Tekhnê
January - June 2022, Vol. 19, No. 1, pp. 13 – 22

ISSN 1692-8407

Docker: A tool for creating images and launching multiple containers with ROS OS

Docker: Una herramienta para crear imágenes y lanzar múltiples contenedores con ROS OS

Fredy H. Martínez S.
Facultad Tecnológica, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia
fhmartinezs@udistrital.edu.co

Docker is a tool that allows to create containers with everything needed to run an application. This feature makes it key to the process of transferring software products in different environments, allowing code to be ported faster, with better use of resources, and more reliably. ROS 2 (Robot Operating System 2) is an open source SDK for robotics applications that provides hardware abstraction for control, which can benefit from the use of containers. This article presents an introduction to Docker, creating images, launching multiple containers, and most importantly, how it can be used in conjunction with ROS OS for robotics applications.

Keywords: Container, docker, image, robotics, ROS

Docker es una herramienta que permite crear contenedores con todo lo necesario para ejecutar una aplicación. Esta característica la convierte en clave para los procesos de transferencia de productos software en diferentes entornos, al permitir transferir código con mayor rapidez, mejor uso de recursos, y de forma más confiable. ROS 2 (Robot Operating System 2) es un SDK de código libre para aplicaciones en robótica que proporciona abstracción del hardware para el control, que puede verse beneficiado por el uso de contenedores. En este artículo se presenta una introducción a Docker, la creación de imágenes, el lanzamiento de múltiples contenedores, y sobre todo, cómo se puede utilizar en conjunto con ROS OS para aplicaciones de robótica.

Palabras clave: Contenedor, docker, imágen, robótica, ROS

Article typology: Research

Received: March 25, 2022
Accepted: May 27, 2022

Research funded by: Universidad Distrital Francisco José de Caldas (Colombia).

How to cite: Martínez, F. (2022). *Docker: A tool for creating images and launching multiple containers with ROS OS*. Tekhnê, 19(1), 13 -22.

Introduction

In robotics and software development in general, you normally need different applications to interact among them (Baltes & Diehl, 2018). This is necessary to ensure the smooth flow of data and information between different systems so that they can work in tandem with one another (Martínez, 2021). However, in many cases, certain software requires specific conditions in the operating system and library versions that make it complex to run on any other machine (Martínez et al., 2018). As a result, it is often difficult to maintain compatibility between different applications and software programs.

The only thing that never changes in the software development sector is change (Immaculate et al., 2019). This implies that software is constantly changing to meet changing demands. If automated testing does not adequately cover a change when it occurs, something is likely to break. As such, software developers must keep up with the changing landscape, constantly updating and refining the software they develop to remain competitive. To ensure this happens, software developers must have a robust automated testing process in place to quickly identify any changes in the software that could cause an unexpected break.

In a computing context, "permissions" refer to the ability of a user or system process to access and make changes to certain files or folders on a computer (Feng et al., 2019). When you change the permissions on a file, you are altering what actions other users or processes can perform on that file (Han et al., 2020). For example, you might give a group of users read-only access to a file, meaning they can view the contents of the file but cannot make any changes to it. Sometimes, it is necessary to change permissions on a file to allow certain processes to run or to restrict access to sensitive information. However, making changes to permissions can also have unintended consequences. If you change the permissions on a file that is essential to the operation of your system, it may cause problems or errors to occur. Similarly, deleting a folder or changing a configuration file can also cause problems. Configuration files contain important settings and instructions that tell your system how to operate. If you delete a configuration file or alter it in a way that is incompatible with your system, it could cause your system to malfunction. It is important to be careful when making changes to the permissions, files, or folders on your system. If you do need to make changes, it is a good idea to test the changes in a controlled environment before implementing them on your production system. This will allow you to identify any potential problems and fix them before they cause issues for your users. If something does go wrong after you have made changes to your system, it can be difficult to diagnose the problem. If you have only changed a single file, it may be easier to track down the cause of the problem. However, if you have installed, deleted, or updated software, or changed folder permissions recursively (meaning that the changes were applied to all subfolders and files within a folder), it may be more challenging to identify the root cause of the issue. In these cases, it may be necessary to perform a thorough analysis of your system to determine what has caused the problem and how it can be fixed.

