Canonical

Robotics

Canonical

Contents

1. Tutorials

Step-by-step tutorials for a hands-on introduction to Canonical's robotics solutions.

These tutorials make as few assumptions as possible and are accessible to anyone. They are a good place to start learning about a specific solution from scratch.

1.1. From zero to hero: deploy a robot with snaps and Ubuntu Core

This series of tutorials introduce you to using **snaps and Ubuntu Core for deploying ROS-based robots**.

Starting from a fresh install, these guides take you through an initial ROS 'Hello World' example, tackle packaging of a complete robot using snaps, and the creation of a custom Ubuntu Core image for that robot—all while exploring design and distribution concerns.

1.1.1. From zero to hero: deploy a robot with snaps and Ubuntu Core

This series of tutorials introduce you to using **snaps and Ubuntu Core for deploying ROS-based robots**.

Starting from a fresh install, these guides take you through an initial ROS 'Hello World' example, tackle packaging of a complete robot using snaps, and the creation of a custom Ubuntu Core image for that robot—all while exploring design and distribution concerns.

Tutorial 1: Packaging our first ROS application as a snap

Robotics developers know app development inside-out, but deploying a robotics application can be challenging. It's not uncommon to compile the code on robots, copy/paste compiled packages and end up with unknown versions of software. Even worse, one can experience the infamous "It works on my machine" syndrome.

Robotics software should benefit from a controlled and stable environment, with the same portability and reliability as any other software application. Achieving this reliability should be simple, even if our software relies on hundreds of dependencies.

What we will learn

This developer guide will take us through the process of packaging our first ROS application as a snap.

Snaps offer a solution to build and distribute containerized robotics applications or any software. It is the de-facto distribution tool for companies deploying software on Ubuntu, including `Microsoft`, `Google`, `Spotify` and more. As such, we will be able to leverage the same tooling and global infrastructure for our application.

Throughout this developer guide, we will cover the basics of snap creation for ROS and ROS 2 applications. By introducing the main concepts behind snaps, we will see how to confine our robotics application and make it installable on dozens of Linux distributions.

Requirements

We will need an up and running Ubuntu 20.04 LTS or similar operating system.

Note:

Ubuntu 20.04 LTS should be the minimum version as it is still under maintenance. The installation could be native or in a VM. If using a container, we must be sure that we can install and run systemd, snapd and snaps.

In addition, we will need basic knowledge about ROS or ROS 2 as well as some basic understanding of the Linux environment (Ubuntu).

This developer guide has been tailored for robotics developers looking for a solution to deploy their robotics software and applications. No previous experience with snaps is necessary.

What is a snap?

Snaps[1] are the perfect solution for software deployment in embedded Linux devices.

Snaps are containers that bundle an application and all its dependencies, offering `roboticists`:

- **A containerized solution**: snaps bundle all our dependencies and assets in one package, making our application installable on dozens of Linux distributions and across distro versions. We won't even have to install anything else on our robots' operating system, no dependencies, not even ROS[2] if we are using it.
- **Strict confinement**: snaps are designed to be secure and isolated[3] from the underlying system and other applications, with dedicated interfaces[4] to access specific resources of the host machine, or of other snaps.
- **CI/CD integration**: the creation of snaps can be integrated into our CI pipeline, making the updates effortless.
- **OTA and delta updates**: snaps can update automatically and `transactionally`[5], making sure the device is never broken.
- **Multi-architecture**: snaps come with a multi-architecture feature[6], allowing us to build our snap package for multiple architectures.

[1] https://snapcraft.io/docs
[2] https://ubuntu.com/robotics/what-is-ros
[3] https://snapcraft.io/docs/snap-confinement
[4] https://snapcraft.io/docs/supported-interfaces
[5] https://snapcraft.io/docs/managing-updates
[6] https://snapcraft.io/docs/architectures

- **Managing updates**: snaps can be updated automatically, or we can control the update[7] options (update hours, update holds, update history). It also comes with multiple release channels[8] for role-based access controls and application versioning.
- **Reduce boot time**: We can configure a snap application as a daemon, so it starts automatically at boot.

What can snaps do for our robotics applications?

Snaps are meant to deploy software that has been developed and tested.

Snaps offer a solution to deploy and distribute our software. It's an alternative package manager (like APT). With snaps, we can manage updates and keep track of the version installed on our robot without ever breaking our installation. Sharing and deploying our application to all our users or all our devices is made easy. With snaps, we can seamlessly run an application on our distro and access the host machine and its resources securely.

As such, a snap is a solution to deploy our robotics applications.

What can't snaps do for our robotics applications?

Snaps are not meant for testing and debugging.

Snaps don't embed our source code. As such, developers can't use snaps to test some fresh code. It's not a distribution mechanism for early debugging sessions. On its own, snaps are not meant for cloud web services deployments or web services applications; i.e. Docker. Snaps are used in environments where secure access to the host machine and resources is crucial.

Snaps were designed for embedded Linux applications, with optimisations for ROS packaging.

Install snapd, Snapcraft and LXD

Snapd

Snapd is a daemon required to download, install and run snaps. Snapd also includes the snap command, used to communicate with snapd.

Installing `snapd` is straightforward in most Linux distributions. `Snapd` comes pre-installed on most Ubuntu flavours. In most cases, `snapd` can be installed with:

```
sudo apt update
sudo apt install snapd
```

For different distributions, we can refer to the documentation to install `snapd`[9].

`Snapd` is also available through a snap. We can get a more recent version of `snapd` with:

[7] https://snapcraft.io/docs/managing-updates
[8] https://snapcraft.io/docs/channels
[9] https://snapcraft.io/docs/installing-snapd

```
sudo snap install snapd
```

After installing `snapd` you should be able to type in a terminal:

```
snap --version
```

A quick look at snap help will show everything that we can do with the snap command

```
snap help
```

Commonly used commands can be classified as follows:

```
        Basics: find, info, install, remove, list
      ...more: refresh, revert, switch, disable, enable, create-cohort
       History: changes, tasks, abort, watch
       Daemons: services, start, stop, restart, logs
   Permissions: connections, interface, connect, disconnect
 Configuration: get, set, unset, wait
   App Aliases: alias, aliases, unalias, prefer
       Account: login, logout, whoami
     Snapshots: saved, save, check-snapshot, restore, forget
        Device: model, reboot, recovery
     ... Other: warnings, okay, known, ack, version
   Development: download, pack, run, try
```

Snapcraft

While `snapd` is used to download and run snaps, `snapcraft` is the tool we need to build snaps.

If we installed Snapcraft as a `.deb` package previously, we will have to uninstall it, the Debian package is no longer updated. To do so, just run: `sudo apt remove snapcraft --purge`

To install Snapcraft simply run:

```
sudo snap install snapcraft --classic
```

The `--classic` flag refers to the confinement. We will address the confinement topic later in this guide.

`Snapcraft` is not only a tool to build snaps, but more generally a developer tool, meaning that we will use it to build, upload and share our snaps.

LXD

`LXD`[10] is the container technology used by snapcraft to isolate our snap build. `LXD` is not only dedicated to snaps. It's a next generation system container and virtual machine manager.

LXD can be installed with a snap and must be configured to be used. To install LXD simply run:

```
sudo snap install lxd
```

[10] https://linuxcontainers.org/lxd/introduction/

Now that LXD is installed, we must configure it. We are going to use a default profile to do so:

```
sudo lxd init --auto
```

We can make sure everything went well by listing the profiles and making sure the default profile is listed:

```
$ lxc profile list
+---------+---------------------+---------+
| NAME    | DESCRIPTION         | USED BY |
+---------+---------------------+---------+
| default | Default LXD profile | 0       |
+---------+---------------------+---------+
```

First ROS 2 snap

Our first snap will be a basic ROS 2 Humble talker-listener. We are going to use `ros2_demos:demo_nodes_cpp`[11]. It contains a talker publishing a message and a listener subscribing to it. Both nodes can be launched with the help of the `talker_listener.launch.py`[12].

Understanding the `snapcraft.yaml` file

First clone the package from GitHub[13]:

```
git clone https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap.git
```

The repository contains a snap folder with a `snapcraft.yaml` file. Snaps are defined in a single YAML file placed in our project. Let's explore our `snapcraft.yaml`[14] and break it down in the next section:

Metadata

The `snapcraft.yaml` file starts with a small amount of human-readable metadata, which usually can be lifted from the GitHub description or project `README.md`. This data is used in the presentation of our app in the Snap Store (for example, see PlotJuggler front page[15]).

```
name: ros2-talker-listener
version: '0.1'
summary: ROS 2 Talker/Listener Example
description: |
    This example launches a ROS 2 talker and listener.
```

The `name` must be unique in the Snap Store in case we later want to publish it. Valid snap names consist of lower-case alphanumeric characters and hyphens. They cannot be all numbers, and they cannot start or end with a hyphen.

[11] https://github.com/ros2/demos/tree/humble/demo_nodes_cpp
[12] https://github.com/ros2/demos/blob/humble/demo_nodes_cpp/launch/topics/talker_listener.launch.py
[13] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap.git
[14] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap/blob/main/snap/snapcraft.yaml
[15] https://snapcraft.io/plotjuggler

This is a declarative `version` of the packaged software and is not linked to the version of the snap software itself. It's also possible to write a script to calculate the version, or to take a tag or commit from a git repository.

The `summary` cannot exceed 79 characters.

- For more information about core versions, please see top-level-metadata[16].

Base

Next in the `YAML` file we will find the `base` keyword. This defines a special kind of snap that provides a run-time environment with a minimal set of libraries that are common to most applications. They're transparent to users, but they need to be considered, and specified, when building a snap. As snap developers, we should consider it, but our users will be able to install the final snap independently of their OS version.

```
base: core22
```

core22[17] is the current standard base for snap building and is based on Ubuntu 22.04 LTS[18]. It is therefore the base for ROS 2 Humble snaps.

- For more information about core versions, please refer to the Snapcraft base documentation[19].

Security model

Following, we will see the `confinement` keyword. Snaps are containerized to ensure predictable application behaviour and to provide greater security. We will *review this topic* (page 11) later in this guide.

To get started, we won't confine this application. Unconfined applications are specified with `devmode`.

```
confinement: devmode
```

Parts

`Parts` follow next. They define how to build our app and can be anything: programs, libraries, or other assets needed to create and run our application. Their source can be local directories, remote git repositories, or tarballs. Multiple `parts` can be defined within the same `snapcraft.yaml` in order to build dependencies or even an additional application.

In this example, we have a single part: '`ros-demos`'.

```
parts:
  ros-demos:
    plugin: colcon
```

(continues on next page)

[16] https://snapcraft.io/docs/snapcraft-yaml-schema
[17] https://snapcraft.io/core22
[18] http://releases.ubuntu.com/22.04/
[19] https://snapcraft.io/docs/base-snaps

(continued from previous page)

```
    source: https://github.com/ros2/demos.git
    source-branch: humble
    source-subdir: demo_nodes_cpp
    stage-packages: [ros-humble-ros2launch]
```

Snapcraft relies on well known and well established ROS tools such as, in this example, colcon[20]. Plugins[21] allow us to identify such tools.

