Lab1: Introduction to Docker & ROS2 Command Line Interfaces (CLIs)

Table of Contents

Setup

Docker allows you to package and run applications in a loosely isolated environment called a *container*. The isolation and security features enable you to run multiple containers simultaneously on a single host. Containers are lightweight and include everything needed to run the application, eliminating the need to depend on software installed on the host.

The use of ROS2 in this class makes Docker and similar tools valuable, as they allow us to leverage ROS's extensive support for the Ubuntu (Linux) operating system (OS).

Let's start by first installing Docker Desktop on your computer. Scroll to the bottom of the documentation page and click the hyperlink corresponding to your OS. Docker Desktop currently supports Windows, macOS, and Linux. Follow the installation instructions, which also cover common errors and troubleshooting methods.

Docker Command Line Interface (Docker CLI)

The Docker CLI offers several useful features. It supports standard UNIX-style arguments and often provides both short and long flag options. For example, the flags -H, --host are both for specifying the host to connect. Additionally, the CLI accepts environment variables, and can also accept configuration from a file (not covered). Most of the reference material can be found on the official documentation site.

For the purposes of this course, we focus on a small subset of Docker CLI commands – notice their similarity to the UNIX interface. These include:

- `docker start`
  - Start a container
- `docker pull`
  - Pull an "image" from the docker repo-hub
- `docker exec -it`
  - Execute command in a container. The flags "-it" mean "interactive terminal" and can be used in conjunction with the "exec" command to not only run the command but also open a new terminal inside the container.
- `docker run -it`
  - Run an instance of the image and start up a container and terminal

In the context of Docker, an *image* contains all the instructions and files needed to run an application (OS, code, dependencies) but it is not running. We would use a *Dockerfile* to build an image (e.g. *ubuntu:latest*). A container, on the other hand, is the actual process executing the application with isolated resources (CPU, memory, etc). For example, executing the command *docker run ubuntu:latest* creates a container from the *ubuntu* image. Furthermore, you can have one image but many containers running from that same image. Containers are meant to be temporary; images stay stored unless deleted.

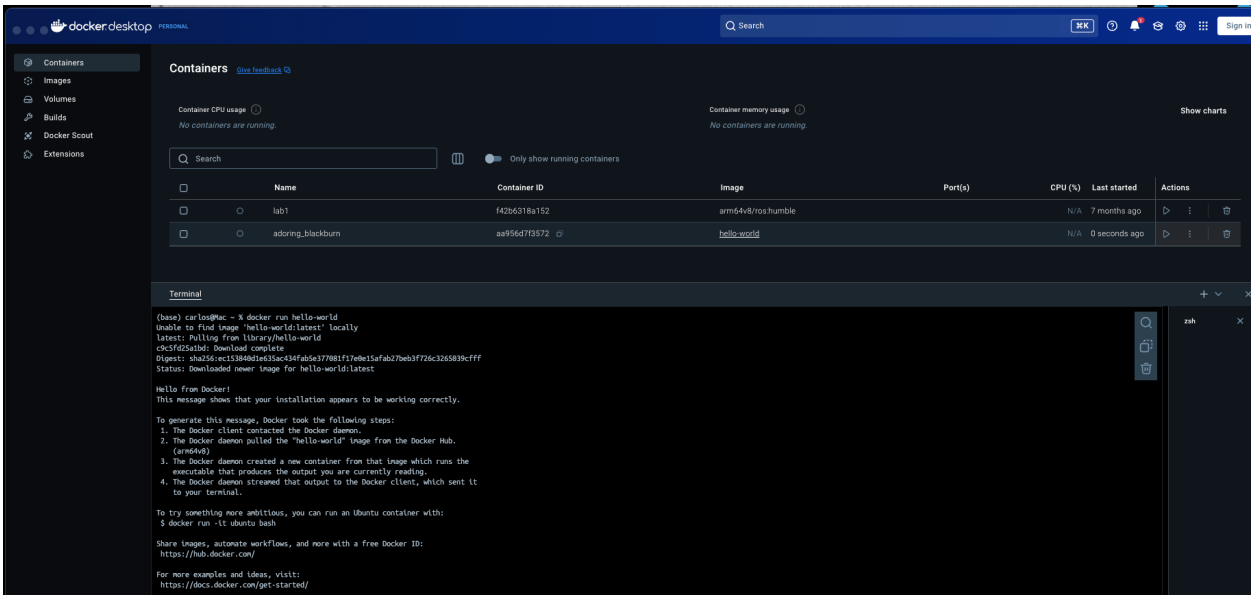
Docker Desktop

Run hello-world

First let's start by making sure that our installation works correctly. Open Docker Desktop, click on the terminal option (bottom right corner) and run the following command:

```
docker run hello-world
```

You should see the following output if the command successfully executed.



