate the apache container with (external) HTML pages . . . 24
3.3.4 Creating a user-defined bridge . . . 25
3.3.5 Running the database server . . . 25
3.3.6 Creating a custom image with Dockerfiles . . . 26
3.4 Using Docker Compose to coordinate multiple containers . . . 29
3.5 Extra Exercises . . . 30
3.5.1 The Problem of Cooperative applications . . . 30
3.5.2 Sharing network namespaces with Docker . . . 31
3.5.3 Building a container runtime from scratch in Go . . . 32
3.5.4 From simple containers to Pod . . . 32
3.5.5 PIDs in Linux . . . 34
3.5.6 The problem of "Zombies" in Containers . . . 34

4 Appendix **35**
4.1 Installing cgroup-tools . . . 35
4.2 Ansible playbook . . . 35

1 Introduction

The goal of this laboratory is to guide the reader through the concepts of *lightweight virtualization* and some of its possible applications.

Over the years, it has been widely recognized the necessity for *strong process isolation* to, e.g., allow the execution of arbitrary code on a third-party server and prevent that a possible intrusion in a service does compromise the entire machine it is running on. Nonetheless, classical full-blown virtualization is often considered undesirable, due to the associated overheads, especially in terms of memory requirements and operating system start-up time.

To solve these issues, over the last two decades, new functionalities have been gradually introduced in the Linux kernel to allow multiple programs to perceive a different and isolated view of a part the resources available in a system, as well as to limit and account their usage. You will experiment with some of these techniques, namely *namespaces* and *cgroups*, in Chapter 2.

Subsequently, leveraging the building block introduced in the Linux kernel, the concept of lightweight virtualization has been introduced. To this end, *Docker* is probably the most known software platform allowing to easily create lightweight, portable and self-contained containers. Under the hood, it exploits the basic primitives provided by the Linux kernel in order to create isolated virtual environments where applications can be executed, along with all their required dependencies. Chapter 3 will provide you a practical introduction to the main functionalities made available by Docker.

Extra: In addition to this text, we provide several videos which discuss some parts of the topics of this Lab. You can find them at

`https://www.youtube.com/playlist?list=PLTAfidx4guQImT5beuAs4YAhIzuBBoEHk`

1.1 Physical setup

This lab requires a physical setup such as depicted in Figure 1.1. Each student will be provided with a couple of VM running in the POLITO datacenter, accessible through the Crownlabs dashboard, which

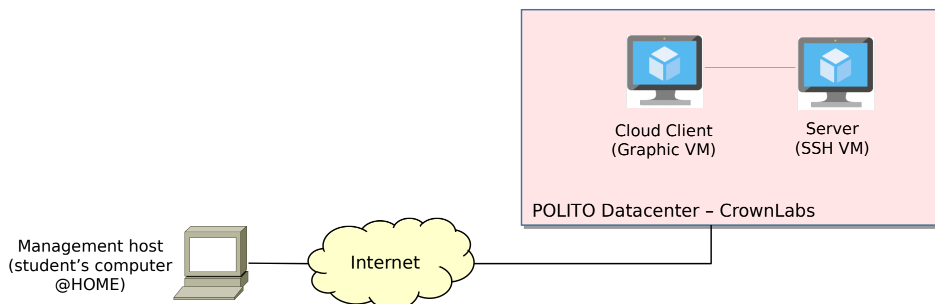

Figure 1.1: Physical setup: how to connect to your virtualized resources.

represents a **client** and a server.

A Linux (or equivalent) client, using the "Cloud Client" template, should be provisioned. This machine represents our **management host**, which is used to configure our (remote) server environment, the "Cloud Computing: Docker" laboratory template.

On the **server** side, this lab operates with a single VM, in which we will complete all our exercises.

2 Namespaces and cgroups

2.1 Introduction

This part of the laboratory aims to provide a practical introduction to the main functionalities provided by the Linux kernel to implement strong process isolation. Although usually not exploited directly by final users nor by general-purpose programmers due to their complexity, *namespaces* and *cgroups* represent the main building blocks upon which lightweight virtualization is constructed. Hence, the main goal of the following is to gain better confidence with the some of the technologies exploited under the hood by *Linux Containers* and *Docker* while performing some simple tasks.

2.2 Traditional filesystem isolation: chroot

This section provides an introduction to one of the filesystem isolation facilities provided by the Linux Kernel: *chroot*. Its purpose is to change the apparent root directory (i.e. `/`) for one process and its children: thus, a program executed in such a modified environment cannot see and access files outside the designated directory tree, effectively creating a sort of *jail*.[1]

Generally speaking, a chroot environment can be exploited to create and host a separate virtualized copy of the software system, where applications can be executed for testing purposes, along with all the expected dependencies, without risking to modify and damage the hosting system. Nonetheless, one of the most known usages of chroot is probably associated with the recovery of unbootable systems. In this situation, the usually adopted solution consists in booting a live distribution and then using chroot to move back to the corrupted environment, to e.g. reinstall the bootloader and perform any recovery task that may be required.

In the following, you will be guided along the execution of a `bash` shell within a chroot environment, to verify the provided level of isolation. To begin, let's create the directory that will be mounted as `/` in the chroot environment:

```
mkdir --parents docker-lab/chroot
```

Then, copy the `bash` executable into the `docker-lab/chroot/bin` directory:

```
mkdir --parents docker-lab/chroot/bin
cp --verbose /bin/bash docker-lab/chroot/bin
```

Before executing the `chroot` command, you need to copy to the chroot system all the libraries required by `bash`, which can be listed by typing `ldd /bin/bash`.

In this respect, please, note that:

[1] Please note that it is fairly easy to escape the jail if you can get `root` privileges within the jail itself. See, for instance, `https://warsang.ovh/prison-break-chroot/` and its references for more information.

- `linux-vdso.so` is a special library that is directly provided by the kernel. Hence, it is not associated with a file name and you do not need to copy it in the chroot environment.
- All the required libraries needs to be copied in the `docker-lab/chroot` folder, preserving the original folder structure (e.g., if a library is under `/lib64` in the root filesystem, it must be present in the same path of the *chrooted* environment).

In case of problems, you can leverage the following command that copies all the required files and preserves the original folder structure:

```
for LIB in $(ldd /bin/bash | grep --only-matching --perl-regexp '/.*(?= \(0x)'); do \
    DEST="docker-lab/chroot$LIB"; \
    mkdir --parent "$(dirname $DEST)"; \
    cp --verbose "$LIB" $DEST; done
```

Finally, observe the content of the `chroot` directory by typing `tree docker-lab/chroot` (or, if you prefer, `ls -l --recursive docker-lab/chroot`).

Now, you are ready to create the chroot environment with the following command:

```
sudo chroot --userspec="$USER" docker-lab/chroot /bin/bash
```

where:

- we enter the chroot environment with the current user as root;
- the chroot environment is created starting from `docker-lab/chroot` folder;
- once we enter, we execute the `bash` shell. Please note also that the `/bin/bash` path is relative to the chroot system, thus the command will invoke `bash` from `docker-lab/chroot/bin` and not from the `/bin` folder of the host system.

Note: in case you experience an error like `chroot: failed to run command '/bin/bash': No such file or directory` it may be that the libraries are not all in the proper place, while the executable may be ok.

If the `chroot` command succeeds, your `bash` prompt should have changed to `bash-x.y#`, with `x.y` being its version number.[2] Now, it is possible to perform some simple experiments from within the chroot jail. However, please notice that only `bash` and its built-in commands can be invoked, while any other command (e.g., `ls`) will fail with a `command not found` error, since the corresponding executable is not present within the jail.

Now, let's explore the root filesystem from the chroot environment, remembering that the `ls` command is not available within the chroot environment, hence it needs to be emulated as follows:

```
# shopt is a shell builtin command to set and unset (remove) various
#   bash shell options. Here we set globstar, which enables the
#   double asterisk, which expands to any number of directories.
shopt -s globstar
for PATH in /**/*; do echo "$PATH"; done
```

As you can see looking at the listed elements, only the files and directories contained in the `chroot` directory can be accessed from within the chroot environment.

Now, let's create a new file in the (chroot) root folder:

[2] Prompt '`bash-x.y#`' is the default one used by `bash` in case no customized directives for the prompt are available in the user's home folder.

```
echo "This is a test file created within the chroot environment" > testch.txt
```

This file would be visible under `/testch.txt` from within the chroot environment, while under `docker-lab/chroot/testch.txt` when opening a new terminal operating outside the chroot environment.