The packages we're building must have `install` rules, or else Snapcraft won't know which components to place into the snap. We should make sure we install binaries, libraries, header files, launch files, etc. Here, we selected the `humble` branch of ros2-demos GitHub repository[22] as `source-branch`. Since `ros2-demos` contains multiple packages, we select `demo_nodes_cpp` with the `source-subdir` entry.

We notice that `ros-humble-ros2launch` is listed as a `stage-packages`. Stage packages are packages required to run the `part`. Usually this exec dependency is missing from the `package.xml` hence we must specify it. The rest of the dependencies are going to be automatically downloaded with `rosdep` based on the `package.xml`.

- For more information about general parts metadata, see parts-metadata[23].
- For more information about plugins, please refer to the Snapcraft documentation[24].

Apps

After `parts` we find the `apps` keyword. These are the commands and services exposed to end users.

```
apps:
  ros2-talker-listener:
    command: opt/ros/humble/bin/ros2 launch demo_nodes_cpp talker_listener.launch.py
    extensions: [ros2-humble]
```

The entry under apps is the app name that should be exposed to the end users. In our case the app name is `ros2-talker-listener`.

In snap, an application is usually prefixed by the snap name so that the application `my-app` from the snap `my-snap` can be executed by calling `my-snap.my-app`. However, if both the snap and the app are called the same, as is the case in our ROS 2 example, the execution command collapses to avoid the tediousness of writing twice the same words.

As a result, the command `ros2-talker-listener.ros2-talker-listener` simply becomes `ros2-talker-listener`. We will see this when we run the snap.

Multiple apps can be defined within the same `snapcraft.yaml`. Our snap will then expose multiple commands. Then, after the `app` name, we find the `command` entry. This specifies the path to the binary to be run, along with arguments. This is resolved relative to the root of our snap contents (hence there is no '/' before `opt`).

[20] https://snapcraft.io/docs/colcon-plugin
[21] https://snapcraft.io/docs/snapcraft-plugins
[22] https://github.com/ros2/demos/tree/humble
[23] https://snapcraft.io/docs/snapcraft-yaml-schema
[24] https://snapcraft.io/docs/snapcraft-plugins

Finally, the `ros2-humble extension`[25] will set our ROS 2 humble build and runtime environment. This way we don't have to source manually our ROS 2 environment.

- For more information about ROS extensions, please refer to the Snapcraft documentation[26].

Building, installing and running a snap

Now that our `snapcraft.yaml` is ready, we will describe how to build our package. In this section, we will also cover how to install and run the created snap.

Building

The file `snapcraft.yaml` is expected to be found in the `snap/` directory. So the `snapcraft` command is expected to be run at the root of our repository where we can find the `snap/` directory. The `snapcraft` command is going to look for `snap/snapcraft.yaml` and start building our snap.

Snapcraft is building the snap in steps:

1. **pull**: downloads or otherwise retrieves the components needed to build the part.
2. **build**: constructs the part from the previously pulled components. The plugin[27] of a part specifies how it is constructed.
3. **stage**: copies the built components into the staging area. This is the first time all the different parts that make up the snap are actually placed in the same directory.
4. **prime**: copies the staged components into the priming area, to their final locations for the resulting snap. This is very similar to the stage step, but files go into the priming area instead of the staging area. The prime step exists because the staging area might still contain files that are required for the build but not for the snap.

To build our snap, we will run:

```
snapcraft
```

This will take some time, but once it's done we will see:

```
Created snap package ros2-talker-listener_0.1_amd64.snap
```

This `.snap` file is our packaged application.

[25] https://snapcraft.io/docs/ros2-humble-extension
[26] https://snapcraft.io/docs/snapcraft-extensions
[27] https://snapcraft.io/docs/snapcraft-plugins

Installing

If we could install `snapd` on our current distribution, it means that we can install our freshly built snap. We don't need to be running Ubuntu 22.04 LTS to run this ROS 2 Humble snap. We don't even need to install ROS on our host system to install and run the snap.

Snaps bundle all their dependencies as well as their "`core`" which make them host-agnostic.

Since our snap is currently not confined, we will install it with the flag `--devmode`.

```
sudo snap install ros2-talker-listener_0.1_amd64.snap --devmode
```

Running the snap

Now let's run the snap that we just installed.

We can start the snap by running:

```
ros2-talker-listener
```

We will see the talker listener starting to exchange messages. We can then `ctrl-c` it to stop it. Note that we built, installed and ran this ROS application without even installing ROS 2 on our host.

In this example, our snap has only one app, but snap can contain as many applications (commands) as we need. We can easily get info on our installed snap with the snap info command:

```
$ snap info ros2-talker-listener

name:      ros2-talker-listener
summary:   ROS 2 Talker/Listener Example
publisher: -
license:   unset
description: |
  This example launches a ROS 2 talker and listener.
Commands:
  - ros2-talker-listener
refresh-date: today at 10:54 CEST
installed:    0.1 (x1) 64MB devmode
```

We can see all kinds of metadata as well as the commands available from the snap.

Confining our first snap application

Our application was installed in `devmode`. This means that our snap can access every resource from our host system (files, devices, etc.). For security, snaps are meant to be run and distributed as strictly confined applications.

In this section we will explore the confinement types, grades and interfaces available. Then, we will strictly confine our application.

Confinement types

So far, in our `snapcraft.yaml`, we only declared:

```
confinement: devmode
```

Let's have a closer look at the types of confinement:

- **Devmode**

  A special mode for snap creators and developers. A `devmode` snap runs as a strictly confined snap with full access to system resources, and produces debug output to identify unspecified interfaces. Installation requires the `--devmode` command line argument.

- **Classic**

  Allows access to our system's resources in much the same way traditional packages do. To safeguard against abuse, publishing a classic snap requires manual approval[28], and installation requires the `--classic` command line argument. The typical applications allowed with classic confinement are IDEs (VS Code, Qt Creator).

- **Strict**

  Used by the majority of snaps. Strictly confined snaps run in complete isolation, up to a minimal access level that's deemed always safe. Consequently, strictly confined snaps can not access our files, network, processes or any other system resource without requesting specific access via an interface.

In this case, our application should be confined as strict since we want to be able to share it securely. Everything it needs to access can be declared through interfaces. Let's make that change:

```
confinement: strict
```

- For more information about security models, please see choosing security models[29].

Grade

By adding the `grade` keyword, we can declare the quality of our snap. By defining the grade, we can make sure that a development version never goes into a stable channel.

There are only two grades possible:

[28] https://snapcraft.io/docs/reviewing-classic-confinement-snaps
[29] https://snapcraft.io/docs/choosing-a-security-model

- **`Devel`**

  A `devel` snap indicates that this is a development version, and it is not meant to be released on either a stable or candidate channel.

- **Stable**

  The default one. Meant for production grade snaps, so it can later be released to every user.

For this example, let's add the grade keyword and select `stable`:

```
grade: stable
```

Interfaces

Interfaces enable resources from one snap to be shared with another or with the system. An interface consists of a connection between a slot and a plug. The slot is the provider of the interface while the plug is the consumer, and a slot can support multiple plug connections.

The list of available interfaces is available on the online documentation[30]. We can find interfaces to access the home directory, the CANBus, the network etc.

The interfaces to use are declared in the `snapcraft.yaml` for each application. In the online documentation, we will see that some interfaces are listed as "`auto-connect=yes`" and some are not. The auto-connectable interface will connect at the installation of the snap, while the others will have to be connected manually. This resembles the security validation of an app requesting permissions to the user to access some resources.

Which interfaces a snap requires, and provides, is very much dependent on the type of snap and its own requirements.

For our ROS 2 snap, we will need two auto-connect interfaces: `network` to enable network access and `network-bind` to let our snap operate as a network service.

```
plugs: [network, network-bind]
```

- For more information about interfaces, please see the online documentation[31].

Confining and rebuilding our snap

By changing the confinement level and adding the `grade` and `plugs`, in our `snapcraft.yaml`. We will confine our application.

We can find the updated code from the confined branch[32].

To switch to the confined branch:

[30] https://snapcraft.io/docs/supported-interfaces
[31] https://snapcraft.io/docs/supported-interfaces
[32] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap/tree/confined

```
git switch confined
```

Our `snapcraft.yaml` file should have these modifications:

```
-confinement: devmode
+confinement: strict
+grade: stable
apps:
  ros2-talker-listener:
  command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.launch.py
+ plugs: [network, network-bind]
```

Once our changes are done, let's rebuild our snap:

```
snapcraft
```

This time our snap is confined, so we don't need the `--devmode` flag any more. Yet, we will need the `--dangerous` flag, since our snap hasn't been signed by an official store[33].

To install our snap:

```
sudo snap install ros2-talker-listener_0.1_amd64.snap --dangerous
```

Let's check the connections of our freshly confined snap. Run:

```
$ snap connections ros2-talker-listener

Interface    Plug                                Slot          Notes
network      ros2-talker-listener:network        :network      -
network-bind ros2-talker-listener:network-bind :network-bind -
```

In the above output, we can see that our snap is connected to the network and network-bind slot.

If everything looks good, let's run our confined snap:

```
ros2-talker-listener
```

We will now face this log:

```
user@host:~$ [talker-1] 2022-07-13 15:47:10.570 [RTPS_TRANSPORT_SHM Error]
Failed to create segment cbbe40933e75c60a: Permission denied -> Function
compute_per_allocation_extra_size > [listener-2] 2022-07-13 15:47:10.654
[RTPS_TRANSPORT_SHM Error] Failed to create segment 59f8e836a0800439:
Permission denied -> Function compute_per_allocation_extra_size > [talker-1]
2022-07-13 15:47:10.657 [RTPS_MSG_OUT Error] Permission denied -> Function
init > [listener-2] 2022-07-13 15:47:10.657 [RTPS_MSG_OUT Error] Permission
denied -> Function init > [talker-1] [INFO] [1657720031.730359135] [talker]:
Publishing: 'Hello World: 1' > [listener-2] [INFO] [1657720031.730705569]
[listener]: I heard: [Hello World: 1] > [talker-1] [INFO]
[1657720032.730352803] [talker]: Publishing: 'Hello World: 2' > [listener-2]
[INFO] [1657720032.730571444] [listener]: I heard: [Hello World: 2]
```

[33] https://snapcraft.io/docs/releasing-your-app

The error that we see is related to the shared memory transport not being able to create its file[34]. This is expected. We will cover how to manage shared memory in our snaps later. For now, even with the shared memory failing, ROS 2 falls back to network transport (UDP) and messages are properly sent and received.

Apart from our shared memory error message, our snap is now running strictly confined with only access to our network.

Run our snap as a daemon

One of the advantages of using snaps is that they can turn our application into a service (or a daemon)[35] in an incredibly easy way. Once we have turned our application into a service, it can automatically start at boot and end when the machine is shut down. We can also start and stop on demand through socket activation.

A daemon can take different forms, where the first two daemons are the most used forms:

- **simple**: Run for as long as the service is active - this is typically the default option.
- **oneshot**: Run once and exit after completion, notifying systemd.
- **forking**: The configured command calls fork() as part of its start-up, and the parent process is then expected to exit when start-up is complete.
- **notify**: Assumes the command will send a signal to systemd to indicate its running state.