Docker is a tool that allows developers to package applications in containers, which are lightweight and portable software packages that include all of the necessary dependencies and libraries needed to run an application (Pan et al., 2019). Using Docker can help to solve many of the problems that can occur when running and deploying applications. One major benefit of Docker is that it allows you to create a consistent environment for your application to run in. When you create a Docker image, it is a snapshot of your application and all of its dependencies at a specific point in time. This means that once you have created a Docker image, it will never change. This is in contrast to traditional application deployment methods, where it is common for applications to be updated and modified over time. With Docker, you can be confident that your application will always run in the same way, regardless of any changes that may have been made to the host operating system or other software on the system.

Another advantage of Docker is that it makes it easy to share applications with others (Diekmann et al., 2019). Once you have created a Docker image, you can share it with anyone else, and they should be able to run the application in the same way that you do. This is because the Docker image includes all of the necessary dependencies and libraries needed to run the application, and these are bundled together in a single package. This means that there is no need to worry about configuring the environment or installing additional software to run the application. Overall, Docker provides a reliable and consistent way to run and deploy applications, and it can help to avoid many of the problems that can occur when running applications in traditional environments. By using Docker, you can be confident that your application will always run in the same way, no matter how many times you try.

When you create a Docker image, you are taking a snapshot of your application and all of its dependencies at a specific point in time. This image serves as a template for creating containers, which are essentially running instances of the image. If you want to run your application in a Docker container, you simply create a new container from the image and start it up.

One of the key advantages of this approach is that it allows you to be sure that you can always replicate the same environment for your application (Moreno et al., 2022). Because the image contains all of the necessary dependencies and libraries, you can be confident that your application will run in the same way every time you create

a new container from the image. This means that you can be sure that you can always replicate the same results when you run your application, regardless of the environment or platform you are using.

It is possible to install or update different applications within a Docker container, but these changes will not affect the original image. Instead, they will only be applied to the running container (Akhilesh et al., 2021). If you want to make changes to the original image, you will need to create a new image with the same label as the original image, replacing it. This allows you to easily create new versions of your application while still preserving the original image as a reference point.

In the software industry, it is common for companies to use applications developed by other parties. These applications may be off-the-shelf software products that are widely available, or they may be custom-built applications that have been developed specifically for the company. While most software is developed with good intentions and is generally reliable, there are instances where problems can occur.

One potential problem with using software developed by others is the risk of critical bugs or vulnerabilities. A bug is a mistake or error in the code of an application, and it can cause the application to behave unexpectedly or fail to function properly. A critical bug is a bug that has the potential to cause serious problems or disruptions. For example, a critical bug might allow an attacker to gain unauthorized access to a system or steal sensitive information.

Another risk of using software developed by others is the potential for malicious or maliciously-crafted applications. In some cases, software may be developed with the intention of causing harm or exploiting vulnerabilities in systems. For example, an attacker might develop a malicious application that appears to be legitimate software, but which contains hidden code that allows the attacker to gain access to a system and steal sensitive information.

When you create a Docker container, you can specify which resources on your computer the container has access to. For example, you can specify how much memory and CPU a container can use, as well as which files or folders the container has access to. This allows you to control the impact that the container has on your system, and to ensure that it does not consume more resources than you are willing to allocate to it.

You can also specify the permissions that a container has when accessing resources on your system. For example, you can make a file or folder read-only, meaning that the container can view the contents of the file but cannot make any changes to it. This helps to prevent accidental or malicious modification of important resources on your system.

Using Docker allows you to have full control over your computer, as you can specify exactly which resources containers have access to and what permissions they have when accessing those resources. This can help to protect your system from potential problems or vulnerabilities that may arise when running containers.