As evident, the chroot user cannot escape from its environment, but the base system has the complete visibility over the entire filesystem, including the chroot folders.

Before moving on to the next section of the laboratory, remember to exit from the chroot environment (i.e. type `exit` to quit bash).

2.3 Linux namespaces

Quoting Wikipedia[3], "*Linux namespaces* are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources". In other words, they enable the creation of distinct virtual environments with respect to a given type of kernel resource, so that different namespaces have a completely independent view of it. For instance, two `network` namespaces are associated with independent networking stacks, virtual interfaces, IP addresses, routing tables and so on.

As of today, the Linux kernel provides eight types of namespaces, which constitute one of the building blocks of lightweight virtualization. In the following, you will be required to perform some simple experiments with three of them:

- `Process ID (PID)`, to create different process trees;
- `Mount (MNT)`, to create a completely new view of the filesystem with the desired structure;
- `Network (NET)`, to allow two processes to perceive a totally different network setup.

2.3.1 Process ID namespaces

As a first example of namespaces, let's experiment with the creation of a `PID` namespace. In Linux, processes originate a single process tree, with each process being associated to a parent and possibly having many children. To this end, the root of the tree is the `init` process, having `PID=1` and being in charge of a set of special tasks (e.g. all orphaned processes are attached to it).

`PID` namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID. In particular, they allow a process to create a new process tree, with its own `PID=1` process, which receives most of the same special treatment as the normal `init` process. `PID` namespaces are nested, meaning that when a new process is created it will have a PID for each namespace from its current namespace up to the initial `PID` namespace. Hence the initial `PID` namespace is able to see all processes.

In the following, you will be requested to run a `bash` shell in a new `PID` namespace, observing the PID assigned to it (and its children) both in the initial and in the new namespaces. To this end, you will leverage the `unshare`[4] program, an utility allowing to run a program with some namespaces unshared from its parent.

In this case, the command we have to run is the following:

[3] `https://en.wikipedia.org/wiki/Linux_namespaces`

[4] A more detailed description of all the available flags is present in the official documentation (`man unshare`).

```
sudo unshare --pid --mount-proc --fork /bin/bash
```

where:

- `--pid`: specifies the creation of a PID namespace.
- `--mount-proc`: mounts a new `proc` filesystem at `/proc` to ensure that the executed process gets `PID=1` (under the hood, it also creates a new `MNT` namespace).
- `--fork`: forces `unshare` to fork before running the command instead of running it directly, again to ensure that the process gets `PID=1`.
- `/bin/bash`: specifies the command to be executed, in this case the `bash` shell.

Note: in case the error `bash: /root/.bashrc: Permission denied` is triggered, you can safely ignore it (to get rid of the error, append the `-norc` flag to the `bash` invocation).

Now, issue the `htop` command in the newly created namespace. As you can see, the `/bin/bash` process got assigned `PID=1` and `htop` is displaying only the processes running in the current namespace.

Then, open a new terminal (thus, back to the initial `PID` namespace) and issue the `ps -af` command: can you identify the `/bin/bash` and `htop` processes running in the newly created namespace? If yes, which are the PIDs associated with them?

Before moving on to the next section of the laboratory, exit from the `PID` namespace.

2.3.2 Mount namespaces

Slightly adapting the introduction provided by *LWN.net*[5], `MNT` namespaces are a flexible tool that allows to isolate the list of mount points seen by the processes in a namespace. In other words, each `MNT` namespace has its own list of mount points, meaning that processes in different namespaces see and are able to manipulate different views of the single directory hierarchy.

When the system boots, there is a single mount namespace, the so-called “initial namespace”. Then, new `MNT` namespaces can be created, with a copy of the mount point list replicated from the namespace of its creator. Finally, mount points can be independently added and removed in each namespace. Changes to the mount point list are by default visible only to processes in the mount namespace where the process resides: the changes are not visible in other mount namespaces.

In the following, you will be required to run a `bash` shell in a new `MNT` namespace. Then, you will modify some of the mount points and verify that they are independent from the ones present in the initial namespace.

To begin, let’s use again the `unshare` utility, leveraging the `--mount` flag to create a new `MNT` namespace.

```
sudo unshare --mount /bin/bash
```

Differently from what happened when playing with `chroot`, the newly created namespace contains the same mount points as the original one. Hence, you can access the very same files and programs available in your original system; in fact, `bash` was started from its original location (`/bin/bash`).

Now, let’s create a new mount point:

[5] https://lwn.net/Articles/689856/

```
sudo mkdir --parents /mnt
sudo mount --bind docker-lab/chroot/ /mnt/
```

Here, the `mount` command remounts (notice the `--bind` flag) the `docker-lab/chroot/` directory and all its subdirectories, i.e. the ones you created in Section 2.2, with target `/mnt`. Hence, if you now type `tree /mnt`, you will see the directory structure you created previously.

Finally, create a new file in the new mount point, named `/mnt/testns.txt`:

```
echo "This is a test file created within the new mount namespace" > /mnt/testns.txt
```

You can verify that it is reflected back to the `docker-lab/chroot/` folder.

To verify the isolation between different `MNT` namespaces, let's open a new terminal (thus, back to the original `MNT` namespace) and follow the steps below:

- Show the content of the `/mnt` directory. Is there any file?
- Type `tree docker-lab/chroot/`. Can you see the `testns.txt` you created previously? Why?
- List, in both terminals, all the mounted filesystems by typing `findmnt`. Is there any difference between the two outputs? Tip: try to identify the mount point you created previously (having target `/mnt`).

Concluding, in a typical scenario, `MNT` namespaces would be used in combination with `chroot` to "customize" the filesystem view seen by a process. Simplifying a bit the different steps, a new `MNT` namespaces would be initially created and entered to avoid cluttering the original one with additional mount points. Then, one would perform all the `mount` and `umount` operations required to setup the environment, before finally executing the `chroot` command.

Before moving on to the next section of the laboratory, exit from the `MNT` namespace.

2.3.3 Network namespaces

A `NET` namespace allows two processes to perceive a completely different and independent network setup. In other words, each `NET` namespace is characterized by its own set of IP addresses, its own routing table, firewall rules, and so on. Even the loopback interface is different. Whenever a new network namespace is created, it contains only a loopback interface. Afterwords, it is possible to attach either physical or virtual interfaces to the newly created namespace (each network interface is present in exactly one namespace).

In the following you will create two `NET` namespaces, their interconnection and configuration to permit the communication between each other, achieving the logical topology shown in Figure 2.1.

To begin, let's create our first namespace named `ns1`:

```
# Create a new network namespace called 'ns1'
sudo ip netns add ns1
```

The **list** of available `NET` namespaces is visible by typing `ip netns list` (or the shortest version `ip netns`). A **command** (e.g., running an application) can be issued in the newly created namespace with `ip netns exec` <namespace_name> <command>.

For instance, let's show the network interfaces available within `ns1`:

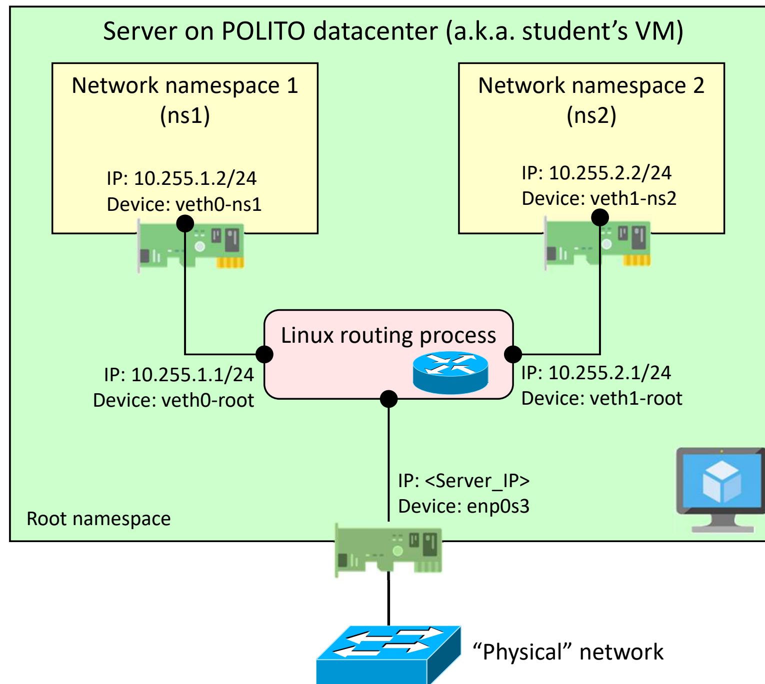

Figure 2.1: Playing with network namespaces: logical setup.

```
sudo ip netns exec ns1 ip link
```

Currently, you should only see the `lo` (loopback) interface, with associated state `DOWN`.