The daemon feature is per command and not per snap. This means that one snap containing multiple commands can have some running as daemon and others running as simple commands.

To turn our talker-publisher into a simple daemon[36], we will add `daemon: simple` in our `snapcraft.yaml`.

```
apps:
  ros2-talker-listener:
    command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.launch.py
+   daemon: simple
    plugs: [network, network-bind]
    extensions: [ros2-humble]
```

That's it. To switch to this version of the file run:

```
git switch daemon
```

Now let's rebuild and install our snap. The build should be quicker as this has been done before.

```
snapcraft
sudo snap install ros2-talker-listener_0.1_amd64.snap --dangerous
```

Let's explore what happens now that we have turned our snap application into a daemon.

First, we get some info about our snap:

[34] https://ubuntu.com/robotics/docs/ros-2-shared-memory-in-snaps
[35] https://snapcraft.io/docs/services-and-daemons
[36] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap/tree/daemon

```
$ snap info ros2-talker-listener

name: ros2-talker-listener
summary: ROS 2 Talker/Listener Example
publisher: -
license: unset
description: |
  This example launches a ROS 2 talker and listener.
services:
  ros2-talker-listener: simple, enabled, active
refresh-date: today at 16:39 CEST
installed: 0.1 (x9) 64MB -
```

Our command is now listed as a service and marked enabled and active. This means that our `talker-listener` is currently running as a daemon.

`Active` means the talker-listener is now running in the background. And enabled means that it will automatically start at boot and restart in case of failure.

Log our service

We can inspect the log of our running service with the `snap` tool. To inspect our snap log run:

```
$ sudo snap logs ros2-talker-listener

ros2-talker-listener.ros2-talker-listener[970635]: [talker-1] [INFO] [1657724117.
996643358] [talker]: Publishing: 'Hello World: 957'
...
ros2-talker-listener.ros2-talker-listener[970635]: [listener-2] [INFO] [1657724121.
996731460] [listener]: I heard: [Hello World: 961]
```

```
:input: sudo snap logs ros2-talker-listener
ros2-talker-listener.ros2-talker-listener[970635]: [talker-1] [INFO] [1657724117.
996643358] [talker]: Publishing: 'Hello World: 957'
...
ros2-talker-listener.ros2-talker-listener[970635]: [listener-2] [INFO] [1657724121.
996731460] [listener]: I heard: [Hello World: 961]
:input: command 2
output 2
```

We can also add the `-f` flag if we want to wait for new lines and print them as they come in.

Snap logs are actually available in the `systemd` journal. Hence, we can log a snap service directly from the `journalctl` command:

```
journalctl -fu snap.ros2-talker-listener.ros2-talker-listener.service
```

(`-f` for follow and `-u` to show the log of our specific unit)

Interact with our service

We saw that our service was enabled and active, we can obviously interact with these states.

In case we want to temporarily stop our service, we can by using running:

```
sudo snap stop ros2-talker-listener
```

We have stopped our service, so it's no longer running in the background. But if we reboot, our service will still start automatically. This is because our service is still enabled.

To disable our service, we can do it with:

```
sudo snap stop --disable ros2-talker-listener
```

Now our service won't start again. We can verify the result of our actions by running the `snap info` command on our snap.

We have seen how to stop/disable our service, but of course we also have the corresponding start/enable command. Please visit the documentation to know more about service-management[37].

Conclusion

In this developer guide, we went through the creation of a basic ROS 2 snap. But in the process, we learned the basics to create our own ROS snap as well. We have covered the basic concepts of a snap, how to build and run them. We also shared the benefits and good development practices. There are more features and advanced development tips that we have yet to cover. The turtlebot3 snap example[38] shows how we can use snap to make your robot software easily installable.

Visit the *robotics documentation* (page 2) to go further. If you have any questions or need help, you can visit and post your question on the Ubuntu robotics forum[39].

[37] https://snapcraft.io/docs/service-management
[38] https://ubuntu.com/blog/how-to-set-up-turtlebot3-in-minutes-with-snaps
[39] https://discourse.ubuntu.com/c/project/robotics/121

Part 1 - exercise: Add additional application to the snap

Important:

This exercise requires having followed the *Tutorial 1: Packaging our first ROS application as a snap* (page 2).

Let's do a little exercise. Right now, our talker-listener only has one application. How about you add another one? Having the ROS 2 topic tools inside our snap would be great for some introspection.

Assignment

Modify the `snapcraft.yaml` file:

- Add a new app entry for ros2topic
- Add the corresponding dependencies

Build and install the new snap.

Outcome

By the end of this exercise, you will have a new command available:

```
ros2-talker-listener.ros2topic hz /chatter
```

- Note that the ROS 2 topic tools are only going to work with message types that are included in the snap.

Solution

Click here for the solution

- Add `ros2topic` as a `stage-package` to have it available at run time:

```
parts:
  ros-demos:
    plugin: colcon
    source: https://github.com/ros2/demos.git
    source-branch: humble
    source-subdir: demo_nodes_cpp
-   stage-packages: [ros-humble-ros2launch]
+   stage-packages: [ros-humble-ros2launch, ros-humble-ros2topic]
```

  - We add our dependency (`ros-humble-ros2topic`) to the `stage-packages` list. `Stage-packages` are packages that are required to run the part. Here we decided to add our dependency to our already existing `ros-demos` part for simplicity. Alternatively, we could have created an additional empty part simply to add our `stage-packages` dependency.

- Add an additional app:


```
apps:
  ros2-talker-listener:
    command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.
launch.py
```

Important:

Before you start

1. Make sure you have completed **Tutorial 1: Packaging our first ROS application as a snap** (page 2) before starting this tutorial. This tutorial builds on key concepts introduced earlier, including the benefits of snaps for ROS applications and the essential features of Snapcraft.
2. This guide is meant for ROS snap beginners and advanced users looking for guidelines.
3. This is not a quick guide nor a ROS tutorial. A new developer will take approximately **one full day** to cover the entire guide.

The first step for deploying a robotics application is to package it. Snaps are specifically designed to confine embedded Linux applications, making it easier to install and manage software versions. They offer useful features for deployment, such as over-the-air (OTA) updates, delta updates, automatic rollback, security policies, and more. While bundling all software dependencies, snaps provide a comprehensive deployment infrastructure for robotics software.

While snaps adapt to the way ROS developers work, they bring new concepts and tools that we need to understand first.

How to use this developer guide

In *Tutorial 1: Packaging our first ROS application as a snap* (page 2) of our developer guide series, we introduced the main concepts and learned what snaps can do for ROS applications. We also explored the most important features of Snapcraft.

In this developer guide, we will explore advanced snap topics and tools that will show you how to structure, package, and test complex robotics applications.

While this developer guide is meant to be read completely, a second-time reader or an experienced snapcraft developer might come back to a specific section. Some sections can be used independently to learn how to solve a specific problem.

What we will learn

Deploying robotics software is usually tackled when a project reaches certain maturity. We need a fast and reliable way to distribute software and subsequent updates. Additionally, we need the deployed software to be fully isolated in its sandbox, thus minimizing security risks.

Addressing those points and more, snaps represent an ideal solution to deploy ROS applications to devices and users.

Snapping a simple talker-listener in *Tutorial 1: Packaging our first ROS application as a snap* (page 2) was good enough to see the potential of snaps. However, when packaging a more complex application, one might wonder:

- Should we build one snap for our entire robot?
- Should we have one part per ROS package?
- What should a final robot snap look like?

Snap and snapcraft contain tons of features that will come in handy when defining our robot application. This guide will help you understand these features and how to use them.

In this second part, we will learn:

- How to create a snap for a robot software stack.
- How to take the decisions that a ROS snap developer has to take to design a snap.
- We will introduce new features and concepts associated with snaps that will be useful in a robot application.
- The different means to debug a snap package and its snapped applications.

While covering the theoretical aspects of ROS snaps, we will apply all this knowledge to TurtleBot3[40] for a more interesting real-world scenario.

Requirements

1. You will need basic ROS knowledge. Ideally, you will already understand how to build packages, call a service, and know what a launch file is. You should also be familiar with the process of creating a map and using it to navigate.
2. A basic understanding of the Linux environment (Ubuntu) is also required.
3. You should have a working knowledge of snaps. This means either previous experience with snap and snapcraft, or having followed the *Tutorial 1: Packaging our first ROS application as a snap* (page 2).

Setup

Since this developer guide builds a snap for a robot, we will need:

- TurtleBot3 Gazebo classic[41] simulation running under ROS Noetic.
- Depending on our current OS, we can choose between the following Native setup and the Multipass setup.

Option 1: Native Setup

Option 2: Multipass

The Native setup will need Ubuntu 20.04[42]. This is because we will need the TurtleBot3 simulation to run. We don't intend to embed the simulation into our snap.

With the running Ubuntu 20.04, follow the next step to install all the dependencies:

- Install ROS Noetic desktop-full following the official documentation[43]

[40] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
[41] https://classic.gazebosim.org/
[42] https://releases.ubuntu.com/focal/
[43] http://wiki.ros.org/noetic/Installation/Ubuntu

- Install the TurtleBot3 simulation with the following command:

```
sudo apt install -y ros-noetic-turtlebot3-gazebo
```

- Make sure snapd[44] and snapcraft[45] are installed

If a native setup is not possible, we will use a Virtual Machine. In this case, we are going to use Multipass[46] to quickly generate an Ubuntu VM.

Install Multipass

- On Ubuntu, the installation of Multipass is straightforward:

```
sudo snap install multipass
```

- We can also install it on Windows and macOS by following the documentation[47].

Launch the VM

- To launch the VM, we are going to use a cloud-init[48] configuration stored in GitHub. In essence, we will launch the VM, and install ROS Noetic along with the TurtleBot3 simulation automatically. Note that this might take some time depending on our configuration (~15 minutes).

```
multipass launch --cpus 2 --disk 20G --memory 4G --cloud-init https://raw.
githubusercontent.com/canonical/ros_snap_workshop_part2_multipass/main/cloud-init.
yaml 20.04 --name workshop-part2 --timeout 10000
```

- The VM must now be up and running. We can verify that with:

```
$ multipass list

Name           State   IPv4       Image

workshop-part2 Running 10.26.1.87 Ubuntu 20.04 LTS
```

Attach to the VM

- Multipass offers a way to attach a shell to the VM with:

```
multipass shell workshop-part2
```

- Unfortunately, with this solution, we cannot forward X11 (for graphic applications).
- We can connect with ssh into the VM with X11 forwarding with the command:

[44] https://snapcraft.io/docs/installing-snapd
[45] https://snapcraft.io/snapcraft
[46] https://multipass.run/docs
[47] https://multipass.run/install
[48] https://canonical-cloud-init.readthedocs-hosted.com/en/latest/

```
ssh -X ubuntu@$(multipass list --format csv | awk -F, '$1=="workshop-part2"{print $3}
')
```

- The password for the Ubuntu user is simply: Ubuntu.