Literature review

In ROS OS, the files are in XML format and are difficult to write and understand for nontechnical users. To address this problem (Narayanamoorthy et al., 2015) propose a visual programming software tool that helps in the creation and visualization of these ROS launch files. (Naik, 2016a) present the simulation of building a virtual system of systems (SoS) for the distributed software development process on multiple clouds. (Naik, 2016b) present the simulation and evaluation of the development of a distributed system using virtualization and dockerization. (Kelley et al., 2016) propose a quantum network security framework for the cloud. They also decrease the overall security, as each included component-necessary or not-may bring in security issues of its own, and there is no isolation between multiple applications packaged within the same container image. (Rastogi et al., 2017) propose algorithms and a tool called Cimplifier, which address these concerns: given a container and simple user-defined constraints, the tool partitions it into simpler containers, which (i) are isolated from each other, only communicating as necessary, and (ii) only include enough resources to perform their functionality. Recent massive data projects have revealed several bottlenecks for projects with >100,000 assessors (i.e., data processing pipelines in XNAT). In order to address these concerns (Damon et al., 2017) develop a new API, which exposes a direct connection to the database rather than REST API calls to accomplish the generation of assessors. The live migration may cause significant delays when the applications running inside a container modify large amounts of memory faster than a container can be transferred over the network to a remote host. (Stoyanov & Kollingbaum, 2018) propose a novel approach for live migration of containers to address this issue by utilizing a recently published CRIU feature, the so-called “image cache/proxy”. To reduce the barrier for newcomers and to prevent deprecation of aging software (East et al., 2019) create the NMRdock container. Other influential work includes (Chouhan et al., 2021; Trapti Gupta and Abhishek Dwivedi, 2017).

Docker Images

Docker is a software tool that simplifies software’s building, running, managing, and distributing. It works by virtualizing a computer’s operating system, on which the software is installed and executed. In this sense, Docker works as a box in which it can install an application under

specific operating system conditions and keep it running in isolation from the rest of the existing software on the machine. In principle, each software is isolated in its box or container and therefore does not interfere with other existing software, which generally happens on a personal computer. This occurs easily when working in an application development environment and when desired to guarantee that the application under development will run the same way in any other machine (everything always works the same inside the box, which can be inside a robot).

The first step is to install Docker on the local machine or host. For Linux Mint, Ubuntu, MX Linux and other Debian derivatives, in a terminal window run one by one the following commands:

```
sudo apt update
sudo apt install docker.io docker-compose
```

And then the Docker service is started:

```
sudo service docker start
```

At this point it is possible to use Docker commands. To be able to use the docker command from any user and without the need of the sudo command, it is necessary to add it to the docker group:

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

Docker can also be installed natively on Windows 10 onwards (for Windows 7 and 8 it is necessary to use Virtualbox). In the case of not having virtualization enabled, Docker will detect this feature in the installation process and will ask to install it automatically. To check it, you can use the *Task Manager*. Docker Desktop for Windows can be downloaded from this link:

https://docs.docker.com/desktop/install/windows-install/

Download and double click on the file to install. When finished Docker runs automatically.

The next step will always be to ensure that there is no ROS publisher or subscriber running by previous jobs. These can be stopped with the key combination **CTRL + C** on each of the terminals. The box where the software is isolated for execution is called Docker Image. Pulling a Docker image means downloading a Docker image to a local location (our computer, for example) from some repository. For example, let's pull a Docker image that contains ROS Foxy, in a terminal type:

```
docker pull ros:foxy-ros-base
```

This command produces in the terminal a response similar to the one shown in Fig. 1, which indicates that the image is pulling a Docker image that contains Docker ROS Foxy installed (note the Status line at the end). The `docker pull` command is used to pull an image from a registry. Now, what if we want to work with ROS Noetic? In this case, to work with this other Distro, we simply download the desired image, which can be done with the following command:

```
docker pull ros:noetic-ros-core
```

Note that both are done with a simple command, and in neither case was it necessary to install them on the computer. To determine which Docker images are available, check with the following command:

```
docker images
```

Additional help for the `docker pull` command can be obtained as follows:

```
docker pull --help
```

This help provides the general structure for the command (as for any other). Therefore, the commands used before referenced an image named `ros` that has a tag named `foxy-ros-base` (or `noetic-ros-core`). Because of our approach ros was used, but in general any Linux distro can be used, for example Ubuntu 20.04:

```
docker pull ubuntu:20.04
```

Thus, a third image is now available. Images downloaded and available on the local machine can be run with the `docker run` command, which must be fed with the image name. When a ROS image is executed, it becomes available for use.

