NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

# Docker containers and UDP sockets in Python

(Lab-01 – 11/12 September 2025)

# Part 1 – Docker containers

**Summary**

- Virtualization
- Docker
- Exercises

## 1 Virtualization

In short, hardware virtualization is a technique for creating virtual instances of a computing platform, using software to "simulate" the virtual hardware platform.

### 1.1 Virtual machines

A virtual machine is a form of virtualization that is intended to simulate a full hardware platform on a real machine - the host machine, on top of which a *guest* operating system is installed and operated. The *guest* operating system can then be used to execute **multiple applications** on behalf of the user.

The technique has applications in legacy systems, security and **distributed systems**.

### 1.2 Containers

Containers is a restricted form of OS-level virtualization that is mostly intended to virtualize (and isolate) the execution environment of **specific applications**.

Containers do not execute full dedicated virtualized operating systems. Instead, the **host operating system** only provides a virtualized environment to the target application and its dependencies, thus, consuming fewer host resources.

Containers are a relatively lightweight and simple way to package and deploy complex applications/services.

Docker and Kubernetes are examples of container technologies.

# 2 Docker

Docker is a container technology that can be used in personal computers with modest CPU and RAM requirements. It is available for Windows, Linux and MacOS.

Usually, when we talk about **docker containers**, we mean Linux docker containers, packaging Linux applications/services.

## 2.1 Docker under the hood

In Linux, Docker executes containers **natively** using the isolation provided by the kernel, via *namespaces*.

In MacOS, a docker installation consists of a (hidden) guest Linux virtual machine that hosts and executes the docker containers. The MacOS Docker application interfaces with a server executing in the guest virtual machine.

In Windows, docker containers also rely on the services of a Linux kernel. In recent Windows versions, that kernel is tightly integrated to the Windows kernel via the Windows Linux Subsystem. If the Docker Windows Linux Subsystem backend is not supported on the host machine, an alternative way is to use a virtual machine running a guest Linux installation, and install docker there.

## 2.2 Docker architecture

Docker architecture comprises several components.

- **Docker registry:** A (cloud-based) repository of images. An image contains the static environment (files and executables) needed to execute a container;
- **Docker client:** User application for managing containers, eg., running and stopping containers;
- **Docker daemon** A server that executes in the host machine, listening for commands from the docker client. Do the actual operations on containers.

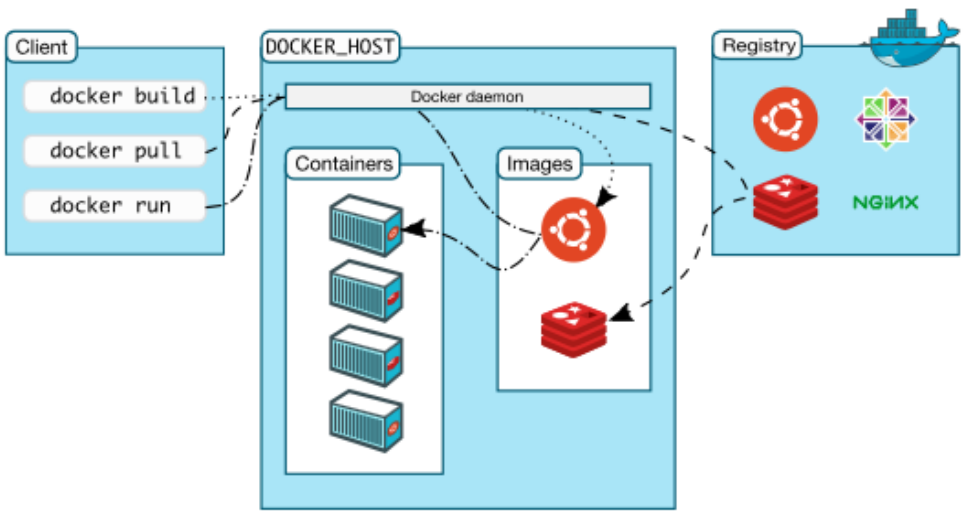


Figure 1: Docker architecture [diagram pulled from docs.docker.com]

## 2.3 Docker installation

Install docker in your personal computer, using one of these guides:

- Linux
- Windows
- MacOS

Note: In Windows, it may be necessary to enable CPU virtualization at the BIOS level. Some Windows versions do not support WSL and the only option may be to install a Linux virtual machine, using VirtualBox, for example.

In Linux, make sure you perform the post installation step to be able to run docker without *sudo* (root privileges).

## 2.4 Docker images

Docker images are built from a set of commands stored in a text file: Dockerfile

A new image starts from a *base image* and is modified mainly by adding files to it or executing Linux commands to install the target application and its dependencies.

**Dockerfile example:**

```
FROM ubuntu
RUN apt-get update && \
        apt-get install -y iputils-ping iproute2 python3
```

The Dockerfile above will create a container image FROM a base *ubuntu* image, which is already present in the docker registry.

The RUN command is used to install the ubuntu packages iputils-ping, iproute2 and python3, via the apt-get utility.

Check the Dockerfile reference for full syntax and semantics of the available Dockerfile commands.