> **Caution:**
>
> Using a VM with a pre-saved password is not a good practice in general. Here we only do so to make it simple to follow the developer guide. If you wish to use a VM for anything else please change the password.*

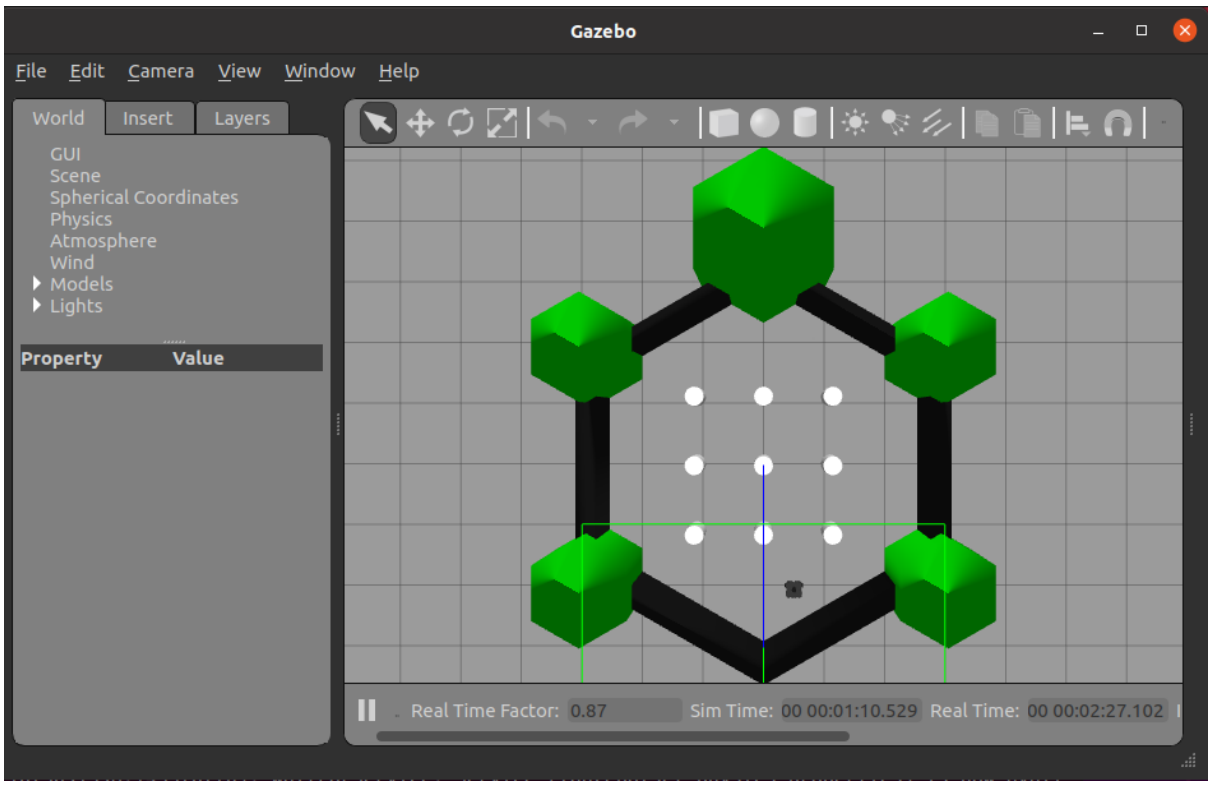
- In the case of a VM setup, all the commands and instructions from this guide must be executed in the VM.

Setup Check

We can make sure that everything is properly installed by launching:

```
TURTLEBOT3_MODEL=waffle_pi roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

We should then see the TurtleBot3 simulated in the `turtlebot3_world` environment:



We can now `ctrl-c` the Gazebo launch file.

We can also:

```
snapcraft --version
```

And it should display a version equal to or above `7.2.9`.

We are now all set up for this developer guide.

Identification of robot components

While we can find many examples to snap various applications, robots might sound more complicated. A robot is usually composed of multiple applications, background services, and applications to call.

Snaps are designed to deploy applications. Thus, **we must think of our robot software as an application and not as a set of launch files** to call in 5 different terminals.

To start snapping our stack we first need to identify the various functionalities and applications of our robot application[49].

We are going to apply the methodology to the TurtleBot3. This way, we will have a clear example of the functionalities and snap entries for a robot, as well as the process to identify these entries.

For this guide, we will focus on the monolithic snap approach since it is the simplest for a first robot snap. All the possible architectures for ROS snap applications are described *in the documentation* (page 216).

Functionalities of our robot snap

Before defining our applications, we must define the final functionalities that the snap should bring to our robot. We want the user to do three things:

1. Teleoperate the robot with a keyboard or a joystick.
2. Create a map of our environment.
3. Navigate into our mapped environment.

These are the three functionalities that we want our users to be able to perform with our robot.

Additionally, we don't want our snap to be compatible only with our physical TurtleBot3. We also want our snap to be compatible with the simulation.

Note that **we won't need the snap to run the simulation itself**, since the simulation is a substitute for the real robot. **The snap should simply interface with the simulation.**

Defining the TurtleBot3 applications

Now that we have defined the functionalities of the robot, let's have a look at the applications we need to cover them.

We want most of it to be ready as quickly as possible on the robot. Hence, the common set of nodes for these functionalities must be always up and running. We will call it the **core** application.

The diagram below summarises all these different applications inside our TurtleBot3c (C for Canonical) snap and how they will be used by the final user. The TurtleBot3C snap contains

[49] https://ubuntu.com/robotics/docs/identify-functionalities-and-applications-of-a-robotics-snap

all the system and ROS dependencies for our applications. We have `core` and `teleop` which are daemons started automatically at boot since they are always necessary. `Joy` and `key` apps are applications that can be called from the terminal by a user to be able to control the robot manually. Finally, `mapping` and `navigation` are background applications that can be enabled or disabled. Once enabled they will act exactly like `core` and simply automatically start at boot. All these applications are communicating with each other with ROS.

https://assets.ubuntu.com/v1/2529db37-tb3snap.png

Let's break down what each of these applications will do and the requirements they have.

Core

Core is going to be our common set of nodes always running in the background. In the case of the TurtleBot3, we are talking about all the nodes to interface with the hardware (mobile base, LIDAR, camera) as well as the `robot_state_publisher` to have the model of the robot available. This `core` snap application must be running in the background.

Teleoperation applications

TurtleBot3 should have the functionality to be teleoperated with a joystick or a keyboard. We want to be able to do that from a controller connected to the robot. We also want another computer on the network to be able to control our robot.

To be able to manage so many different ways of teleoperating the robot, we will need an application to be able to route all this traffic. Similarly to the core app enabling the common part of all functionalities, the `teleop` app will be starting the common part necessary for all teleoperation scenarios.

On the contrary, we don't want the joystick or keyboard specific apps to be running all the time.

We will then need three different snap applications:

- The first one called `teleop` simply runs the teleoperation basics in the background. This way, a controller application running on the robot or remotely can always control the robot.
- The second application will be called `joy` and will be a command to call whenever we want to control the robot with a joystick. Meaning that every node and configuration specific to the joystick control should be launched in the application.
- The third application, `key`, will be similar to the `joy` application, but for the keyboard control.

You can see all of them in the diagram above. This is an example of how we can split our applications. Depending on the use case one might decide for a different split.

With these three different snap applications, we will be able to have all the teleoperation functionality that we wanted.

Mapping

We will use TurtleBot3 to create a map of its environment. Thus, we want a `mapping` application that we can call to start mapping our environment. We also want this application to automatically save the map when we stop the application. This way there will be no need to manipulate any file or to call a service of any kind.

Remember that the idea is to think of creating an application for deployment. The final user of our snap will understand what creating a map of the environment means. On the contrary, our end-user might not know ROS and thus how to save a map the ROS way (calling a `rosservice` from the `map_server`).

Since the `core` & `teleop` daemons will be running, launching `joy` or `key` along with `mapping` will be all we need to start creating a map of any environment.

Navigation

After mapping, we navigate in our freshly mapped environment.

Hence, we will need another application called `navigation`. This application will be launching everything we need for the navigation (localization + navigation). This application should make sure to load the last map that we created (with the `mapping` application). While the mapping is done on rare occasions, the navigation will be rather standard.

Thus, our `navigation` should be running as a daemon in the background. With the specificity that it should be shipped as disabled (we can enable it) since we won't have a map created on the very first install.

TurtleBot3 ROS workspace presentation

We have now seen the different applications that TurtleBot3 snap will have to make its applications easy and error-proof to use.

In the section below we covered the methodology used to define our snap application. **We must think in terms of functionalities once deployed**. The usage of a deployed application can differ from the usage that a developer would have of the same ROS workspace.

It's now time to look at what's inside the TurtleBot3 repositories. This way we will identify how to expand a traditional ROS workspace into an application-oriented workspace.

In this developer guide, we are going to work with three repositories: TurtleBot3, TurtleBot3 messages, and TurtleBot3c.

TurtleBot3

This is the official repository from Robotis[50], which contains most of the TurtleBot3 packages. The main packages for this demo are:

- Turtlebot3_bringup[51]: Contains robot launch files for different configurations.
- Turtlebot3_description[52]: Contains URDF files as well as meshes.
- Turtlebot3_navigation[53]: Contains launch files and parameters for navigation algorithm and ROS packages (AMCL, `move_base`).
- Turtlebot3_slam[54]: Contains launch files and configurations for the different SLAM algorithm (gmapping, hector slam, etc.).
- Turtlebot3_teleop[55]: Containing the node to control TurtleBot3 with a keyboard.

All these ROS packages contain `package.xml`, effectively listing all the build time and runtime dependencies. Recall that these `packages.xml` files are used by `snapcraft` plugins to automatically pull dependencies. If we were to use a different robot stack we should make sure that similarly, the `package.xml` is properly listing the dependencies.

Since we are running ROS Noetic for this guide, we will be using the `noetic-devel` branch along with the corresponding Ubuntu 20.04.

TurtleBot3 messages

Additionally, the `turtlebot3_msgs`[56] repository is another official repository from Robotis and contains the TurtleBot3 messages definition. The messages (`Version`, `Sound`, and `SensorState`) are specific to the TurtleBot3 application.

TurtleBot3c

TurtleBot3c[57] is a collection of configurations/launch files maintained by the robotics team at Canonical created for this developer guide. The purpose of this repository is to easily expose applications-oriented `launchfiles`.

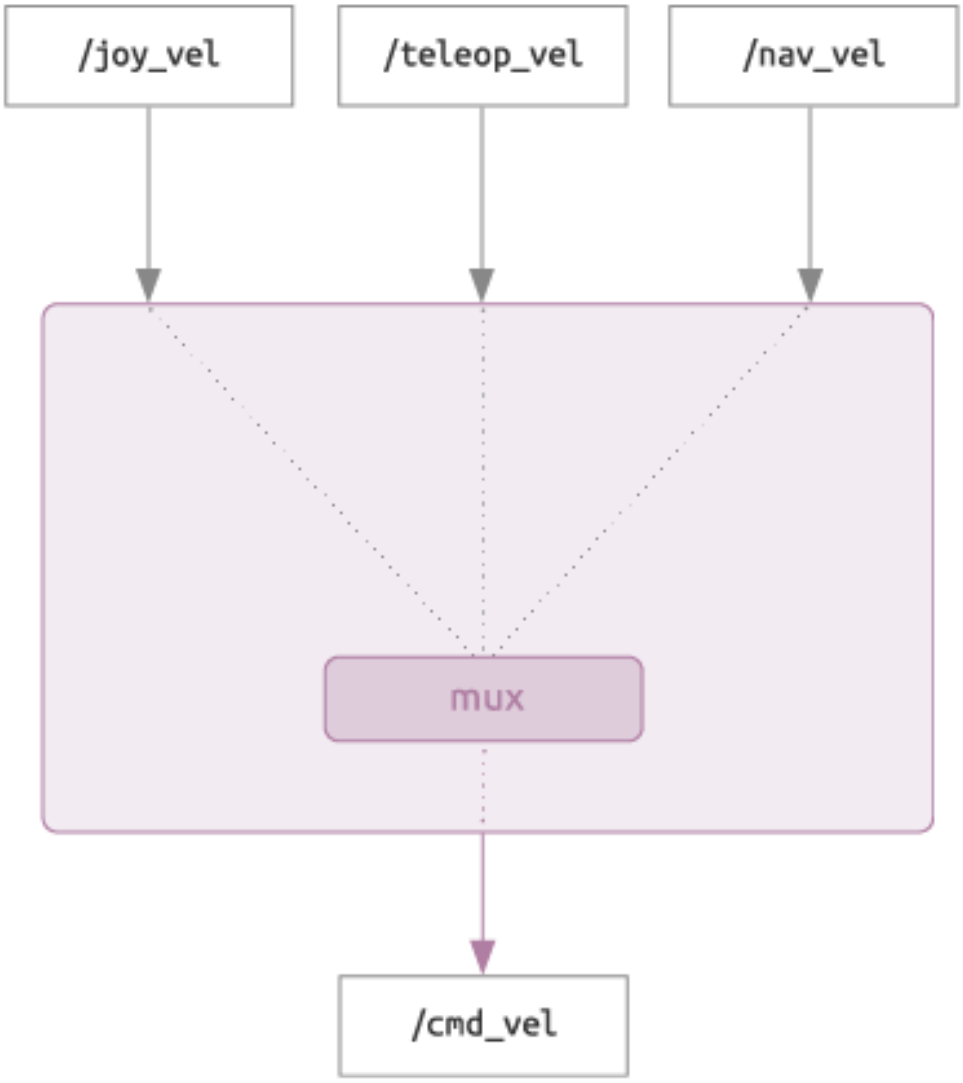
It is composed of four different packages:

- Turtlebot3c[58]: The meta-package of this repository.
- Turtlebot3c_2dnav[59]: Contains one mapping launch file, and one navigation launch file.
- Turtlebot3c_bringup[60]: Contains a general bring-up launch file capable of launching the basic nodes to interface with the real TurtleBot3 or its Gazebo classic simulation.

[50] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic
[51] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic/turtlebot3_bringup
[52] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic/turtlebot3_description
[53] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic/turtlebot3_navigation
[54] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic/turtlebot3_slam
[55] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic/turtlebot3_teleop
[56] https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
[57] https://github.com/canonical/turtlebot3c/tree/noetic-devel
[58] https://github.com/canonical/turtlebot3c/tree/noetic-devel/turtlebot3c
[59] https://github.com/canonical/turtlebot3c/tree/noetic-devel/turtlebot3c_2dnav
[60] https://github.com/canonical/turtlebot3c/tree/noetic-devel/turtlebot3c_bringup

- Turtlebot3c_teleop[61]: Contains launch files as well as configuration for joystick and keyboard teleoperation.

While the first three packages are rather straightforward, the turtlebot3c_teleop[62] package adds logic that is not present in the original TurtleBot3 repository. We indeed use a topic multiplexer called topic_tools/mux[63] responsible for rooting the desired robot control topic to the `/cmd_vel` topic (see figure below). This way our TurtleBot3 could be controlled by a joystick, a keyboard, or even the navigation depending on what we desire without interfering with each other.



Selecting the topic which is going to be redirected, can be done with the following command:

[61] https://github.com/canonical/turtlebot3c/tree/noetic-devel/turtlebot3c_teleop
[62] https://github.com/canonical/turtlebot3c/tree/noetic-devel/turtlebot3c_teleop
[63] https://wiki.ros.org/topic_tools/mux