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

As an example, let's run one of the downloaded and available images:

```
docker run -it --name my_ros ros:foxy-ros-base bash
```

This command uses some run options, `-it` indicates to run on an interactive terminal inside the container, `-name` assigns a name to the container where the image will be executed, and the `bash` option indicates to run on a terminal.

The container with the image can be seen as a normal computer as well as the local machine where it runs. There it can run pretty much all the commands you run on our own computer (host), but everything is still isolated from the host. It should be able to create own ros2 workspaces, and put code there, and that would not affect the files on the host. You should be able to create your ros2 workspaces and put code there, which would not affect your files. For this,

Figure 1

Pulling a Docker image.

```
$ docker pull ros:foxy-ros-base
foxy-ros-base: Pulling from library/ros
675920708c8b: Pull complete
7cf19bc0e0d1: Pull complete
ff483ca449e2: Pull complete
077f60b8d79e: Extracting [==================================================>]     227B/227B
c152c32a5dfa: Download complete
f6e812731a44: Downloading [======>                                            ]  16.15MB/120.1MB
430ce07703bc: Download complete
1890f880d76a: Downloading [>                                                  ]   1.08MB/73.32MB
6d1250a90908: Waiting
67e10b955d67: Waiting
7586ba8a5351: Pulling fs layer
```

inside the container we will create the folder `/home/user`, with a folder inside it called `ros2_test` that will become our workspace. At the end we compile. The following commands do these tasks.

```
mkdir -pv /home/user/ros2_test/src
cd /home/user/ros2_test
source /opt/ros/foxy/setup.bash
colcon build
```

As it is obvious, it is possible to run multiple containers on the same local machine, the running containers can be listed as follows in the terminal (from outside of them, of course, for example on another terminal):

```
docker ps
```

If the containers have not been stopped, the container information will be displayed detailing the image used, the total execution time, its status, and the name assigned when it was executed (`my_ros` in our case). To exit a container, use the key combination `CTRL+D`. When a container is terminated, its activity will no longer be logged with the `docker ps` command. However, an error will appear if you try to rerun it with the same name, indicating that it already exists and is in use. This is because the container was stopped, but the process still exists, so it would be necessary to kill it. To see all the containers independent of their status, the following command is used:

```
docker ps -a
```

The following command is used to remove a container from memory:

```
docker rm my_ros
```

For a complete list of available commands use the command:

```
docker --help
```

The docker images we pulled previously came from https://hub.docker.com/. Docker Hub is a service provided by Docker for finding and sharing container images (cloud-based registry). If there is a popular software you like, it is very likely that you will find its docker image on Docker Hub. How to create your own Docker image, and how to make it publicly available, will be detailed later.

Docker Network and Docker Compose

One important aspect of running applications in containers is networking. Docker provides several options for connecting containers and to the outside world. In this article, we'll explain the basics of Docker networking and how to use Docker Compose to easily launch multi-container applications.

Docker networking basics

In Docker, a network is a group of interconnected containers. By default, each container in Docker is given its own network stack, with its own IP address. This allows containers to communicate with each other and the outside world, as long as they are on the same network. Several different types of networks can be created in Docker:

- `bridge`: This is the default network type in Docker. It creates a private network for containers, which is isolated from the host and other networks. Containers on the same bridge network can communicate with each other using their IP addresses.
- `host`: This network type bypasses the network isolation provided by the Docker daemon, and instead uses the host's networking stack. Containers on the host network can communicate with each other and the host using their IP addresses.
- `none`: This network type disables networking for a container.