**Managing Docker images**

The relevant CLI commands are:

- `docker build . -t <img>`
  - creates an image with name tag <img> using the Dockerfile contained in the current directory;
- `docker push <img>`
  - uploads the image with name tag <img> to the docker registry (the user must have an account);
- `docker images`
  - lists the images stored in the local host repository
- `docker rmi -f <img>`
  - tries to remove forcibly an image from the local host repository

## 2.5 Exercise 1

Create a docker image using the Dockerfile above. The image should be named rc2526-lab01

1. Create a new directory/folder named lab01;
2. Place a text file named Dockerfile in lab01 with the contents from the example above;
3. Build the image with the command docker build ...;
4. List the images found in the local repository with docker images to confirm the image name is as expected;
5. Remove the image you just built with docker rmi -f;
6. Rebuild the image as in step 1.

Optionally, push the image to the Docker repository. You will need to create a free account, and name your images, accordingly, like so:

<login>/<image>

## 2.6 Managing containers

There are several CLI commands to manage the execution of containers. Moreover, the behavior of each command is controlled by a number of flags.

**Launch**

- `docker run -d <img>`
  - launches a new container from image <img>.

  The flag -d means the new container will run in detached mode.
- `docker run --name test1 -t -i <img>`
  - the new container named "test1" runs attached to an output terminal -t and in interactive mode -i.
- `docker run -t -i <img>`
  - the new container runs attached to an output terminal -t and in interactive mode -i.
- `docker run -h C1 -t -i <img>`
  - the new container is launched and given C1 as its hostname.

**Listing**

- `docker ps`
  - lists the containers that are running.
- `docker ps -a`
  - list all containers, including those that are stopped or terminated.

**Stop and Start**

- `docker stop <id>`
  - stops the given container, keeping its saved state.
- `docker start <id>`
  - (re)starts the given container from its saved state.

**Remove**

- `docker rm -f <id>`
  - removes the given container, forcing stop if necessary.

## 2.7 Docker container networking

By default, when containers are launched, each receives a **separate**, dedicated, networking environment, including a private IP networking address.

Communication with the outside world goes through the host system, which acts as a router.

Docker containers can talk with each other, but it is possible to isolate them completely, for example for security reasons. This isolation can be achieved by attaching containers to different *docker networks*.

The obtain the networking information about a given container use the following command in the host computer:

```
docker inspect <id>
```

The inspect command dumps all kinds of information about a container, given its id, including its networking details.

*grep* is a command line utility that can be used to filter and isolate only the output lines that match a given word or pattern:

```
docker inspect <id> | grep IPAddress
```

## 2.8 Exercise 2

1. Launch a new container using the image rc2526-lab01 that you created earlier.

   ```
   docker run -ti rc2526-lab01 /bin/bash
   ```

2. In the container shell, type:

   ```
   ip addr show | grep inet
   ```

   This will show the IP address of the container.

3. In the host, inspect the container:

   ```
   docker inspect <id> | grep IPAddress
   ```

   Use docker ps if necessary to find out the container id.

## 2.9 Docker storage

By default, the filesystem of a container is not directly accessible to the host system (or other containers), and vice versa.

However, it is possible to expose parts of the host filesystem to a container.

Namely, using *-v <host_dir>:<container_dir>* in the docker run command, we can map a directory in the host filesystem to a directory in the container filesystem, like so:

```
docker run -ti -v "/usr/local/…/lab01:/tmp" rc2526-lab01
```

The link should now list the contents of directory */usr/local/…/lab01* in */tmp* of the container.

# Part 2 – UDP Sockets in Python

Processes running in distinct machines can communicate with each other using **sockets** which are devices allowing the access to Internet transport protocols like TCP and UDP. In this class, only UDP sockets will be used.
Learning Materials:

- Python basic concepts can be studied in several online texts namely *Python for Everybody: Exploring Data in Python3* by Charles R. Severance et al, available (including a tar file with the code of the examples used in the book) at https://www.py4e.com/book.php
- How to program UDP sockets in Python – Client and Server Code Example: https://www.binarytides.com/programming-udp-sockets-in-python/

## Exercise 3 - The Python Program

Please get the source code of *UDP_client.py* and *UDP_server.py* available from CLIP. Study the source code, then run the client and server in separate Docker containers. You can use the technique described in section 2.9 "Docker storage" to view and execute the source code files from your machine's filesystem in the created containers. Once inside a container, invoke the Python interpreter in a bash shell to run the client or server Python program.

## Exercise 4 - Adding two numbers

In this assignment, you'll write a client that will use sockets to communicate with a server that you will also write. Here's what your client and server should do:
Your client should first accept an integer between 1 and 100 from the keyboard, create an UDP socket and send a message to your server containing (i) a string containing your name (e.g., "Client of John Q. Smith") and (ii) the entered integer value and then wait for a server reply.
Your server will create a string containing its name (e.g., "Server of John Q. Smith") and then begin accepting connections from clients. On receipt of a client message, your server should:

i. print (display) the client's name (extracted from the received message) and the server's name;
ii. pick an integer between 1 and 100 (it's fine for the server to use the same number all the time) and display the client's number, its number, and the sum of the two numbers;
iii. send its name (string) and the server-chosen integer value back to the client;
iv. if your server receives an integer value that is out of range, it should terminate after releasing any created sockets. You can use this to shut down your server.

Your client should read the message sent by the server and display its name, the server's name, its integer value, and the server's integer value, and the computed sum. The client then terminates after releasing any created sockets.

## Exercise 5 (challenge) - File Transfer

Try to implement a file transfer program where:

- The client is invoked with the name of the file to be transferred from the server;
- The client sends a datagram to the server with the file name in the server's file system;
- The server sends the file in 2048 bytes chunks;
- Client receives the blocks and writes them to its local disk. End of file is detected when the size of the chunk is less than 2048.