```
rosservice call /mux/select "topic: 'key_vel'"
```

Snapping our software

In the previous section, we have defined the applications that our snap must expose. We have also seen the different repositories from TurtleBot3 that we will need to implement those applications. This is the first step to writing our `snapcraft.yaml` file.

The following section is going to cover how to snap applications, step by step. The final version of the snap workspace is available on GitHub[64]. We can refer to it at any time in case we have questions regarding the content of a file.

Completing our snapcraft.yaml

Let's create a directory for our project:

```
mkdir -p turtlebot3c_snap/

cd turtlebot3c_snap
```

And let's initialise our snap project:

```
snapcraft init
```

This will create a `snap/snapcraft.yaml` file.

In this file, we first need to complete the metadata along with `strict` confinement[65] and `core20`[66].

```
name: turtlebot3c
base: core20
version: '1'
summary: Turtlebot3c core snap
description: |
  This snap automatically spawns a roscore and the core components for the
  Turtlebot3.
confinement: strict
```

Everything here is self-explanatory. We start directly from `strict` confinement since we already know that `network` and `network` bind are the necessary plugs for ROS.

[64] https://github.com/canonical/turtlebot3c-snap/tree/noetic-devel
[65] https://snapcraft.io/docs/snap-confinement
[66] https://snapcraft.io/docs/base-snaps

Build the workspace

As mentioned in the previous section, we will use three different repositories. turtlebot3[67] and turtlebot3c[68] but also `turtlebot3_msgs`[69] since TurtleBot3 messages are hosted on a different repository.

Here we will define one snapcraft part[70] for our ROS workspace. For this snapcraft part, we will use the `catkin` plugin[71]. As seen in the previous part of this guide[72], the `catkin` plugin comes in handy with the `ros-noetic` snapcraft extensions[73].

The source is used to retrieve the source tree to build. Here we have three different GitHub repositories (turtlebot3[74], turtlebot3c[75] and `turtlebot3_msgs`[76]). We cannot put all three in the source entry. We will use Vcstool[77] to pull all the repositories that we need based on a rosinstall file[78].

A detailed explanation of this process can be found in the Vcstool and rosinstall file documentation[79].

Writing our turtlebot3 part

The very first thing that we need is a `.rosinstall` file. We will populate this file with our three repositories:

turtlebot3c.rosinstall:

```
- git: {local-name: turtlebot3, uri: 'https://github.com/ROBOTIS-GIT/turtlebot3.git',
version: noetic-devel}
- git: {local-name: turtlebot3_msgs, uri: 'https://github.com/ROBOTIS-GIT/turtlebot3_msgs.
git', version: noetic-devel}
- git: {local-name: turtlebot3c, uri: 'https://github.com/ubuntu-robotics/turtlebot3c.git
', version: noetic-devel}
```

We will place this file at the root of our `turtlebot3c_snap` directory. Right next to the `snap/` folder.

Now we must write the `snapcraft` part to lever this file with the Vcstool. We will need `python3-vcstool` at build time to use our `rosinstall` file.

Our part to build ROS workspace is going to be:

```
parts:
  workspace:
```

(continues on next page)

[67] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic
[68] https://github.com/canonical/turtlebot3c
[69] https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
[70] https://snapcraft.io/docs/snapcraft-yaml-schema
[71] https://snapcraft.io/docs/catkin-plugin
[72] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[73] https://snapcraft.io/docs/ros-noetic
[74] https://github.com/ROBOTIS-GIT/turtlebot3/tree/noetic
[75] https://github.com/canonical/turtlebot3c
[76] https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
[77] https://github.com/dirk-thomas/vcstool
[78] https://docs.ros.org/en/independent/api/rosinstall/html/rosinstall_file_format.html
[79] https://ubuntu.com/robotics/docs/vcstool-and-rosinstall-file

(continued from previous page)

```
    plugin: catkin
    source: . # import our rosinstall file
    build-packages: [python3-vcstool, git]
    stage-packages:
      - ros-noetic-rosbash # necessary if we need rosrun
      - ros-noetic-roslaunch # necessary if we need roslaunch
    override-pull: |
      snapcraftctl pull
      # Here we are going to use the local .rosinstall file
      vcs import --input turtlebot3c.rosinstall
```

Here, the `catkin` plugin[80] is going to use the `package.xml` present in the different source packages to manage the ROS dependencies. It will also build and install our packages.

Before running any kind of build, let's declare our applications.

Building the core app

Above, we defined that we needed a `core` daemon responsible to launch the motor controller, advertising the sensors, uploading the robot model to the `rosparam` server and publishing the robot `tf` tree.

In the TurtleBot3c ROS world this corresponds to the launch file `turtlebot3c_bringup.launch`[81] inside the `turtlebot3_bringup` package (installed by the part we just added). This application must be running in the background, we will thus declare it as a `daemon`. Remember that since it's a ROS application we will need a ROS environment. So we will use the `ros-noetic` snapcraft extensions[82].

In our case, we will define a `core` application that will work with the TurtleBot3 waffle_pi model[83]. In the launch file this is done by defining an environment variable.

The robot is managing all its hardware interfaces through USB. By default, our snap won't have access to it. We then have to declare the usb-raw plug[84].

Let's add this app to our `snapcraft.yaml`:

```
apps:
  core:
    daemon: simple
    environment:
      TURTLEBOT3_MODEL: waffle_pi
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
    plugs: [network, network-bind, raw-usb]
    extensions: [ros1-noetic]
```

We have introduced a new entry to our app called `environment`. This way we can define environment variables that are specific to our application. We can find all the apps and services metadata in the documentation[85].

[80] https://snapcraft.io/docs/catkin-plugin
[81] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_bringup/launch/turtlebot3c_bringup.launch
[82] https://snapcraft.io/docs/ros-noetic
[83] https://emanual.robotis.com/docs/en/platform/turtlebot3/features/
[84] https://snapcraft.io/docs/raw-usb-interface
[85] https://snapcraft.io/docs/snapcraft-yaml-schema

Building the teleop app

Along with `core`, the `teleop` should be declared as a daemon too since we want it to be running in the background. The `teleop` app will be running our teleoperation multiplexer. This corresponds to the launch file `turtlebot3c_teleop.launch`[86] in the `turtlebot3c_teleop` package.

We can declare this application by adding the following to our `snapcraft.yaml`:

```
apps:
  [...]
  teleop:
    daemon: simple
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_teleop turtlebot3c_teleop.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

This application is highly similar to the `core` app. Nothing to declare.

Building the key app

Using the above `teleop` application and adding a `key` application would let us effectively teleoperate the TurtleBot3. Here the `key` application will be a command to call.

This app must publish a `/key_vel` topic based on the keyboard input. This corresponds to the `key.launch`[87] from the `turtlebot3c_teleop` package. Snaps are by default accessing the keyboard, so we won't need any special `plugs` here.