- `overlay`: This network type enables containers to communicate across multiple Docker daemons, allowing containers to span multiple hosts.

In addition to these network types, Docker also provides some additional networking features, such as `aliases`, `links`, and `networks`.

Launching multi-Container applications with Docker Compose

While it's possible to launch multi-container applications using the `docker run` command, it can be cumbersome to specify all of the necessary configuration in the command line. That's where Docker Compose comes in.

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to use a YAML file to specify the details of your application's containers, networks, and volumes. Then, with a single command, you can create and start all of the containers specified in the Compose file.

A simple example of a Docker Compose file is as follows:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/code
  redis:
    image: redis:alpine
```

This Compose file defines a single service, `web`, which is built from the current directory and exposes port 8000. It also defines a `redis` service, which is run from the `redis:alpine` image.

To launch the containers defined in this Compose file, you can use the `docker-compose up` command. This will build the `web` service if necessary, and start both the `web` and `redis` containers.

You can also use Compose to run commands in your containers. For example, to run a command in the `web` container, you can use the `docker-compose run` command:

```
docker-compose run web python manage.py migrate
```

This will start the `web` container if it's not already running, and then run the `python manage.py migrate` command inside the container.

Understanding Docker Compose

To understand Docker Compose, it's important to understand the key concepts it uses:

- Services: A service is a container that is running as part of a multi-container application. Each service is defined in the Compose file with a unique name and can be configured with various options, such as the image to use, port mappings, and environment variables.
- Networks: In Docker Compose, you can create custom networks for your services to connect to. This allows you to specify which services can communicate with each other, and how they can communicate.
- Volumes: Volumes are persistent storage for your containers. You can use volumes to share data between containers, or to persist data outside of a single container's lifecycle.
- Compose file: The Compose file is a YAML file that defines the details of your multi-container application, including the services, networks, and volumes.

With these concepts in mind, let's take a closer look at the structure of a Docker Compose file. The top-level keys in a Compose file are `version`, `services`, `networks`, and `volumes`.

- `version` specifies the version of the Compose file format. This is used to ensure that the file is compatible with the version of Docker Compose you are using.
- `services` defines the containers that make up your application. Each service is given a unique name and can be configured with various options, such as the image to use, port mappings, and environment variables.
- `networks` allows you to create custom networks for your services to connect to. You can specify which services should be connected to which networks, and configure the network's driver and other options.
- `volumes` allows you to define named volumes that can be shared between containers, or used to persist data outside of a single container's lifecycle.

Here's an example of a more complete Docker Compose file that includes all of these top-level keys:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/code
      - logvolume:/var/log
  redis:
    image: redis:alpine
```

```
    volumes:
      - redis-data:/data
networks:
  default:
    driver: bridge
volumes:
  logvolume:
  redis-data:
```

This Compose file defines a `web` service that is built from the current directory and exposes port 8000, and a `redis` service that is run from the `redis:alpine` image. It also defines two named volumes, `logvolume` and `redis-data`, which are used to persist data for the `web` and `redis` services respectively. Finally, it creates a default network using the `bridge` driver and connects the `web` and `redis` services to it.

With this Compose file, you can use the `docker-compose up` command to launch all of the containers in your multi-container application. You can also use the `docker-compose run` command to execute commands in your containers, or the `docker-compose down` command to stop and remove the containers. Docker Compose is a powerful tool for developing and deploying multi-container applications. By using a Compose file, you can easily launch and manage your containers with a single command.