Screenshot of Docker output after executing the run hello-world command

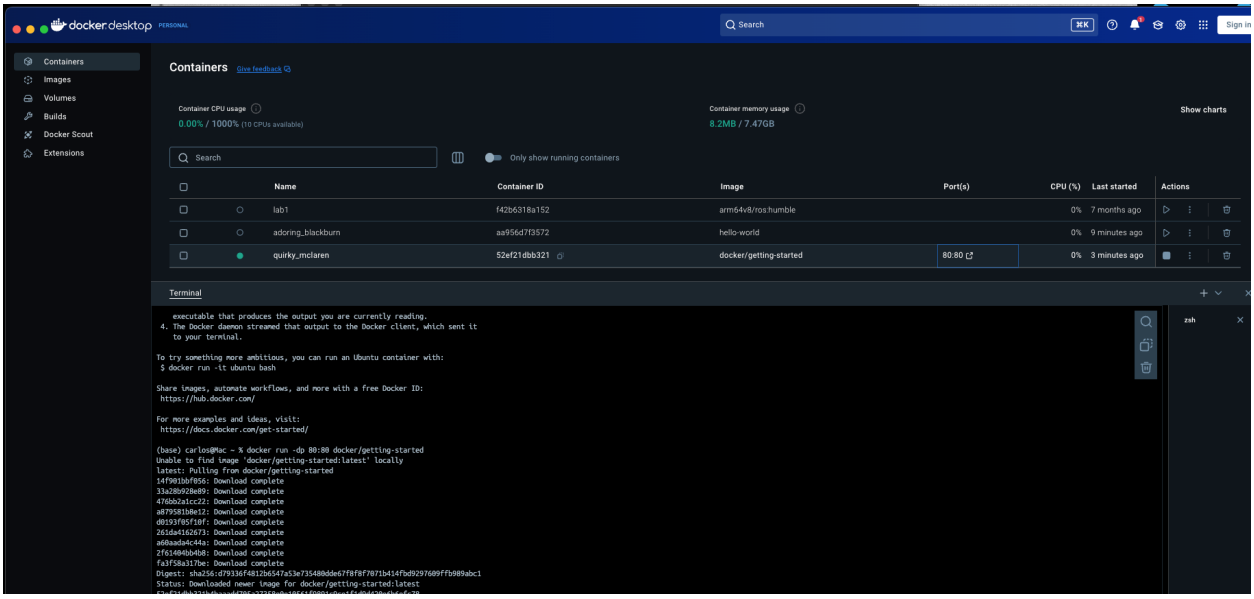
When you first run a Docker command (in this case `docker run`) for an image that isn't already downloaded locally, Docker automatically handles the entire setup process. First, it checks your local system for the specified image—like searching your computer for a file. When it doesn't find the image, Docker connects to Docker Hub (a public repository of container images) and downloads the image along with all its necessary components. Once downloaded, Docker uses the image to create a fully isolated container environment with its own filesystem and resources. This entire process happens in the background, requiring no extra effort from you beyond running the original command.

Containers Tab

Next, run the following command:

```
docker run -dp 80:80 docker/getting-started
```

Again, this command will first check if the image is stored locally (and if the image is not then download) and then output a long string of characters (the id of the container). If the command executed correctly, you should be able to click the new link provided in the Port(s) column of the new container you created. Refer to the image below for an idea of the layout.



This successful connection confirms that Docker's networking stack and port mapping functionality are properly configured on your system. When you're able to access services running inside a container through specified ports, it verifies some important components: First, Docker's internal virtual network is correctly routing traffic between your host machine and the container. Second, the port forwarding rules established are functioning as intended. Third, there are no conflicts or network restrictions blocking the communication. The fact that you can successfully reach your containerized service demonstrates that all these layers are working.

Pulling an Ubuntu ROS Image and Starting a Container

1. First, let's pull an Ubuntu ROS2 image. In other words, an image where Ubuntu and ROS2 are already installed.

Choose the command based on your processor architecture:

- For an x86 image (Intel Chips):
  - `docker pull osrf/ros:humble-desktop`

- For an arm64 image (Apple Silicone, etc.):
  - `docker pull arm64v8/ros:humble`

2. Next, start a container with an interactive shell

- We can start a container from an image and connect via a bash shell by using the `docker run` command. Note the syntax below:
  - `docker run -it --name YOUR_CONTAINER_NAME IMAGE_NAME`

- For example, to use docker to open a terminal inside the image we just pulled we could run the command (note I am working on Apple Silicone):
  - `docker run -it --name lab1 arm64v8/ros:humble`