Let's enable it and verify it works with a ping to `localhost` (of the namespace):

```
# Move the state of device loopback ('dev lo') to up
sudo ip netns exec ns1 ip link set dev lo up
sudo ip netns exec ns1 ping 127.0.0.1
```

Note: in case the loopback device is *down*, you cannot ping neither the loopback address nor any additional address you may have in the namespace (e.g., address `10.255.1.2` that will be configured in the following).

Now, we need to connect the `ns1` namespace to the "root namespace". To this end, we need to create a Virtual Ethernet Device (`veth`), a network abstraction that can be seen as a wire with two ends, attached to different namespaces. In other words, a `veth` acts as a tunnel, allowing the traffic to cross the namespace border and to be delivered to another namespace.

To proceed, let's create the new `veth` and push it properly in the desired namespace:

```
# Create the first virtual Ethernet link (i.e., 'veth' pair); the two
#   ends are virtual interfaces called 'veth0-root' and 'veth0-ns1'
# Right now, both ends are in the 'root' namespace
sudo ip link add veth0-root type veth peer name veth0-ns1

# Attach one side of the veth pair ('veth0-ns1') to the namespace 'ns1'
sudo ip link set veth0-ns1 netns ns1
```

The `ip link add` command creates a new link of type `veth`, with one end (`veth0-root`) attached to the "root namespace" and the other (`veth0-ns1`) to the `ns1` namespace.

Alternatively, this is a shorter command that does everything in just one step:

```
sudo ip link add veth0-root type veth peer name veth0-ns1 netns ns1
```

Once the veth is in place, we can assign an IP address to both ends and verify that they can ping each other. Let's start with the namespace:

```
# veth0-ns1 configuration (ns1 namespace)
# Set the interface 'up'
sudo ip netns exec ns1 ip link set veth0-ns1 up

# Assign an IP address to device veth0-ns1
sudo ip netns exec ns1 ip addr add 10.255.1.2/24 dev veth0-ns1
```

Now, you can see that the new address is reachable if *pinged* from the ns1 namespace, while it is unknown if it is *pinged* from the root namespace:

```
# Ping from the root namespace: we expect no answers
ping 10.255.1.2

# Ping from the ns1 namespace: we expect answers to our requests
sudo ip netns exec ns1 ping 10.255.1.2
```

Let's complete the configuration also of the other end of the veth, and let's check that the connectivity is ok:

```
# veth0-root configuration (root namespace)
sudo ip link set veth0-root up
sudo ip addr add 10.255.1.1/24 dev veth0-root

# Connectivity check: pinging remote endpoint
#   From root, contacting the ns1 namespace:
ping 10.255.1.2

#   From the ns1 namespace, contacting the root:
sudo ip netns exec ns1 ping 10.255.1.1
```

In case both ping commands succeeded, you have successfully created an IP interconnection between ns1 and the "root namespace".

Before moving on, please note the differences between the commands that need to be issued to configure the veth0-root and veth0-ns1 interfaces. The former interface, in fact, is attached to the *root* namespace and can be configured using the vanilla ip command. The latter, instead, is attached to the ns1 namespace, thus requiring to prepend ip netns exec ns1 to each command to specify the namespace.

Adding a second namespace and the ip forwarding between namespaces

Now, repeat the previous steps to reproduce the second part of the topology depicted in Figure 2.1. In other words, you are required to create the second namespace (ns2), a new veth pair and configure the IP addresses.

In addition, you need to enable the IP forwarding in the root namespace, which transforms a host into a router (i.e., packets received from one interface can be forwarded to another interface, assuming that the routing table is correct):

```
sudo sysctl net.ipv4.ip_forward=1
```

Moreover, on some systems you have to change the default firewall policy on `iptables` for the `FORWARD` chain (i.e., for packets that are being forwarded) from `DROP` to `ACCEPT`:

```
sudo iptables -P FORWARD ACCEPT
```

However, when trying to ping one namespace from the other, we discover that they are not reachable. In fact, the following command fails:

```
sudo ip netns exec ns1 ping 10.255.2.2
```

Suggestion: the problem is related to the routing table of the namespaces, which do not have any specific route for the remote destination. We ask the student to play with the `ip route` command, which has the following syntax[6]:

```
sudo ip route add {NETWORK/MASK} via {GATEWAYIP}
# (example)
sudo ip route add 10.10.10.0/24 via 10.10.10.254
```

and verify that the network topology now works as expected.

Note: in case of problems, use the following commands:

```
sudo ip netns exec ns1 ip route add 10.255.2.0/24 via 10.255.1.1
sudo ip netns exec ns2 ip route add 10.255.1.0/24 via 10.255.2.1
```

Now, the `ping` should work as expected:

```
sudo ip netns exec ns1 ping 10.255.2.2
# PING 10.255.2.2 (10.255.2.2) 56(84) bytes of data.
# 64 bytes from 10.255.2.2: icmp_seq=1 ttl=63 time=0.055 ms
# 64 bytes from 10.255.2.2: icmp_seq=2 ttl=63 time=0.043 ms
```

Full configuration for forwarding

In case you are not able to make the system working using the previous commands, as a last resort you can use this script snippet that sets everything at once:

```
sudo ip netns add ns1
sudo ip netns add ns2

sudo ip netns exec ns1 ip link set dev lo up
sudo ip netns exec ns2 ip link set dev lo up
sudo ip link add veth0-root type veth peer name veth0-ns1 netns ns1
sudo ip link add veth1-root type veth peer name veth1-ns2 netns ns2

sudo ip netns exec ns1 ip link set veth0-ns1 up
sudo ip netns exec ns2 ip link set veth1-ns2 up
sudo ip netns exec ns1 ip addr add 10.255.1.2/24 dev veth0-ns1
sudo ip netns exec ns2 ip addr add 10.255.2.2/24 dev veth1-ns2

sudo ip link set veth0-root up
```

[6] Remeber to prepend `ip netns exec <namespace>` whenever you want to execute a command in a namespace.

```
sudo ip link set veth1-root up
sudo ip addr add 10.255.1.1/24 dev veth0-root
sudo ip addr add 10.255.2.1/24 dev veth1-root

sudo sysctl net.ipv4.ip_forward=1
sudo iptables -P FORWARD ACCEPT

sudo ip netns exec ns1 ip route add 10.255.2.0/24 via 10.255.1.1
sudo ip netns exec ns2 ip route add 10.255.1.0/24 via 10.255.2.1
```

This code snipped is attached to the PDF file as "`snippets/2B-Two-ns-with-routing.sh`"

Now you can ping the other namespace and see that it works.

Clearing the lab

Before moving to the next section of the laboratory, let's clean up the environment:

```
sudo ip netns delete ns1
sudo ip netns delete ns2
```

The above commands will automatically delete also the `veth` interfaces present in the root namespace, hence the corresponding IP routes.

2.4 Linux cgroups

Linux cgroups constitute the second building block necessary to implement lightweight virtualization. Essentially, cgroups are a Linux kernel feature providing a fine-grained control to limit, account, isolate or deny resource usage to a process or to a group of processes.

In the following, you will use cgroups to limit the amount of CPU a given process can use. To this extent, you will need to use the tools provided by the `cgroup-tools` package.[7] In case of problems, refer to the instructions provided in Section 4.1.

To begin, let's create a `bash` script simulating a CPU intensive task by means of an infinite loop:

```
# Create a specific folder for cgroups experiments
mkdir --parents docker-lab/cgroups

# Create a file that consumes endlessly your CPU
cat <<'EOF' > docker-lab/cgroups/infinite.sh
#!/bin/bash
i=0
while true; do
    i=$((i+1));
done
EOF

# Change the permissions of the above file and make it executable
chmod +x docker-lab/cgroups/infinite.sh
```

This code snipped is attached to the PDF file as "`snippets/2A-CgroupsInit.sh`"

[7] Linux includes cgroups in any recent kernel; however, the userspace tool required to control cgroups is not installed by default.