We then add the following to our `snapcraft.yaml`:

```
apps:
  [...]
  key:
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_teleop key.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

Again this is very straightforward with what we have already done.

With this new application, our snap will now make a command available called `turtlebot3c.key`.

[86] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_teleop/launch/turtlebot3c_teleop.launch

[87] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_teleop/launch/key.launch

Building the joy app

Very similar in terms of usage to key, but we will use a different launch file for the joystick. It's `joy.launch`[88] from the `turtlebot3c_teleop` package. The major differences are that this is going to be published on the `/joy_vel` topic and that our application will need access to the joystick. We must use the `joystick` interface[89], so our confined application will be able to access the joystick device from our host.

We then add the following to our `snapcraft.yaml`:

```
apps:
  [...]
  joy:
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_teleop joy.launch
    plugs: [network, network-bind, joystick]
    extensions: [ros1-noetic]
```

With this new application, our snap will now make a command available called `turtlebot3c.joy`.

Now we have all the applications necessary for teleoperation whether we want to use a keyboard or a joystick.

Building the mapping app

For the mapping, we will need a command that we can call whenever we want to start doing a mapping. This command `mapping` is going to rely on the `turtlebot3c_mapping.launch`[90] file from the `turtlebot3c_2dnav` repository. This is also straightforward.

We add the following to our `snapcraft.yaml`:

```
apps:
  [...]
  mapping:
    environment:
      TURTLEBOT3_MODEL: waffle_pi
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

With this new application, our snap will now make a command available called `turtlebot3c.mapping`.

Note that the `turtlebot3c_mapping.launch`[91] accepts ROS arguments to select a different SLAM algorithm. This is nicely compatible with our snap application, and we could imagine calling our application with the argument:

```
turtlebot3c.mapping slam_methods:=hector
```

[88] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_teleop/launch/joy.launch
[89] https://snapcraft.io/docs/joystick-interface
[90] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_2dnav/launch/turtlebot3c_mapping.launch
[91] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_2dnav/launch/turtlebot3c_mapping.launch

Building the navigation app

Finally, our last application, `navigation`. `navigation` will also be a daemon running in the background but with one specificity. It's not going to be enabled at first. The reason is that when we first install our snap, we won't have created any map. This way we can create our first map and then activate our navigation daemon. Daemons can be installed in a disabled state thanks to the additional app entry: `install-mode: disable`. Once enabled it will remain enabled even after an update.

The whole navigation stack relies on `turtlebot3c_navigation.launch`[92] from the package `turtlebot3c_2dnav`. It will launch the localization as well as the navigation.

Let's add the following to our `snapcraft.yaml`:

```
apps:
  [...]
  navigation:
    environment:
      TURTLEBOT3_MODEL: waffle_pi
    daemon: simple
    install-mode: disable
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

Once our snap is installed, we will be able to enable this daemon with the command:

```
sudo snap start --enable turtlebot3c.navigation
```

Application scripts

So far we have defined our part as well as our different applications in our snap. Nothing there was new. It was simply an application of the basic ROS/snap knowledge to the TurtleBot3 case.

Let's say that we want our snap `navigation` daemon to always use the latest recorded map. Or that we want the mapping to automatically save the map without having to manually call a ROS service to save the map.

For these, we can use additional features of `snap` and `snapcraft`. These features will make our applications much more integrated and easier for the final user. The features that we will describing below are:

- Environment variables.
- Local file parts.
- Launcher scripts.
- Daemon scripts.

[92] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_2dnav/launch/turtlebot3c_navigation.launch

Snap environment variables

Environment variables are widely used across Linux to provide convenient access to system and application properties. Both `snapcraft` and `snapd` consume, set, and pass-through specific environment variables to support building and running snaps.

There are snap environment variables, like `$SNAP` and `$SNAP_INSTANCE_NAME` used to identify the snap and its location. The *Snap environment variables*[93] documentation is a great place to learn about these.

Additionally, snaps define environment variables specific to data and file storage. The *Snap data and file storage documentation*[94] is a great place to learn about these.

Local files part

In order to create additional behaviour for a snap, we need to add scripts. These scripts could be for example responsible for calling the ROS node in charge of saving the map.

To import all these local files into our snap we will need an additional `snapcraft` part[95]. They are used to declare pieces of code that will be pulled into your snap package.

So far we only have one part called `workspace` using our `rosinstall` file to build and install our ROS repository. This part is relying on the `catkin` plugin[96].

In order to simply import scripts into our snap we will need a different plugin, the `dump` plugin[97].

As its name suggests, the `dump` plugin is simply used to "dump" the specified sources in a specified location into our snap. Usually, the dumped files are scripts, configuration files, binaries etc.

Adding scripts to our snap's part

In our case we will simply need to dump all the files that we are going to store in `snap/local` into the `usr/bin` of our snap.

We will then need to add the following part to the `snapcraft.yaml`:

```
parts:
  workspace:
    [...]
+ # copy local scripts to the snap usr/bin
+ local-files:
+   plugin: dump
+   source: snap/local/
+   organize:
+     '*.sh': usr/bin/
```

[93] https://ubuntu.com/robotics/docs/snap-environment-variables
[94] https://ubuntu.com/robotics/docs/snap-data-and-file-storage
[95] https://snapcraft.io/docs/snapcraft-yaml-schema
[96] https://snapcraft.io/docs/catkin-plugin
[97] https://snapcraft.io/docs/dump-plugin

The organize keyword is a key/value pair, the key represents the path of a file inside the selected source of the part and the value represents the file's destination in the stage area.

Launcher scripts

Now we have a part to import local files, but we don't have the local files yet.

So far for our `snapcraft` apps, in the `command` entry, we have been writing the full name of the commands. For example, for our `core` application, we wrote:

```
command: /opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
```

Our turtlebot3c_bringup.launch[98] is actually taking a `simulation` argument. It can be set to `false` or `true`, to specify whether we want to run the bring-up for the simulation or not. Since the core application is a daemon, we cannot append anything to the specified command at launch. This means that we must give it directly in the command.

By trying we will quickly realise that we are facing an issue. The character "=" is forbidden in the `snapcraft.yaml`. To specify an argument to a ROS launch file we must use the syntax `arg_name:=arg_value`. For this reason (and actually many others that we will discuss later on) our snapcraft command entry is going to call a script containing a call to our command.

This way, we will not only be able to specify ROS arguments, but we will also be able to potentially perform some simple checks or any kind of behaviour that would not be that easy on a one-liner. For example, verifying that a file exists before calling a command.

Adding scripts' launcher

For our TurtleBot3 snap, we will add a launcher script for `core`, `mapping` and `navigation`.

Let's add these scripts into our snap/local folder.

First, we create the folder:

```
mkdir -p snap/local
```

Then we add the file: *snap/local/core_launcher.sh*:

```
#!/usr/bin/bash

${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
simulation:=true
```

Here we are using the `$SNAP` variable that we saw earlier. This way we are sure of the binary that we are calling. Also, we are assigning the `simulation` argument to `true`. We will see *in the next section* (page 39) how to make it configurable.

Similarly, the next two scripts:

snap/local/mapping_launcher.sh:

[98] https://github.com/canonical/turtlebot3c/blob/noetic-devel/turtlebot3c_bringup/launch/turtlebot3c_bringup.launch#L21

```
#!/usr/bin/bash

${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
```

snap/local/navigation_launcher.sh:

```
#!/usr/bin/bash

${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_navigation.launch
```

Since these three scripts are going to be executed. Let's turn them into executables:

```
chmod +x snap/local/*.sh
```

Using scripts

Now, let's use these scripts in our `snapcraft.yaml` instead of the "raw" commands. Remember that with the new part we added: "`local-files`", our scripts are going to be staged into `usr/bin`.

```
apps:
  core:
    [...]
-   command: opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
+   command: usr/bin/core_launcher.sh
    [...]
  mapping:
    [...]
-   command: opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
+   command: usr/bin/mapping_launcher.sh
    [...]
  navigation:
    [...]
-   command: opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
+   command: usr/bin/navigation_launcher.sh
```

Our `snapcraft.yaml` is now using scripts for the commands. They are the first scripts of our snaps since many are coming. For now, we might think that those scripts are a complexity we could avoid. But in the next sections, we will see that we will greatly need them.

Daemon scripts

In TurtleBot3 we already have a few daemons. These snap services can use multiple new features that will come in handy for our application.

`Snap` and `snapcraft` offer great features to orchestrate daemon applications. The *Application orchestration*[99] documentation is a great place to learn more about orchestration.

The orchestration features that we are going to use for the TurtleBot3 are:

- Command-chain

[99] https://ubuntu.com/robotics/docs/application-orchestration

  - We will use it to select the right multiplexer topic for controlling the robot.

- Stop-command
  - We will use it for our `mapping` application. This way we will trigger the save of the map whenever we exit the mapping. No more maps lost that took forever to create!
- Post-stop-command
  - We will use it to "install" the map for our `navigation` application. This way the `navigation` will always use the last created map.

Applying daemon scripts to TurtleBot3

We have already made our applications run in the background, it's now time to automate and simplify our user experience.

Here we want to make the map creation and use as simple as it can be. We don't want the user to have to call ROS nodes manually or even manipulate map files by himself.

To achieve this we will:

1. Save the map whenever we stop the `mapping`.
2. Use only the last map we created for the `navigation` without having to specify the new map name.
3. Verify that a map is available before starting our navigation stack.
4. Make the velocity command multiplexer automatically select the correct topic depending on the command we are using.

These are basic functionalities that our applications need to automate and simplify the user experience. Similarly, your application could also require implementing similar features.

Save the map

When the `mapping` is running, to save the map we will have to call the following ROS node:

```
rosrun map_server map_saver -f "PATH_AND_NAME_OF_THE_FILE"
```

Having to run it manually is, first, more work, but also the user must provide the correct path. Let's make this automatic!

The `stop-command` will be the perfect feature to call our map server. The only thing is that our `mapping` is currently a command. To use `stop-command` we will have to turn it into a daemon.

Let's modify our `snapcraft.yaml`:

```
apps:
  [...]
  mapping:
    environment:
      TURTLEBOT3_MODEL: waffle_pi
    command: usr/bin/mapping_launcher.sh
```

(continues on next page)

(continued from previous page)

```
+   daemon: simple
+   install-mode: disable
+   stop-command: usr/bin/save_map.sh
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

Now our `mapping` is a daemon (disabled by default, so it doesn't start by itself on the first install). It will launch a script called `save_map.sh` that we will create right now.

The script is simply going to run the map saver in the `$SNAP_USER_COMMON` directory (read/write directory kept over the updates). Let's create the file `snap/local/save_map.sh`:

```
#!/usr/bin/bash

# make map directory if not existing
mkdir -p ${SNAP_USER_COMMON}/map

$SNAP/opt/ros/noetic/bin/rosrun map_server map_saver -f "${SNAP_USER_COMMON}/map/new_map"
```

Now our map will be saved automatically whenever we stop our mapping service (with the `sudo snap stop mapping` command). The current limitation here is that every map that we create is called `new_map`. It would be better to keep all the maps created and simply mark one as the current one. This way we could even restore a previous one later.