Docker with ROS

Now let's look at a demonstration of how to use ROS with Docker. In a terminal (we'll call it terminal window 1) we first download and run ROS Noetic:

```
docker pull ros:noetic-ros-core
```

Then we execute, in the same terminal 1, roscore (Fig. 2):

```
docker run -it ros:noetic-ros-core roscore
```

In the previous command we did not specify a name for the container, so Docker will assign one to it. And now, let's try to access from another terminal (new terminal window, which we will call terminal 2) to the bash of the container, and run the `rostopic list` command there:

```
docker run -it ros:noetic-ros-core bash
```

And then the command:

```
rostopic list
```

However this produces an error, indicating that communication is impossible, this indicates that each terminal is running in a different container (you can verify in another terminal that each container has a different name, but both use the same image), and it is not possible for them to communicate with each other. They could communicate if the containers share the same network as the host. To do this, let's start again by exiting the two containers loaded with ROS Noetic (press `CTRL+C` in terminal 1, and `CTRL+D` in the terminal 2 window), and then run the following in terminal 1:

```
docker run -it --net=host ros:noetic-ros-core roscore
```

And similarly in terminal 2:

```
docker run -it --net=host ros:noetic-ros-core bash
```

And then the command:

```
rostopic list
```

This time it does produce the expected response (see the topics, Fig. 3).

Now, let's start a publisher node in ROS2. In terminal 1 we press the `CTRL+C` keys and then type:

```
docker run -it ros:foxy-ros-base bash
```

And then, in the same terminal 1, we will publish a topic:

```
ros2 topic pub /test std_msgs/msg/String "data: Hello there!"
```

This makes that in the terminal this publication is seen in a cyclical way. Now in the terminal 2 we press the keys `CTRL+D` and we access in a new container:

```
docker run -it ros:foxy-ros-base bash
```

And there we listen with the following command (Fig. 4):

```
ros2 topic list
```

In fact, in terminal 2 we can subscribe to the topic that is being published, and listen to it cyclically:

```
ros2 topic echo /test
```

To stop and terminate the two containers, press the key combination `CTRL+C` and `CTRL+D` on terminals 1 and 2. The containers can be removed with the `docker rm` command.

Project Example

This is an example of how to use Docker with the Robot Operating System (ROS). This example assumes that Docker is already installed on the system.

1. Create a directory for your ROS application. This will be the base directory for your Docker project.

Figure 2

Roscore launching.

```
$ sudo docker run -it ros:noetic-ros-core roscore
... logging to /root/.ros/log/44848c48-3d27-11ed-9f40-0242ac110002/roslaunch-600f209c9c78-1.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://600f209c9c78:40693/
ros_comm version 1.15.14


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.14

NODES

auto-starting new master
process[master]: started with pid [32]
ROS_MASTER_URI=http://600f209c9c78:11311/

setting /run_id to 44848c48-3d27-11ed-9f40-0242ac110002
process[rosout-1]: started with pid [42]
started core service [/rosout]
```

Figure 3

Topics listened through the host network.

```
$ sudo docker run -it --net=host ros:noetic-ros-core bash
[sudo] password for fredymar:
root@mx:/# rostopic list
/rosout
/rosout_agg
root@mx:/#
```

Figure 4

Topics listened in ROS2.

```
[sudo] password for fredymar:
root@e8bbaf4de5e8:/# ros2 topic list
/parameter_events
/rosout
/test
root@e8bbaf4de5e8:/#
```

2. Create a file called `Dockerfile` in the base directory. This file will contain the instructions for building your Docker image.

3. In the `Dockerfile`, specify the base image that you want to use for your ROS application. There are several official ROS images available on Docker Hub, including `ros`, `ros-core`, and `ros-base`. For this example, we will use the `ros-core` image, which includes ROS, the rosconsole and rospy libraries, and other core ROS packages.

4. In the `Dockerfile`, copy the necessary files for your ROS application into the image. For example, you might copy your ROS nodes and launch files into the `/ros_ws/src` directory in the image.

5. In the `Dockerfile`, install any additional dependencies that your ROS application requires. For example, you might install additional ROS packages using the `ROS_PACKAGE_PATH` environment variable.

6. In the `Dockerfile`, specify the command to run when the container is launched. For example, you might specify a command to run a launch file for your ROS application.

7. Build the Docker image using the `docker build` command. For example:

```
docker build -t my_ros_app .
```

8. Run the Docker container using the `docker run` command. For example:

```
docker run --rm -it my_ros_app
```

This will launch the container and run the command specified in the `Dockerfile`.