2.4.1 Running multiple processes without cgroups

To check the default behavior when cgroups are not configured, let's execute the `infinite.sh` script three times, passing a different parameter to each one to make them distinguishable for monitoring purposes. Within this command, the `taskset` utility is used to force the three processes to be executed on the very same core (by setting the same CPU affinity). Otherwise, different results would be obtained depending on the number of available CPU cores.

```
taskset 1 docker-lab/cgroups/infinite.sh N1 &
taskset 1 docker-lab/cgroups/infinite.sh N2 &
taskset 1 docker-lab/cgroups/infinite.sh N3 &
```

To start monitoring the CPU usage of the processes use `htop`, or the following command if you want to show a more compact view:

```
htop --pid=$(pgrep --full --delimiter=, "infinite.sh")
```

As you may expect, the three `infinite.sh` processes quickly saturate the first CPU core, with an equal usage distribution (i.e. 33 % each). Before going on, let's stop the running processes:

```
pkill --full "infinite.sh"
```

2.4.2 Running multiple processes with cgroups

Now, we exploit cgroups to balance the amount of CPU used by each process. Let's start by creating two new ad-hoc `cpu` cgroups, respectively named `throttle25` and `throttle75`:

```
sudo cgcreate -a "$USER" -t "$USER" -g cpu:/throttle25
sudo cgcreate -a "$USER" -t "$USER" -g cpu:/throttle75
```

Then, configure a different CPU usage limit for each of them:

```
cgset -r cpu.shares=512 throttle25
cgset -r cpu.shares=1536 throttle75
```

In this case, processes associated to `throttle25` should take 25% of the CPU, while the other group `throttle75` should tale 75%, which is given by the relative share among the two groups (512 for `throttle25`, 1536 for `throttle75`). In this case, the default value for `cpu.shares`, which is equal to 1024 shares, has no effect, as we set explicitly the shares for each group.

Remember that setting a value of `cpu.shares` for a given cgroup does not mean a process running in that cgroup will always be throttled by the kernel to the corresponding amount of CPU usage. Indeed, in case no other processes are currently running, the process will be allowed to use as much CPU as it needs. The specified value of `cpu.shares`, in fact, is taken into account by the scheduler only when there is a contention on the CPU, hence we need to balance the CPU usage among multiple competing processes running on the same core.

To test the cgroups you just created, let's execute again three times the `infinite.sh` script. However, the command is more complex as we use the `cgexec` tool to execute a command in a given cgroup, specified by the `-g` flag.

```
cgexec -g cpu:throttle25 taskset 1 docker-lab/cgroups/infinite.sh N1 &
cgexec -g cpu:throttle25 taskset 1 docker-lab/cgroups/infinite.sh N2 &
```

```
cgexec -g cpu:throttle75 taskset 1 docker-lab/cgroups/infinite.sh N3 &
```

Now, let's start again `htop` to observe the CPU usage of the three processes:

```
htop --pid=$(pgrep --full --delimiter=, "infinite.sh")
```

Differently from the previous case, now it is possible to observe an uneven CPU usage distribution among the three processes. In particular, the process executed in the cgroup named `throttle75` is using 75% of the first CPU core, while the 25% of CPU usage is shared by the other two processes, both attached to the cgroup named `throttle25`.

2.4.3 Cleaning up the lab

Before moving to next part of the laboratory, let's stop the processes and clean up the environment:

```
pkill --full "infinite.sh"

sudo cgdelete -g cpu:/throttle25
sudo cgdelete -g cpu:/throttle75
```

3 Docker

3.1 Introduction

This part provides a practical introduction to Docker,[1] a software platform allowing to easily create lightweight, portable and self-contained containers. The underlying scenario taken as a reference during this laboratory encompasses a multi-container application that simulates a typical real-world deployment. It is composed by two main components: a Web server (`apache`), with the `php` interpreter, and a database server (`mariadb`).

3.1.1 Docker in brief

Differently from Full Virtualization, containerization leverages an OS-level virtualization to isolate and limit the execution scope of a certain application (i.e. in Linux, cgroups and namespaces). Instead of having visibility of the whole resources available on the host, once run inside a container, an application can only see what it is allocated to the container itself. Firstly released in 2013, Docker Engine is an open source containerization technology for building and containerizing your applications. It has rapidly become the most popular engine to build and run containers.

Docker Engine provides:

- A container image format and a method for building container images (Dockerfile/docker build). A Docker container image is a standalone package that include everything needed (e.g. code, runtime, system tools, system libraries) to run an application.
- A CLI to manage container images (docker images, docker rm , etc.), run and manage container instances (docker ps, docker rm , etc.). The CLI represents the client for a system-level Docker daemon, with a very similar architecture to libvirt.
- A way to share container images (docker push/pull) using remote repositories (i.e. registries). Those registries are by default pointing to Docker Hub (`hub.docker.com`), but can also self-hosted (e.g. `port.us.org`) or hosted by third-parties (e.g. `quay.io`)

3.1.2 Docker setup

We assume that you already have the Docker Engine installed on you system; type `docker --version` to verify that everything is configured properly. Otherwise, please follow the installation instructions available at `https://docs.docker.com/install/`.

Since Section 3.4 will also require Docker compose, verify that also this tool is correctly installed on your machine by typing `docker-compose -v`. In case something goes wrong, please follow the installation instructions available at `https://docs.docker.com/compose/install/`.

[1] `https://www.docker.com/`

Warning: by default, the docker command needs to be executed as root, otherwise a permission error is raised. If you do not want to preface every command with sudo, you can add your own username to the docker group, as explained in the official guide.[2]

3.1.3 Ajusting MTU of containers

When dealing with virtualized environments, it might happen that the host MTU is smaller than 1500. In this case, it is necessary to explicitly configure the MTU for the docker interfaces.

First, let display the local network cards and their MTU through ip link:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    group default qlen 1000
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq_codel state UP mode
    DEFAULT group default qlen 1000
     link/ether aa:bb:cc:dd:ee:ff brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    mode DEFAULT group default
     link/ether uu:vv:ww:xx:yy:zz brd ff:ff:ff:ff:ff:ff
```

If the outgoing interface (in this case ens3) has an MTU smaller than 1500, we need to configure the Docker daemon in such a way that the virtual network card of newly created containers gets an MTU that is smaller than or equal to that value. For this purpose, we can create the file /etc/docker/daemon.json with the following content:

```
{
  "mtu": 1400
}
```

In this example, we chose 1400 as the value, as this corresponds to the value of the outgoing network card (ens3). After restarting the Docker daemon, the MTU of new containers should be adapted accordingly. However, docker-compose creates a new (bridged) network for every docker-compose environment by default.

The docker daemon may be restarted with:

```
sudo systemctl restart docker
```

3.2 Familiarizing with Docker

This section guides the reader along an initial hands-on tour to explore the world of Docker containers. Obviously, only a very limited subset of the available Docker commands and flags are going to be presented hereafter. In case of doubts, remember you can always issue docker --help and docker COMMAND --help to obtain more information.

Part of the following is inspired by the first "Play with Docker" laboratory; additional hands-on tutorials can be found at https://training.play-with-docker.com/.

[2] https://docs.docker.com/install/linux/linux-postinstall/

3.2.1 Hello-world: run, show and delete container and images

First, let's recap the difference between *images* and *containers*. A Docker image can be seen as an archive, comprised of multiple layers, which groups together all the files (e.g. executables, libraries, tools and configuration files) needed to execute code in a Docker container. Docker images can be either pulled from online repositories or created by means of Dockerfiles, as described in Section 3.3.6. Multiple independent containers can be created starting from a single image, each one characterized by its own status. As we are going to experiment soon, modifications performed in a container are not automatically reflected back to the corresponding image, thus guaranteeing isolation.

Now, let's start with an "hello world" container. Type:

```
docker container run hello-world
```

and observe the output produced, which briefly describes the steps performed by Docker to run the container.

You have probably noticed how fast was Docker to create, start and tear-down the container: execution speed is one of the main distinctive characteristics of lightweight containers compared to virtual machines. While VMs are *hardware* abstractions, required to boot full-blown operating systems from scratch, containers are just *application* abstractions, running on top of an already started operating systems pretty much like usual applications.

Now, let's play with some common commands:

- docker image ls: shows a list of all the images locally available on your system. Right now, you should see only the hello-world image just downloaded to execute the previous container.
- docker container ls: shows you all the containers that are *currently* running. Since no containers are running, you should see a blank line.
- docker container ls -a: the -a switch appended to the previous command changes the output to present the list of *all* the containers you ran, including hello-world, associated with the Exited status. As evident, containers are not automatically destroyed upon termination.

  Note: If you do not assign a name to the container when starting the container with the --name option, then the Docker daemon generates a random string name for you, visible under the column NAMES of the previous output. In fact, the *random* name appears to be the concatenation of an English adjective and a name (e.g., nostalgic_mccarthy) for easier memorization.

  Exercise: take note of the name assigned to your container.
- docker container rm <container ID>: removes the selected container, as they need to be manually deleted upon termination.

  Note: containers must be removed by specifying either their *containerID* or their *container name* (e.g., nostalgic_mccarthy seen before) and not their *image name* (e.g., hello-world in this case), as there may be multiple (running) containers started from the same image.

Upon executing the previous command, the container is no longer present (docker container ls -a returns nothing), while the image is still on the machine (docker image ls returns hello-world).

To delete also the image, you can use either one of the following commands:

```
docker rmi hello-world
# (or)
docker image rm hello-world
```

3.2.2 Starting an Alpine Linux

Alpine[3] is a lightweight Linux distribution meant to be smaller and more resource efficient than traditional ones. For instance, the `alpine` image occupies less than 8 MB, compared to the 100 MB required by `ubuntu` or `debian` images. Indeed, this characteristic made it a popular starting point for many other images. However, Alpine may not include the complete set of commands available in other distros.

To get started, let's fetch the `alpine` image (version `3.10`) from the default Docker registry (i.e., Docker Hub):

```
docker image pull alpine:3.10
```

Once the download completed, you can run a new `alpine` container with:

```
docker container run alpine:3.10 echo "hello from alpine"
```

Behind the scenes, Docker created a new container based on the `alpine` image, ran the specified command (`echo "hello from alpine"` in this case) and, once terminated, automatically stopped the container.

In this case, the `run` command did not download the image, as it is already available locally thanks to the `pull` command we issued before.

3.2.3 Verify the isolation of running containers

Now, you can verify the isolation between different containers in terms of filesystem. Let's run a shell within a new container instance, create a new file called `test-file`, and verify that it is present:

```
# Start the alpine container and run the shell in it
docker container run -it alpine:3.10 /bin/sh

# Now we're inside the container: let's type some very simple commands in the shell
touch test-file
# Shows that the above file is really in the container filesystem
ls -l
# Exit from shell, hence terminate the container
exit
```

The `-it` flag is very important here, as it enables the container to run in an *interactive* terminal, making it possible to run shell commands as if you were inside the container itself. Vice versa, without the flag, the container would have started the shell, which would terminate immediately, since we did not supply any additional commands to be executed by the shell.

Note: the `/bin/sh` command is executed within the file system of the container itself. Hence, the `/bin/sh` executable must be present in the container.

Now, let's create a new container instance and list the files in its root directory:

```
docker container run alpine:3.10 ls -l
```

Is `test-file` present? Why?

[3] `https://alpinelinux.org/`

Let now verify the isolation provided by Docker containers in terms of CPU and RAM. To this end, when a new container is started, it is possible to specify its maximum consumption limits, through the `--cpus` and `--memory` flags (under the hood, this configures the appropriate cgroup parameters):

```
# Start the alpine container and run the shell in it
docker container run -it --cpus=1 --memory=$((100*1024*1024)) alpine:3.10 /bin/sh

# Now we're inside the container: let's install some utility packages
apk add htop stress-ng
```

Now, let's run `htop` within the container. Does `htop` show the container or the host resources? Is this behaviour similar or different compared to what happens when dealing with virtual machines?

We then leverage `stress-ng` to simulate a CPU load within the container, configuring it to attempt saturating two CPUs:

```
# Start the stress-ng utility in background and show the resources used
stress-ng --cpu=2 & htop
```

How did the resource usage shown by `htop` change in this case? Are the resource constraints specified at container creation time enforced by the Linux kernel?

As you experienced with this simple example, the resources perceived by a Docker container are typically the same as the ones actually present on the server it is executed on, although their usage might be limited through appropriate quotas. Still, this can create issues to applications which customize their behavior based on the resources availables, e.g., starting a thread for each core or triggering garbage colection according to memory usage, as they might perceive much larger limits compared to their actual quotas.

3.2.4 Running a shell in an already running container

Sometimes, it is necessary to attach a shell to an already running container, e.g. to verify why an application is not behaving as expected. This is the purpose of the `docker exec` command. Let's start by creating a new instance and simulate a deadlock with a very long sleep through:

```
docker container run --name deadlock-app --rm --init alpine:3.10 sleep 1d
```

where:

- `--name`: assigns a name to the container (`deadlock-app`), to make it easier to be identified (Section 3.2.1);
- `--rm`: automatically destroys the container upon termination; for instance, command `docker container ps -a` will not return this container anymore;
- `--init`: runs an `init` inside the container that forwards signals and reaps processes (necessary to be able to `kill` the `sleep` process):
- `sleep 1d`: represents the command executed in the container, i.e., go to sleep for one day.

Note: the above command will keep the terminal busy, giving the impression that nothing is happening.

Now, open a **new terminal** (in fact, the previous one is *busy*) and issue the following command, which attaches an interactive shell to the running container and terminates the sleeping process (and the container itself):

```
# Attach a shell to the still running 'deadlock-app' container
docker exec -it deadlock-app /bin/sh
# Show all running processes (within the container)
ps -a
# Kill the 'sleep' process
killall sleep
```

The container terminates and, thanks to the `--rm` flag, it also clears itself from the list of containers.

3.2.5 Docker networking brief

Finally, let's investigate the network configuration. Open two terminals and create two new Alpine containers with an interactive shell as you did previously in Section 3.2.3.

Inspect the IP addresses assigned to the containers (command `ip addr`) and answer to the following questions, using some additional commands (e.g., `ping` and `wget`) if needed:

- Can the two containers communicate to each other?
- Can a container ping the host machine?
- Is the Internet reachable from the containers?
- From the information you gained, what is the default network configuration in Docker?

Warning: ICMP Echo messages (i.e. pings) may be blocked by network administrators due to security reasons. In particular, the student's VM hosted inside the PoliTO datacenter may *not* be allowed to ping destinations outside the campus network. Hence, when testing the Internet connectivity from the container, please either ping an internal destination (e.g. `www.polito.it`) or leverage `wget` to perform an HTTP request.

3.3 Running a real service (LAMP stack)

This section will create a basic LAMP stack based on Dockerized components. Following the "one service per container" philosophy, you will run two separated containers: one executing the web server (`apache` with the `php` interpreter) and one in charge of the database server (`mariadb`).

Modularity is a very important concept in Docker, since it allows to keep applications separated, improving maintainability and enabling the reuse of the same container for multiple purposes.

3.3.1 Running the web server

To begin, let's start creating a new `apache` container:

```
docker run -d --name dockerlab-apache --rm -p 8080:80 httpd:2.4-alpine
```

where:

- `-d`: starts the container in background.

  Note: without this flag, the `apache` server starts using the console as its standard input/output, preventing the execution of other commands on the same shell.

- `-p`: the default `http` port used in the container (`TCP 80`) is exposed on port `TCP 8080` of the host. This flag specifies which TCP/UDP ports are published to the host, making the application (e.g. `apache`) to appear as it it is running directly on the host.

To verify that the web server is working properly, point your browser to `http://localhost:8080` or, if you are operating in console mode, type `curl http://localhost:8080`: you should be greeted by *"It works!"*.

3.3.2 Populate the apache container with (embedded) HTML pages

You now have a working web server running as a container on your host machine. However, it is pretty useless unless you can specify the content that has to be served. Let's attach a shell to the `dockerlab-apache` container and create a very simple *hello world* page:

```
# Attach a shell to the running apache container
docker exec -it dockerlab-apache /bin/sh
# Create a new main HTML file
echo '<html><body><h1>Hello world!</h1></body></html>' > htdocs/index.html
```

Now, if you refresh the web page, you should see the output of the new `index.html` page.

However, the direct modification of data files within a container is almost always a bad idea. In fact, changes are not automatically reflected back to the original image: if you run a new container, all the modifications would be lost.

Even putting the code directly inside a container, by means of Dockerfiles (Section 3.3.6), would require the image to be rebuild every time we make some modifications, thus not being suitable for fast changing development cycles.

3.3.3 Populate the apache container with (external) HTML pages

Let's try a better solution: first, stop the `dockerlab-apache` container:

```
docker container stop dockerlab-apache
```

and prepare the `html` file on your local machine:

```
mkdir --parents docker-lab/apache
echo '<html><body><h1>Hello world!</h1></body></html>' > \
    docker-lab/apache/index.html
```

This time, you will use the `-v` flag to *bind-mount* the directory you just created (i.e. `docker-lab/apache`) to the `/usr/local/apache2/htdocs` folder inside the container.

Quoting the official documentation[4] "Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.". In this case, the volume will be mounted as read-only (`ro`), since the files do not need to be modified from inside the container itself.

Restart the `dockerlab-apache` container and use the above mentioned `-v` flag to properly mount the external volume within the container:

```
docker run -d --name dockerlab-apache --rm -p 8080:80 \
    -v ${PWD}/docker-lab/apache:/usr/local/apache2/htdocs:ro httpd:2.4-alpine
```

[4] https://docs.docker.com/storage/volumes/

Verify the correct functioning by refreshing the web page. Now, you are able to update the content served by `apache` by simply modifying the content of your local directory.

Finally, let's stop the `apache` container, since you will be requested to create a new one later on:

```
docker container stop dockerlab-apache
```

3.3.4 Creating a user-defined bridge

Before moving on, let's create a user-defined bridge that will be used to interconnect the web server and the database. While the default bridge can perfectly fulfill this task, user-defined bridges do present some advantages. Among the others, they enable automatic DNS resolution of container names to IP addresses.

To proceed, just type:

```
docker network create dockerlab-network
```

3.3.5 Running the database server

Now, it's time to run the `mariadb` container. Let's begin by creating the directory where the database will be kept persistently on your local system, along with the `sql` scripts used to initialize a test table and populate it with some data:

```
mkdir --parents docker-lab/mariadb/database
mkdir --parents docker-lab/mariadb/scripts

cat <<'EOF' > docker-lab/mariadb/scripts/01-create-tables.sql
CREATE TABLE DOCKER_IMAGES (
    NAME    VARCHAR(100) NOT NULL,
    VERSION VARCHAR(100) NOT NULL,
    PRIMARY KEY (NAME)
);
EOF

cat <<'EOF' > docker-lab/mariadb/scripts/02-insert.sql
INSERT INTO DOCKER_IMAGES(NAME, VERSION) VALUES ("httpd", "2.4-alpine");
INSERT INTO DOCKER_IMAGES(NAME, VERSION) VALUES ("mariadb", "10");
EOF
```

This code snipped is attached to the PDF file as "`snippets/3A-DatabaseInit.sh`"

Then, execute the following Docker command to start the database server:

```
docker run -d --name dockerlab-mariadb --rm --network dockerlab-network \
    -e MYSQL_DATABASE=database -e MYSQL_ROOT_PASSWORD=rootpass \
    -e MYSQL_USER=testuser -e MYSQL_PASSWORD=testpass \
    -v ${PWD}/docker-lab/mariadb/database:/var/lib/mysql \
    -v ${PWD}/docker-lab/mariadb/scripts:/docker-entrypoint-initdb.d:ro \
    mariadb:10
```

where:

- `--network network-name`: defines an alternate bridge where the Docker has to attach to, hence ignoring the default Docker network.

- `-e`: passes a given environment variable to the container. Environment variables represent an easy way to configure parameters associated with a Docker container: in this case, they are used to specify the pieces of information required to initialize the database.

  Note: obviously, the application running inside Docker has to be created in such a way to understand the meaning of the passed environment variables.

  Warning: while not being relevant in this toy example, passwords exported through environment variables may introduce security issues in production and shall be avoided.

- `-v <source>:<dest>`: mounts the specific source folder, located in the host, into a specific mount point (`dest`) in the Docker namespace. In particular, in this case volumes are used to specify both the local folder where the database is kept and the one containing the scripts initializing the content of the database the first time.

Note: both the initialization scripts and the environment variables are leveraged only the first time the database is created: no changes are applied if the database is already present (i.e. if you destroy and recreate the `dockerlab-mariadb` container maintaining the same local copy of the database).

To verify that everything is configured correctly, type the following command to execute a `mysql` shell inside the container and connect it to the database of interest (the credentials are specified using the `--user` and `--password` flags):

```
docker exec -it dockerlab-mariadb \
    mysql --user=testuser --password=testpass --database=database
```

Note: the previous command may fail in case `mariadb` has not yet completed to create and configure the database. The set-up operations are performed in background with no visual feedback, since the container has been started with the `-d` flag. If you occur in this situation, you can fetch the logs from the container:

```
docker logs --follow dockerlab-mariadb
```

and retry once the background tasks terminated.

Once you see the "`MariaDB [database]>`" prompt, it is possible to issue `SQL` statements:

```
SELECT * FROM DOCKER_IMAGES;
quit
```

You should see the content of the newly created table.

3.3.6 Creating a custom image with Dockerfiles

At this point, you have a running `mariadb` server instance. Now, it's time to run the web server and show a page with the content of the `DOCKER_IMAGES` table. Although off-the-shelf `apache-php` images are already available on Docker Hub[5], in the following you will create your own image through a Dockerfile. Quoting the official documentation[6] "Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using `docker build` users can create an automated build that executes several command-line instructions in succession."

Let's start by creating the necessary directories and preparing the Dockerfile:

[5] https://hub.docker.com/_/php#phpversion-apache

[6] https://docs.docker.com/engine/reference/builder/

```
# Create the required folders
mkdir --parents docker-lab/apache-php/docker
mkdir --parents docker-lab/apache-php/html

# Now, let's create the Dockerfile to build our custom container
cat <<'EOF' > docker-lab/apache-php/docker/Dockerfile
FROM alpine:3.10

# Run the command to install the required packages
RUN apk add --update --no-cache apache2 php7-apache2 php7-mysqli

# Allow graceful termination of apache
# https://httpd.apache.org/docs/2.4/stopping.html#gracefulstop
STOPSIGNAL WINCH

# Expose the http port
EXPOSE 80

# Execute apache in foreground as default
CMD ["/usr/sbin/httpd", "-DFOREGROUND"]
EOF
```

This code snipped is attached to the PDF file as "snippets/3B-Dockerfile.sh"

While the complete description of the commands allowed in the Dockerfile is out of scope here, let's briefly analyze the main keywords introduced in the Dockerfile:

- `FROM`: A Dockerfile must start with a `FROM` instruction, specifying the base image from which you are building your own image.
- `RUN`: the `RUN` instruction executes any commands in a new layer on top of the current image and commits the results. The resulting committed image is used for the next step in the Dockerfile.

  Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like in version control systems (e.g., GIT). In this specific case, the `RUN` instruction is used to download and install the `apache` web server along with the `php` interpreter and the `mysqli` extension required to interact with the `mariadb` database. `apk` is package manager in Alpine Linux, used to install, upgrade or delete software.
- `STOPSIGNAL`: sets the Linux signal that will be sent by `docker` to the container to exit (i.e. when issuing `docker stop`). It is used to allow the graceful termination of `apache`.
- `EXPOSE`: informs Docker that the container listens to the specified network ports at runtime. You can specify whether the port listens to TCP or UDP, and the default is TCP if the protocol is not specified.
- `CMD`: specifies the default command for an executing container (e.g. start the `apache` server).

It's time to build your `dockerlab-apache-php` image from the Dockerfile. Let's type:

```
docker build --tag dockerlab-apache-php docker-lab/apache-php/docker
```

and observe the output. You can see the different steps performed to build the image.

Note: when you build a Docker image, you specify a context (i.e. the folder containing the Dockerfile) whose entire content is sent recursively to the daemon. In most cases, it is best to start with an empty directory as context and add only the files needed for building the Dockerfile.

To show the layers composing the newly created `dockerlab-apache-php` image, along with the command used to build each of them, you can leverage the `docker history` command:

```
docker history --no-trunc dockerlab-apache-php
```

To test the newly created image, let's prepare a simple php file reading some information from the database previously generated:

```
cat <<'EOF' > docker-lab/apache-php/html/index.php
<?php
$server = "dockerlab-mariadb";
$username = "testuser";
$password = "testpass";
$database = "database";

$connection = new mysqli($server, $username, $password, $database);
if ($connection->connect_error) {
   die("Connection failed: " . $connection->connect_error);
}