Install the last map

Here we are going to make sure that we keep every map and also that `current_map` is always pointing to the last created map. This way our navigation will always load the same symlink file but always point to the latest map. To do so we will have to create a script installing the last map and execute it as a `post-stop-command`. Once the `stop-command` and our service is stopped our map is created, and it's time to install it.

Let's modify our `snapcraft.yaml` to add the `post-stop-command`:

```
apps:
  [...]
  mapping:
    [...]
+   stop-command: usr/bin/save_map.sh
    post-stop-command: usr/bin/install_last_map.sh
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

Now let's create our `install_last_map.sh` script in `snap/local`:

```
#!/usr/bin/bash
set -e
# backup our map with the date and time
DATE=`date +%Y%m%d-%H-%M-%S`
mv ${SNAP_USER_COMMON}/map/new_map.yaml ${SNAP_USER_COMMON}/map/$DATE.yaml
sed -i "s/new_map.pgm/$DATE.pgm/" ${SNAP_USER_COMMON}/map/$DATE.yaml
mv ${SNAP_USER_COMMON}/map/new_map.pgm ${SNAP_USER_COMMON}/map/$DATE.pgm
```

(continues on next page)

(continued from previous page)

```
# create symlink to use the map
rm -f ${SNAP_USER_COMMON}/map/current_map.yaml
ln -s ${SNAP_USER_COMMON}/map/$DATE.yaml ${SNAP_USER_COMMON}/map/current_map.yaml
```

Here we move the freshly created map files (prefixed with `new_map`) to new file names based on date. Finally, we create a UNIX symlink to point to the last map. This way we keep track of every map created and when it was created. Our navigation service is already loading the file `${SNAP_USER_COMMON}/map/current_map.yaml`. So by loading the symlink our `navigation` is always going to load the last created map.

Navigation launcher adaptation

Here we are simply going to adapt our navigation launcher to verify that our map file exists before launching the navigation launch file.

We already created the `navigation_launch.sh` script in `snap/local`:

```
#!/usr/bin/bash
+if [ -f "${SNAP_USER_COMMON}/map/current_map.yaml" ]; then
   ${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_navigation.launch
+else
+  >&2 echo "File ${SNAP_USER_COMMON}/map/current_map.yaml does not exist." \ "Have you
run the mapping application?"
+fi
```

The script is simply checking if the file `${SNAP_USER_COMMON}/map/current_map.yaml` exists and launches the navigation. Otherwise, we simply log an error message.

Velocity multiplexer selection

In the first sections of this guide, we described that a multiplier has been set to forward only one topic (`/joy_vel`, `/key_vel` or `/nav_vel`). Once selected the topic is forwarded to `/cmd_vel`. To select a topic we must call a ROS service with the command:

```
rosservice call /mux/select "/joy_vel"
```

Every time we call the `key` command (to start the keyboard teleoperation) we will have to call the ROS service to select our `/key_vel` topic.

Let's make this automatic every time we start any of the applications that need a specific topic. We are talking about `key`, `joy` and `navigation`.

Let's add a `command-chain` in our `snapcraft.yaml` calling a selection script for all our apps that need it.

```
apps:
  [...]
  key:
+   command-chain: [usr/bin/mux_select_key_vel.sh]
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_teleop key.launch
  [...]
  joy:
```

(continues on next page)

(continued from previous page)

```
+   command-chain: [usr/bin/mux_select_joy_vel.sh]
    command: opt/ros/noetic/bin/roslaunch turtlebot3c_teleop joy.launch
  [...]
  navigation:
+   command-chain: [usr/bin/mux_select_nav_vel.sh]
    command: usr/bin/navigation_launcher.sh
```

Now let's add the different scripts:

snap/local/mux_select_key_vel.sh:

```
#!/usr/bin/bash
$SNAP/opt/ros/noetic/bin/rosservice call /mux/select "/key_vel"
exec $@
```

snap/local/mux_select_joy_vel.sh:

```
#!/usr/bin/bash
$SNAP/opt/ros/noetic/bin/rosservice call /mux/select "/joy_vel"
exec $@
```

snap/local/mux_select_nav_vel.sh:

```
#!/usr/bin/bash
$SNAP/opt/ros/noetic/bin/rosservice call /mux/select "/nav_vel"
exec $@
```

Note that the exec `$@` is necessary at the end of our `command-chain` scripts since our actual command is given as an argument of the `command-chain`. Now with the help of our `command-chain` scripts whenever we start the `joy` app (or another controlling app) we are ready to control the robot with any additional commands.

Turtlebot3 configurations

In the previous section, we have seen how to add advanced behaviour to our snap by leveraging new `snapcraft` YAML keywords. Our final TurtleBot3 snap application must cover multiple cases. It should work with all the flavours of TurtleBot3 as well as with the simulation and the real robot.

Adding such behaviour is going to require a new feature from snap: `hooks`. `Hooks` are executable files that run within a snap's confined environment when a certain action occurs. They are typically used for configurations. The documentation on *snap configurations and hooks*[100] explains this in detail.

[100] https://ubuntu.com/robotics/docs/snap-configurations-and-hooks

Adding TurtleBot3 hooks

Here there are multiple things that we want to achieve:

- Our snap must be working for the simulation but also for the real robot. We will need a `simulation` configuration.
- TurtleBot3s come in different flavours: `burger`, `waffle` and `waffle_pi`. We will then also need a `turtlebot3-model` configuration.
- We will have to declare hooks to define our configurations and make sure the new value matches the requirements.
- Additionally, we will integrate those new parameters into our different scripts.

Simulation parameter

For the simulation, the only application that is going to be impacted is the `core` daemon. Since the snap is not meant to run the simulation itself but simply to interface with it, there is no big challenge here.

For this first parameter, we will have to create the hook scripts first.

Hooks

The very first hook that we are going to define is `The install hook`[101]. This hook is called upon initial installation only. We are going to use the `The install hook` to declare our parameter default value.

First we create the directory:

```
mkdir -p snap/hooks
```

Let's create a file called `install` inside the directory `snap/hooks` and add the following content:

```
#!/bin/sh -e
# set default configuration values
snapctl set simulation=false
```

We used `snapctl` to set the default value to false.

Additionally, we are going to create a file called `configure` in the `snap/hooks` directory.

Add the following content to `snap/hooks/configure`:

```
#!/bin/sh -e
SIMULATION="$(snapctl get simulation)"
case "$SIMULATION" in
  "true") ;;
  "false") ;;
  "0") ;;
  "1") ;;
```

(continues on next page)

[101] https://snapcraft.io/docs/supported-snap-hooks#heading--install

(continued from previous page)

```
  *)
    >&2 echo "'$SIMULATION' is not a supported value for simulation." \ "Possible values
are true, false, 0, 1"
    return 1
    ;;
esac

# restart core and teleop on new config
snapctl stop "$SNAP_INSTANCE_NAME.core"
snapctl stop "$SNAP_INSTANCE_NAME.teleop"
snapctl start "$SNAP_INSTANCE_NAME.core"
snapctl start "$SNAP_INSTANCE_NAME.teleop"
```

Here again we used `snapctl`. We use it to read the parameter and make sure it contains a valid value (in our case: `true`, `false`, `0` or `1`). In case the value is wrong we simply return an error.

Additionally, we made sure to restart our `core` and `teleop` daemons (meant to be always running). This way when we change a parameter we are sure that it is taken into consideration immediately.

Since our hooks are scripts we must make sure to make them executable. We can do so with:

```
chmod +x snap/hooks/*
```

Use the configuration

Now that our snap is defining our configuration we must use it in our application.

Let's modify our `core_launcher.sh` script located in `snap/local`:

```
#!/usr/bin/bash
+SIMULATION="$(snapctl get simulation)"
-${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
simulation:=true
+${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
simulation:=$SIMULATION
```

This way we read the snap configuration to store it into a variable and finally use it as a ROS argument for our launch file.

This way, when changing the configuration of the snap with the command:

```
sudo snap set turtlebot3c simulation=true
```

Our `configure` script is going to be called, restarting the `core` daemon. And finally our `core_launcher.sh` is going to be started with the correct configuration.

Turtlebot3 model

The TurtleBot3 has several hardware configurations. These models are `burger`, `waffle`, and `waffle_pi`. The different models are available for the real robot but also for the simulation. While we will mostly work with the `waffle_pi` simulation, the others can also be tested easily. Similarly, we will complete our hooks and the different scripts where we must specify the model.

Hooks

First, let's define the default value for our configuration `turtlebot3-model`. Let's add our configuration to the `snap/hooks/install` file:

```
snapctl set simulation=false

+# set default turtlebot 3 model
+snapctl set turtlebot3-model=waffle_pi
```

Similarly, we must complete the `snap/hooks/configure` hook:

```
+TURTLEBOT3_MODEL="$(snapctl get turtlebot3-model)"
+case "$TURTLEBOT3_MODEL" in
+  "burger") ;;
+  "waffle") ;;
+  "waffle_pi") ;;
+  *)
+    >&2 echo "'$TURTLEBOT3' is not a supported value for turtlebot3-model." \
+"Possible values are burger, waffle and waffle_pi"
+    return 1
+    ;;
+esac

# restart core and teleop on new config
snapctl stop "$SNAP_INSTANCE_NAME.core"
snapctl stop "$SNAP_INSTANCE_NAME.teleop"
```

Now, our hooks are correctly handling the `turtlebot3-model` configuration. Nothing new here we have applied what we used for the other configuration.

Use the configuration

Now let's use the robot model configuration for our applications. So far we have specified the robot model via the `environment` keyword in our application. This is no longer possible since we now need to use `snapctl` to read our configuration. The TurtleBot3 model is read by the mean of an environment variable[102]. We will then have to remove the hard-coded environment variable in our `snapcraft.yaml` and define it in our different launcher scripts.

First, let's remove the hard-coded one in our `snapcraft.yaml`:

[102] https://github.com/ROBOTIS-GIT/turtlebot3/blob/noetic/turtlebot3_bringup/launch/turtlebot3_robot.launch#L4

```
apps:
  core:
-   environment:
-   TURTLEBOT3_MODEL: waffle_pi
  [...]
  mapping:
-   environment:
+     TURTLEBOT3_MODEL: waffle_pi
  [...]
  navigation:
-   environment:
-     TURTLEBOT3_MODEL: waffle_pi
```

Now let's define this `TURTLEBOT3_MODEL` in the launcher scripts that need it. The first one is the `core` daemon. We can modify `snap/local/core_launcher.sh` this way:

```
SIMULATION="$(snapctl get simulation)"
+TURTLEBOT3_MODEL="$(snapctl get turtlebot3-model)"

+export TURTLEBOT3_MODEL
```

We read the parameter and then simply export the environment variable.

Now let's do the same thing but to the `navigation` and `mapping` launcher script: *snap/local/mapping_launcher.sh*.