Alternatively, you can use Docker Compose to launch multiple containers for a single ROS application. To do this,

you will need to create a `docker-compose.yml` file in the base directory of your project. This file will specify the different containers and their configurations, as well as any networks and volumes that they should be connected to.

Here is an example `docker-compose.yml` file for a ROS application:

```
version: '3'
services:
  ros_master:
    image: ros:melodic-ros-core
    hostname: ros_master
    networks:
      - ros_network
  ros_node:
    image: my_ros_app
    hostname: ros_node
    networks:
      - ros_network
    environment:
      - ROS_MASTER_URI=http://ros_master:11311
      - ROS_HOSTNAME=ros_node
networks:
  ros_network:
    driver: bridge
```

This Compose file defines two services, `ros_master` and `ros_node`, which are connected to a custom `ros_network` using the `bridge` driver. The `ros_master` service is based on the `ros:melodic-ros-core` image and is given the hostname `ros_master`. The `ros_node` service is based on the `my_ros_app` image that we built earlier and is given the hostname `ros_node`. The `ros_node` service also specifies the `ROS_MASTER_URI` and `ROS_HOSTNAME` environment variables, which are used to configure the ROS network. The `ROS_MASTER_URI` variable specifies the URL of the ROS master node, which is running on the `ros_master` container. The `ROS_HOSTNAME` variable specifies the hostname of the `ros_node` container.

To launch the containers defined in this Compose file, you can use the `docker-compose up` command:

```
docker-compose up
```

This will start the `ros_master` and `ros_node` containers and connect them to the `ros_network`. The `ros_master` container will function as the ROS master node, and the `ros_node` container will function as a ROS node that is connected to the master node.

You can also use the `docker-compose run` command to execute commands in the `ros_node` container. For example, to run a ROS node in the `ros_node` container, you can use the following command:

```
docker-compose run ros_node rosrun my_ros_package my_ros_node
```

This will start the `ros_node` container if it's not already running, and then run the `rosrun my_ros_package my_ros_node` command inside the container.

Conclusion

Docker is a powerful tool for running and managing applications in containers. Containers allow developers to package up an application with all of its dependencies and ship it as a single package, making it easier to deploy and manage applications in different environments.

One important aspect of running applications in containers is networking. Docker provides several options for connecting containers and to the outside world, including bridge, host, and overlay networks.

Docker Compose is a tool for defining and running multi-container Docker applications. It allows developers to use a YAML file to specify the details of their application's containers, networks, and volumes, and then launch all of the containers with a single command.

In the field of robotics, Docker and Docker Compose can be especially useful for developing and deploying applications on the Robot Operating System (ROS). ROS is a powerful open-source framework for building robot applications, but it can be challenging to install and manage all of the necessary dependencies. By using Docker and Docker Compose, developers can package up their ROS applications and all of their dependencies into a single container, making it easier to deploy and manage their applications on different platforms.

Additionally, Docker Compose can be used to launch multiple containers for a single ROS application, making it easier to manage the different components of the application. For example, a ROS application might consist of a main control node, a sensor node, and a visualization node, each of which can be run in its container. By using Docker Compose, these containers can be launched and managed with a single command.

Overall, Docker and Docker Compose are powerful tools for developing and deploying applications, including those built with ROS. By using these tools, developers can package up their applications and dependencies into containers, making it easier to deploy and manage their applications on different platforms.