3. If the last command executed correctly, you have created a container from an Ubuntu image and accessed it through a terminal shell. The container can be treated just like a UNIX environment with ROS2 installed.
   - For example, you can run the `ls` command to view the */root* filesystem of the container. Since this is an Ubuntu container you will see the files needed for the operating system to function.
   - All necessary files should be present
   - Note, the container will continue running if the shell session is active. To simultaneously stop and exit the container you may use Ctlr+D

Finally note the following:
- Containers can be active (running processes) or inactive (stopped but still existing)
- Multiple containers can be active simultaneously from the same image
- Activating a container is not the same as connecting a terminal shell. With the `docker exec` command we can connect to an already active container
- Exiting the container does not necessarily mean stopping the container. To exit the shell but keep the container running press the following: Ctrl+P, followed by Ctrl+Q

ROS2 Command Line Interface (ROS2 CLI)

To continue with this portion of the lab assignment, you'll need an active bash shell connection to a running ROS2 Humble container. Any container created from the Docker image you pulled earlier (either *arm64v8/ros:humble* for ARM systems or *osfr/ros:humble-desktop* for x86 architectures) will work for this purpose. First, verify that your container is running by executing `docker ps` in your host terminal - this will list all active containers. If you see your ROS2 container in the list but aren't currently connected to it, you can attach to its shell using `docker exec -it <container_name_or_ID> bash`. You can also view all containers (active and inactive) from the *Containers* tab in the Docker Desktop application.

Should you need to start a new container, simply run the appropriate docker run command with the image you previously downloaded. Once connected, your terminal prompt should change to indicate you're now operating inside the container environment. This containerized environment provides all the necessary ROS 2 tools and dependencies while maintaining isolation from your host system.

Sourcing the Underlay

Now that we've established a terminal shell inside our ROS2 Humble container, let's explore the ROS2 command line interface (CLI). Before we can use any ROS2 commands, we need to configure our environment by sourcing the ROS2 setup file:

```
source /opt/ros/humble/setup.bash
```

The above command does several important things:

- Makes all ROS2 commands available in your current shell
- Sets up necessary environment variables
- Configures paths to ROS2 tools and libraries

Important Note: You'll need to run this sourcing command every time you open a new terminal session in the container. This is particularly crucial if you're working with multiple ROS versions, as it ensures you're using the correct version's tools and environment.

After sourcing, let's verify everything is working properly by checking the ROS2 CLI. Run the following command:

```
ros2
```

This should display the main help menu showing all available ROS2 commands. If you see this output, congratulations - your ROS2 environment is properly configured and ready for use!

Practice Questions (2 points each)

Before attempting these questions, ensure you have completed all previous sections of the lab.
Important Notes:

- Several of the questions below will require you to consult the official ROS 2 documentation. This is intentional—the lab is designed to:
  - Develop your ability to navigate technical documentation
  - Encourage independent exploration of ROS2 concepts
  - Bridge between guided exercises and independent problem solving
- While this lab provides a structured starting point:
  - Treat it as a springboard for deeper learning, **not** a comprehensive resource
  - Expect to supplement it with external research (tutorials, forums, ROS2 wiki, videos)
  - The most effective learning often happens when you *follow the thread* of curiosity beyond the prescribed material
- Suggested Approach
  - Answer the questions using the lab's framework, references and official documentation
  - I encourage you to compare your solutions with classmates
  - Finally, explore related concepts not covered here on your own

1. How many sub-commands does the ros2 cli have? What is the sub-command for package related issues? (Command Line Interface)

2. What is the command for listing the active topics? How many topics are currently running? (Topics)

3. What is the command for listing the active nodes? How many nodes are currently running? (Nodes)

4. What kind of messages are used by the */rosout* topic? How many publishers are sending data to that topic? (Messages)

5. How many string member variables are part of the message type being used in the */rosout* topic? What are their names? (Messages)

6. What is the ros2 command for listing the currently installed packages? What is the output when you run the following command `ros2 pkg list | grep urdf`? (Packages)

7. What is the command for installing the *turtlesim* package in the container? What is the command for updating the package? (Installing a package)

8. Why is it not a good idea to install packages inside a container?

9. How many executables does the *demo_nodes_cpp* package have that begin with the prefix *talker*? (Executables)

10. Run the *talker* executable form the *demo_nodes_cpp* package. Describe the output of the node. (Nodes)

Now, open a separate terminal session and enter the same container via a new bash session. Note that the node from the previous question should be left running.

11. What topics did the executable start? What kind of ros messages are being sent over the topic?

12. How many member variables does that message type from the previous question have in its definition?

13. Run the listener executable from the same *demo_nodes_cpp* package. Describe the output of the *listener* node.

At this point, you should have two different terminals connected to the same container running two different nodes that are communicating with each other!

14. Draw a picture of what the ROS computation graph looks like while running the talker and listener nodes?