$query = "SELECT * FROM DOCKER_IMAGES";
$result = $connection->query($query);

echo "<center><table border='1' cellpadding='10'>";
echo "<tr><th>Name</th><th>Version</th></tr>";
while ($row = $result->fetch_assoc()) {
    echo "<tr><td>" . $row["NAME"] . "</td>";
    echo "<td>" . $row["VERSION"] ."</td></tr>";
}
echo "</table></center>";
?>
EOF
```

This code snipped is attached to the PDF file as "`snippets/3C-PhpScript.sh`"

Note: the previous php script specifies the target database server by means of its container name (i.e. `$server = "dockerlab-mariadb"`) instead of using its IP address. This is made possible by the connection of the containers to a user-defined bridge (Section 3.3.4). The default bridge, instead, would not resolve the container name to its IP address, due to backward compatibility issues.

Now, run a new container:

```
docker run -d --name dockerlab-apache-php --rm -p 8080:80 \
    -v ${PWD}/docker-lab/apache-php/html:/var/www/localhost/htdocs:ro \
    --network dockerlab-network dockerlab-apache-php
```

Point your browser (or `curl`) to `http://localhost:8080`: you should see the content of the `DOCKER_IMAGES` table.

Once created, Docker images may be shared to the Docker Hub registry to make them available to the community.[7] However, this requires you to either have or create an account on Docker Hub.

Note: given the necessity to have an account on Docker Hub, this part of the laboratory is optional.

To start, just issue the `docker login` command and type your own login credentials. Now, it's possible to share the `dockerlab-apache-php` image:

[7] Alternatively, they can also be pushed to self-hosted registries.

```
docker tag dockerlab-apache-php <your-username>/dockerlab-apache-php:1.0-alpine
docker push <your-username>/dockerlab-apache-php:1.0-alpine
```

where the `tag` command is used to assign a new name to the image before pushing it. Image names must be unique and are specified in the format `<repository>/<image>:<tag>`.

When you previously built the image, you simply assigned `dockerlab-apache-php` as its name with the `--tag` flag. Now, instead, you have to preface the repository name (your own username on Docker Hub) and assign the image a version number (e.g. `1.0-alpine`). Second, the `docker push` command does actually perform the image upload to Docker Hub.

Warning: do not forget to logout from Docker Hub (`docker logout`) if you are no longer using a shared PC.

Cleaning up the environment

Before moving on to the next section, let's stop all containers and remove the user-defined network:

```
docker container stop dockerlab-apache-php
docker container stop dockerlab-mariadb
docker network rm dockerlab-network
```

3.4 Using Docker Compose to coordinate multiple containers

You have now successfully started a basic LAMP stack based on Dockerized components.

However, while experimenting, you might have imagined how cumbersome it soon becomes to create multiple correlated containers, along with all their configuration parameters, using bare `docker run` commands. To this end, different orchestrators have been developed to simplify the management of the containers life-cycle. Among the most famous, it is possible to cite Docker Compose,[8] Docker Swarm[9] and Kubernetes.[10]

In this section, you will experiment with Docker Compose. Quoting the official documentation, "Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a `YAML` file to configure your application's services. Then, with a single command, you create and start all the services from your configuration."

Differently from other orchestrators, Docker Compose manages containers running on a single node. Docker Swarm and Kubernetes, on the other hand, target at clusters encompassing even thousands of nodes and provide many more advanced features.

Now, let's create the `docker-compose.yml` necessary to replicate the LAMP stack created in the previous section.

```
cd docker-lab/

cat <<'EOF' > docker-compose.yml
version: "3"
services:
```

[8] https://docs.docker.com/compose/
[9] https://docs.docker.com/swarm/overview/
[10] https://kubernetes.io/

```
    dockerlab-apache-php:
        build: ./apache-php/docker
        ports:
            - "8080:80"
        volumes:
            - ./apache-php/html:/var/www/localhost/htdocs:ro

    dockerlab-mariadb:
        image: mariadb:10
        volumes:
            - ./mariadb/database:/var/lib/mysql
            - ./mariadb/scripts:/docker-entrypoint-initdb.d:ro
        environment:
            MYSQL_ROOT_PASSWORD: rootpass
            MYSQL_USER: testuser
            MYSQL_PASSWORD: testpass
            MYSQL_DATABASE: database
EOF
```

This code snipped is attached to the PDF file as "snippets/3D-DockerCompose.sh"

Warning: YAML files (e.g. `docker-compose.yml`) strongly rely on indentation for their structure. Hence, do *not* copy the previous snippet directly from the PDF file, since it will totally scramble the indentation. Moreover, before moving on, verify (and fix, if necessary) the file layout using an editor of your choice (e.g. `nano docker-compose.yml`).

As you can see, the file lists all the containers that are requested to be started, associated with their image or the Dockerfile used to build them. Additionally, it specifies the same information previously defined by mean of flags, including volumes, ports and environment variables. As always, a much more detailed description about all the available features can be found in the official documentation.[11]

Once the `docker-compose.yml` file is created, starting and stopping the containers becomes as easy as typing:

```
# Start in background the containers defined by the docker-compose.yml file
docker-compose up -d

# Stop the containers defined by the docker-compose.yml file
docker-compose stop
```

3.5 Extra Exercises

This section provides some further and more advanced exercises that can be useful to enrich your cloud-related skills.

3.5.1 The Problem of Cooperative applications

As we mentioned before in 3.3, Docker container best practices state very clearly: a single process should run inside each container. This is due to several reasons:[12]

[11] https://docs.docker.com/compose/compose-file/

[12] https://devops.stackexchange.com/questions/447/why-it-is-recommended-to-run-only-one-process-in-a-container

- **Separation of Concerns**: Containers without multiple processes do not have a complex internal state. This simplifies a lot container orchestration (e.g; horizontal scaling, dynamic upgrades).
- **Self-Healing**: The Docker engine is good at recognizing crashed containers. However, it is not able to detect the same with multiple processes. For example, the main process could correctly work, but some auxiliary processes could have crashed without notice.
- **Troubleshooting**: Logs from containers can get complex to understand. For example, error messages from a process can be ignored when multiple processes are pushing many logs to the same console.

In fact, this represented an important roadblock for docker containers adoption, since traditional monolithic application can be hardly split in several containers.

Therefore, in several situation, you may have multiple applications that need to be executed in a symbiotic way, where one is actually helping the others working. We can make multiple examples of this scenario:

- **Configuration Reload**: A side-application watches a configuration, recompiles it and triggers the reload of the main application.
- **Metrics Exposition**: A side-application exposes internal metrics of the main application that it is not exposing by itself.
- **Proxy**: A side-application exposes a services offered by the main application with another protocol (e.g.; adding TLS)

As we stated before, containers use a lot of Linux namespaces to change how a process views resources on the system. Therefore, in this scenario, containers have to share some of those namespaces in order to correctly cooperate.

In the next paragraphs, we will see how namespaces can be shared across containers and why this represents a very important feature when dealing with container orchestration.

Part of the following is inspired by: "Sharing Network Namespaces in Docker"[13] and "The Almighty Pause Container"[14].

3.5.2 Sharing network namespaces with Docker

First of all, containers can easily share their network namespace. As a recap, when a process listens on a port, it binds to a specific network interface. This is the reason why many containers of the same application can run on the same host: all those containers are listening to port 80 on their own interface in their own namespace and there is no conflict. When we share the same network namespace in different containers, all of the network interfaces are shared. Therefore, if one container starts listening to a port, no other containers in the same namespace can use that port. As a result, containers can communicate to each other by simply using localhost, without needing any discovery mechanism.

To share the same namespace in Docker, we just have to specify the network flag when starting a container. With this flag, we can either put a container on a network (normal usage) or change it and use the namespace of another container. For example:

```
docker container run -d --name=nginx nginx
docker container run -it --network=container:nginx alpine
```

[13] https://blog.mikesir87.io/2019/03/sharing-network-namespaces-in-docker/
[14] https://www.ianlewis.org/en/almighty-pause-container

When we specify `--network=container:[name|id]`, our new container will share the network namespace of the specified container.

Inside the second container:

```
wget localhost
```

We will obtain the output of the landing page of the first nginx container we executed.

This can be easily scripted by using Docker Compose. In this situation, we have to use `network_mode`, instead of `network`.

One additional feature is the ability to use the service name through `--service:[service-name]`.