References

Akhilesh, N. S., Aniruddha, M. N., Ghosh, A., & Sindhu, K. (2021). A system to create automated development environments using docker. In *Innovations in computer science and engineering* (pp. 555–563). Springer Singapore. https://doi.org/10.1007/978-981-33-4543-0_59

Baltes, S., & Diehl, S. (2018). Towards a theory of software development expertise. *arXiv*, 1–14. https://doi.org/10.1145/3236024.3236061

Chouhan, D., Gautam, N., Purohit, G., & Bhdada, R. (2021). Implementation of docker for mobile edge computing embedded platform. *WEENTECH Proceedings in Energy*, 440–454. https://doi.org/10.32438/wpe.402021

Damon, S. M., Boyd, B. D., Plassard, A. J., Taylor, W., & Landman, B. A. (2017). DAX - the next generation: Towards one million processes on commodity hardware. In T. S. Cook & J. Zhang (Eds.), *Medical imaging 2017: Imaging informatics for healthcare, research, and applications*. SPIE. https://doi.org/10.1117/12.2254371

Diekmann, C., Naab, J., Korsten, A., & Carle, G. (2019). Agile network access control in the container age. *IEEE Transactions on Network and Service Management (2018)*, 1–14. https://doi.org/10.1109/TNSM.2018.2889009

East, K. W., Leith, A., Ragavendran, A., Delaglio, F., & Lisi, G. P. (2019). NMRdock: Lightweight and modular NMR processing. *Biorxiv*. https://doi.org/10.1101/679688

Feng, Y., Chen, L., Zheng, A., Gao, C., & Zheng, Z. (2019). AC-net: Assessing the consistency of description and permission in android apps. *IEEE Access*, *7*(2019), 57829–57842. https://doi.org/10.1109/access.2019.2912210

Han, Z., Li, X., Xu, G., Xiong, N., Merlo, E., & Stroulia, E. (2020). An effective evolutionary analysis scheme for industrial software access control models. *IEEE Transactions on Industrial Informatics*, *16*(2), 1024–1034. https://doi.org/10.1109/tii.2019.2925422

Immaculate, S., Begam, M., & Floramary, M. (2019). Software bug prediction using supervised machine learning algorithms. *2019 International Conference on Data Science and Communication (IconDSC)*. https://doi.org/10.1109/icondsc.2019.8816965

Kelley, B., Prevost, J. J., Rad, P., & Fatima, A. (2016). Securing cloud containers using quantum networking channels. *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. https://doi.org/10.1109/smartcloud.2016.58

Martínez, F. (2021). Turtlebot3 robot operation for navigation applications using ROS. *Tekhnê*, *18*(2), 19–24.

Martínez, F., Rendón, A., & Arbulú, M. (2018). An algorithm based on the bacterial swarm and its application in autonomous navigation problems. In *Lecture notes in computer science* (pp. 304–313). Springer International Publishing. https://doi.org/10.1007/978-3-319-93815-8_30

Moreno, A., Páez, D., & Martínez, F. (2022). An E2ED-based approach to custom robot navigation and localization. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *13*(6), 910–916.

Naik, N. (2016a). Building a virtual system of systems using docker swarm in multiple clouds. *2016 IEEE International Symposium on Systems Engineering (ISSE)*. https://doi.org/10.1109/syseng.2016.7753148

Naik, N. (2016b). Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems. *2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*. https://doi.org/10.1109/mesoca.2016.9

Narayanamoorthy, A., Li, R., & Huang, Z. (2015). Creating ROS launch files using a visual programming interface. *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*. https://doi.org/10.1109/iccis.2015.7274563

Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G., & Sinnott, R. (2019). A performance comparison of cloud-based container orchestration tools. *2019 IEEE International Conference on Big Knowledge (ICBK)*. https://doi.org/10.1109/icbk.2019.00033

Rastogi, V., Davidson, D., Carli, L. D., Jha, S., & McDaniel, P. (2017). Cimplifier: Automatically debloating containers. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. https://doi.org/10.1145/3106237.3106271

Stoyanov, R., & Kollingbaum, M. J. (2018). Efficient live migration of linux containers. In *Lecture notes in computer science* (pp. 184–193). Springer International Publishing. https://doi.org/10.1007/978-3-030-02465-9_13

Trapti Gupta and Abhishek Dwivedi. (2017). Data storage & load balancing in cloud computing using container clustering. *International Journal Of Engineering Sciences And Research Technology*, *6*(9), 656–666. https://doi.org/10.5281/ZENODO.996046