```
version: "3.7"
services:
  nginx:
    image: nginx
  alpine:
    image: alpine
    command: sh -c "wget localhost"
    network_mode: service:nginx
```

Pay attention to not to include an accidental space after the colon.

3.5.3 Building a container runtime from scratch in Go

In this tutorial[15], Liz Rice shows how it is possible to create a very simple container engine in Golang. This unveils which is the mechanism behind the docker engine by simply recreating it in a minimal way. This tutorial is strongly advised if you want to understand the internals of the interaction between container engines and Linux kernel. No preliminary Golang skills are required as she comments the code she writes and the tests she makes.

3.5.4 From simple containers to Pod

This further exercise objective is to explicit which is the path from simple Docker containers to Kubernetes Pods. Kubernetes pods, as we will see in next laboratory, are containers configured via a dedicated specification (i.e.; the pod Spec). The pod specification can declare multiple containers, by defining under the hoods shared namespaces and resources. In fact, containers in a pod share namespaces among them. In this exercise, we will see how to create a pod from scratch by using the `pause` container and sharing namespaces.

First, we will need to start the **pause** container with Docker so that we can add our containers to the pod.

```
docker run -d --ipc=shareable --name pause -p 8080:80 \
    gcr.io/google_containers/pause-amd64:3.0
```

This container is normally hidden in Kubernetes, you will not notice it via a kubectl, but if you type a *docker ps*, you will see plenty of pause containers.

Now, we can add the containers to our "pod". First we will run an nginx instance (proxy), configured to proxy the requests to a second container (application server).

[15] `https://www.youtube.com/watch?v=Utf-A4rODH8`

Note that when we started the pause container, we also mapped the host port 8080 to port 80 on the pause container. This has be done on this first container rather than the nginx container because the pause container sets up the initial network namespace that nginx will be joining.

First of all, we should create the configuration file `nginx.conf` to configure the proxying:

```
error_log stderr;
events { worker_connections  1024; }
http {
     access_log /dev/stdout combined;
     server {
         listen 80 default_server;
         server_name example.com www.example.com;
         location / {
             proxy_pass http://127.0.0.1:2368;
         }
     }
}
```

This code snipped is attached to the PDF file as "`snippets/nginx.conf`"

And then launch the nginx container with the mounted volume:

```
docker run -d --name nginx -v $(pwd)/nginx.conf:/etc/nginx/nginx.conf \
    --net=container:pause --ipc=container:pause --pid=container:pause nginx
```

And then we will create another container for the ghost blog application which serves as our application server.

```
docker run -d --name ghost --net=container:pause \
    --ipc=container:pause --pid=container:pause ghost
```

As reported on Wikipedia, "Ghost is a free and open source blogging platform written in JavaScript and distributed under the MIT License, designed to simplify the process of online publishing for individual bloggers as well as online publications." This will represent the final target for our pod.

In both cases we specify the pause container as the container whose namespaces we want to join. This will effectively create our pod.

Now, if you access `http://localhost:8080/` on your host, you should be able to see ghost blog running through an nginx proxy because the network namespace is shared among the pause, nginx, and ghost containers.

Let's connect to the nginx container and launch `htop`:

```
docker exec -it nginx /bin/bash
```

And then inside the ghost container:

```
apt update && apt install htop
htop
```

You can observe that all the containers share the same PID namespace and pause is the "init" process, with number 1.

With this exercise, we have just manually created a "pod" Kubernetes-style. The nginx server can be also configured with a TLS endpoint in order to provide secure exposition. This can be done without modifications on the **ghost** pod, which represents our main application. The choice of pause container

to use as "init" (PID 1) container is also justified by another task: reaping the zombie processes as we will see in next section.

3.5.5 PIDs in Linux

As we described in section 2.3, in Linux, processes in a PID namespace form a tree with each process having a parent process identified by a PPID (Parent Process ID). Only one process at the root of the tree doesn't really have a parent. This is the "init" process, which has PID 1.

Processes can start other processes using the `fork` and `exec` syscalls. When they do so, the new process' parent is the process that called the fork syscall. The main difference between fork and exec is that fork is used to start another copy of the running process and exec is used to replace the current process with a new one. In fact, using exec the process will keep the very same PID. In the case that you want to run a completely separate application you need to run fork to allocate the a process and PID, and then modify its behavior via the exec syscall.

In fact, the result of the fork is a new copy of itself with a new PID. Then, when the child process runs, it executes the exec syscall to replace itself with the one you actually want to run.

Zombie processes are processes that have stopped running but their process table entry still exists because the parent process has not retrieved it via the `wait` syscall. Technically each process that terminates is a zombie for a very short period of time but they could live for longer.

Longer lived zombie processes occur when parent processes don't call the wait syscall after the child process has finished. When a process' parent dies before the child, the OS assigns the child process to the "init" process or PID 1. i.e. The init process "adopts" the child process and becomes its parent. This means that now when the child process exits the new parent (init) must call wait to get its exit code or its process table entry remains forever and it becomes a zombie.

Resource consumption is the main reason to wipe dead processes. When a process is dead, all resources associated with it are deallocated so that they can be reused by other processes. A zombie process does not use more memory than is required for keeping its entry in the resource table, which is negligible. The problem occurs when you have too many zombies, since the OS has a limited number of PIDs.

3.5.6 The problem of "Zombies" in Containers

In containers, one process must be the **init** process for each **PID namespace**. With Docker, each container usually has its own PID namespace and the process set as `ENTRYPOINT` process in the Dockerfile is the init process. However, as we show in this last exercise, a container can be made to run in another container's namespace. In this case, one container must assume the role of the init process, while others are added to the namespace as children of the init process. For example:

```
docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf -p 8080:80 nginx
docker run -d --name ghost --net=container:nginx --ipc=container:nginx \
    --pid=container:nginx ghost
```

In this case, nginx has to assume the role of PID 1 and ghost is added as a child process of nginx. If, for instance, ghost forks itself or runs child processes using exec, and crashes before the child finishes, then those children will be adopted by nginx. However, nginx is not designed to be able to run as an init process and reap zombies. That means we could potentially have lots of them and they will last for the entire life of that container.

4 Appendix

4.1 Installing cgroup-tools

First, check if `cgroup-tools` are already installed on your system by typing:

```
command -v cgcreate >/dev/null && echo "Installed" || echo "NOT installed"
```

In case of negative answer, you can install them issuing `sudo apt install cgroup-tools` or the equivalent command according to the Linux distribution you are currently using.

4.2 Ansible playbook

In the following, you can find an *ansible playbook* that installs and configures all the packages required for this laboratory. Warning: the playbook is not general enough and has *not* been thoroughly tested. Thus, it may not work with certain distributions or releases.

```
---
- hosts: all
  become: yes

  tasks:
  - name: Update the apt cache
    apt:
      update_cache: yes

  - name: Install the prerequisites
    apt:
      name: "{{ lab_prerequisites }}"
      state: present
    vars:
      lab_prerequisites:
      - tree

  - name: Install the tools required to manage namespaces and cgroups
    apt:
      name: "{{ ns_cgroups_packages }}"
      state: present
    vars:
      ns_cgroups_packages:
      - cgroup-tools
      - util-linux

  - name: Install Docker prerequisites
    apt:
      name: "{{ docker_prerequisites }}"
      state: present
    vars:
```

```
    docker_prerequisites:
    - apt-transport-https
    - ca-certificates
    - curl
    - gnupg-agent
    - software-properties-common

- name: Add Docker GPG key
  apt_key: url=https://download.docker.com/linux/{{ansible_distribution|lower}}/gpg

- name: Add Docker APT repository
  apt_repository:
    repo: deb [arch=amd64] https://download.docker.com/linux/{{ansible_distribution
        |lower}} {{ansible_distribution_release}} stable

- name: Install Docker packages
  apt:
    name: "{{ docker_packages }}"
    state: present
  vars:
    docker_packages:
    - docker-ce
    - docker-ce-cli
    - containerd.io

- name: Install Docker-compose
  get_url:
    url: "https://github.com/docker/compose/releases/download/{{ compose_version
        }}/docker-compose-{{ansible_system}}-{{ansible_architecture}}"
    dest: /usr/local/bin/docker-compose
    mode: +x
  vars:
    compose_version: 1.24.1

- name: Add the local user to the docker group
  user:
    name: "{{ ansible_user }}"
    append: yes
    groups: docker
```

This code snipped is attached to the PDF file as "`snippets/4A-DockerlabPlaybook.yml`"