```
+TURTLEBOT3_MODEL="$(snapctl get turtlebot3-model)"
+export TURTLEBOT3_MODEL
${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_2dnav turtlebot3c_mapping.launch
```

And *snap/local/navigation_launcher.sh*:

```
+TURTLEBOT3_MODEL="$(snapctl get turtlebot3-model)"
+export TURTLEBOT3_MODEL
if [ -f "${SNAP_USER_COMMON}/map/current_map.yaml" ]; then
```

Then we can make sure that all these scripts are executable:

```
chmod +x snap/local/*
```

From now on, we can select an alternative TurtleBot3 model with the simple command

```
sudo snap set turtlebot3c turtlebot3-model=burger
```

We have finally finished defining all our snap hooks. Additionally, we have implemented every application and functionality we need for our snap. It's time to build it and test it!

We will run all our tests in simulation, but with a TurtleBot3 robot available we could also test it on the real robot.

Final test

We can now do a final test of everything we achieved so far.

Since we did multiple tests, our writable directories already have data. To reproduce the first install of our snap, we can remove and purge our snap. This will not only remove our snap and previous revisions, but it will also clean all its data.

To do so:

```
sudo snap remove turtlebot3c --purge
```

We can now proceed to a complete test, from installation to navigation. Let's go through the following steps:

1. Install the snap:

   ```
   sudo snap install turtlebot3c_*.snap --dangerous
   ```

2. Set the simulation parameter

   ```
   sudo snap set turtlebot3c simulation=true
   ```

3. Start the simulation

   ```
   TURTLEBOT3_MODEL=waffle_pi roslaunch turtlebot3_gazebo turtlebot3_world.launch
   ```

4. Start the `mapping`

   ```
   sudo snap start turtlebot3c.mapping
   ```

5. Roam around with the `key` application

   ```
   turtlebot3c.key
   ```

6. Stop the `mapping`

   ```
   sudo snap stop turtlebot3c.mapping
   ```

7. Start the `navigation`

   ```
   sudo snap start turtlebot3c.navigation
   ```

8. Start `RViz`

   ```
   rviz -d /opt/ros/noetic/share/turtlebot3_slam/rviz/turtlebot3_gmapping.rviz
   ```

9. Test the navigation

We now have a complete snap that is completely tested. We created a map, we can now make sure the `navigation` daemon starts automatically at boot by enabling the service. Let's call the following command.

```
sudo snap start --enable turtlebot3c.navigation
```

In case we want all the `turtlebot3c` daemons to stop and not start again, we can disable the snap with:

```
sudo snap disable turtlebot3c
```

This will prevent any application and daemons from starting. Additionally, it will also prevent updates.

By checking the output of:

```
snap info turtlebot3c
```

We can confirm that all the daemons are stopped and disabled. Of course, we can enable it later with the snap enable command.

We have now presented every single step to snap a TurtleBot3 software stack. We went from application identification to actual snapping and finally added some advanced behaviour to automatize the map management tasks.

Exercise

Now that we have learned the ins and outs of the TurtleBot3 snap, we can apply some of our fresh knowledge for a quick exercise. The exercise as well as the solution are available in the *Developer guide part 2 - exercise* (page 46).

Part-2 - exercise: Clean old unused maps over time

This exercise requires having followed the *Tutorial 2:* (page 18).

This exercise is meant for developers to train on solving a problem with snaps. One can apply the freshly learned knowledge to a practical problem.

Assignment

With the `mapping` daemon we might create a lot of maps. Over time, this could become an issue. All the maps image might take too much space on our disk and if they are old they are most probably not necessary any more. Our snap could embed another application that would check once in a while if there is any old map that we don't need any more.

The assignment is to add a new background process that runs every day. This process must identify maps that are no longer used and older than one month. Finally, we must delete those old maps.

We might want to have a look at the `timer` feature of daemons, presented in the documentation[103].

Outcome

Our snap must have one more daemon called `map-cleaner`. We shouldn't have to call this application manually. The `map-cleaner` daemon should clean up our old and unused maps once a day.

Solution

Click here for the solution

First we must create a script that cleans up the maps. Our maps are located in `$SNAP_USER_COMMON`. Every map consists of two files, the `PGM` and the `YAML`. Since our map symlink points to the `YAML` we will use these files to identify the maps to delete.

Our `snap/local/map_cleaner.sh` script will look like:

```
#!/usr/bin/bash
# path to the last map
CURRENT_MAP=$(readlink -f $SNAP_USER_COMMON/map/current_map.yaml)
# get the list of YAML files older than a month except for the current map
LIST_OF_FILES=$(find $SNAP_USER_COMMON/map -maxdepth 1 -type f -mtime +30 -name "*.yaml
" ! -path $CURRENT_MAP)
# delete the YAML files
rm -f $LIST_OF_FILES
# delete the associated PGM
echo $LIST_OF_FILES | sed 's/yaml/pgm/' | xargs rm -f
```

After making the `map_cleaner.sh` script executable, we can add the following to our `snapcraft.yaml`:

```
  map-cleaner:
    command: usr/bin/map_cleaner.sh
    daemon: simple
    timer: "04:00" # runs every day at 4 am
```

Next Steps: Debugging

Now that we've explored and integrated all the new Snap features, our TurtleBot3 snap is feature-complete and ready for testing. We've defined our various `parts` and `apps` in the `snapcraft.yaml`, set up hooks for configuration management, and implemented scripts to simplify app launches and map creation. The next step is to thoroughly test the snap using the simulation with the TurtleBot3 `waffle_pi` model.

Below is a how-to guide focusing on debugging the snap itself and troubleshooting any issues with how the snap is packaged and how its components interact, rather than debugging the code behaviour or compilation errors. This will help us identify any potential problems with the snap's configuration and ensure a smooth user experience.

What we achieved here is perfectly working. If you would like to learn more about how to debug possible issues, please refer to the *how-to guide: Debugging Snap Applications* (page 94).

Conclusion

We are reaching the conclusion of this developer guide. This developer guide addressed almost any topic that one might need to ship a robot software as a snap.

The final version of the snap workspace is available on GitHub[104]. This is the official code that is used to publish the TurtleBot3c snap[105] from the store. This means that the GitHub repository will keep getting updated.

What we achieved

In this developer guide, we went through the creation of the TurtleBot3 snap. We have covered all the steps that one must go through when creating a snap for a robot.

First, we identified the different applications and daemons that our snap must expose. We also identified the scope of our snap. Then we applied the knowledge of the first part of the developer guide to build the first draft of our snap. By discovering new features of snaps we were able to implement features to manage our maps and to select the right velocity topic. We implemented these features to simplify and automate the usage of the snap.

We finally covered the snap debugging aspect. This helped us to solve all the different issues we encountered while testing the snap.

Now the whole stack of TurtleBot3 for the simulation or the real robot is packed in a single snap.

[104] https://github.com/canonical/turtlebot3c-snap/tree/noetic-devel
[105] https://snapcraft.io/turtlebot3c

Distribute ROS applications with the Snap Store

Distribute ROS applications with the Snap Store

In part 1[106] and part 2[107] of our developer guide series, we learned about what snaps can do for ROS applications by exploring the use of snap and snapcraft in complex robotics applications.

However, after the ROS application has been snapped, one might wonder:

- How can we distribute this package?
- How can we distribute updates?
- When can our devices or users benefit from the update?

In this developer guide, we will explore how to share and distribute our snapped robotics applications like a global software vendor.

This guide is meant for ROS snap beginners and advanced users looking for guidelines for their snap distribution.

What we will learn

Deploying robotics software is usually tackled when a project reaches a certain level of maturity. We need a fast and reliable way to distribute software and subsequent updates. Snapping a robotics stack is our means to distribute our software globally.

Snap and snapcraft contain many features that will come in handy when distributing our robot application. This guide will help us understand these features and how to use them.

In this guide, we will learn:

- How to upload our snap for the first time to the public store.
- How to update the snap and distribute the update.
- Ways for the final user or device to benefit from the updates.

While covering the theoretical aspects of distributing ROS snaps, we will apply all this knowledge to the ROS 2 Humble talker-listener example[108] as a practical use case.

We will cover the use of the public Snap Store in this developer guide. Dedicated Snap Stores are also available under a commercial licence. They include all the features from the public Snap Store along with additional resources described in the dedicated Snap Store documentation[109].

[106] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[107] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-2
[108] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap/tree/confined
[109] https://ubuntu.com/core/docs/dedicated-snap-stores

Requirements

We will need basic ROS knowledge. Ideally, we will already understand how to build ROS packages, call a service, and know what a launch file is. We should also be familiar with the process of creating a map and using it to navigate.

A basic understanding of the Linux environment (Ubuntu) is also required.

Finally, we expect the reader to have a working knowledge of snaps. This means either previous experience with `snap` and `snapcraft`, or having followed the Deployment Guide - PART 1[110].

Setup

In terms of setup, we require a more minimal setup than we used in part 2[111].

We will need an up and running Ubuntu Desktop (20.04 being the minimum version, as it is still under maintenance). This tutorial cannot be followed inside a container or a headless environment.

On top of this OS we will need to:

- Install and initialise LXD.[112]
- Install snapcraft.[113]

Additionally, we will need an Ubuntu One[114] account for the Snap Store[115]. We will use this account to identify ourselves to the store.

As we're building a strictly confined ROS 2 snap, we don't need additional ROS 2, or any other, dependencies.

Globally distribute our first snap

In part 1[116] we covered the creation of a strictly confined ROS 2 Humble talker-listener snap[117]. For this guide, we will reuse the same example.

Let's recall what we did during part 1[118].

[110] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[111] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[112] https://ubuntu.com/lxd/install
[113] https://snapcraft.io/snapcraft
[114] https://login.ubuntu.com/+description
[115] https://snapcraft.io/
[116] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[117] https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap/tree/confined
[118] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1

Creation of our snap

We will first download the work done in part 1[119]. We will then adapt the snap to make it unique before finally testing our snap.

ROS 2 humble talker listener snap

To avoid repeating the work from the previous developer guide, we will clone the ROS 2 humble talker-listener snap repository:

```
git clone https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap.git -b confined
```

This repository mostly contains the file `snap/snapcraft.yaml`, which is the entry point to create the snap through Snapcraft.

This snap is using the demo_nodes_cpp[120] package from the ROS 2 demo[121] repository.

The `snapcraft.yaml` contains only one part called `ros-demos` that simply builds and installs the `demo_nodes_cpp` package. It also exposes one application called `ros2-talker-listener` that launches the ROS 2 talker-listener demo launch file.

Customise our snap

Every snap on the public Snap Store[122] must have a unique name, which means the first step is to give our own snap a unique name.

For this guide, we will simply prefix our snap name by our pseudonym.

To do so, we will edit the `snapcraft.yaml`:

```
- name: ros2-talker-listener
+ name: yourname-test-ros2-snapstore
```

Now our snap will be unique, we can build it to make sure everything is working as expected.

Test our snap

The first step is to build the snap with the `snapcraft` command:

```
snapcraft
```

Even if the snap had been built previously, the new snap name will cause the build process to start again from the beginning.

Once the snap has been built, we will see:

[119] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[120] https://github.com/ros2/demos/tree/humble/demo_nodes_cpp
[121] https://github.com/ros2/demos.git
[122] https://snapcraft.io/store

```
Created snap package yourname-test-ros2-snapstore_0.1_amd64.snap
```

We can now install it from the generated file with the following command:

```
sudo snap install yourname-test-ros2-snapstore_0.1_amd64.snap --dangerous
```

And finally, we can test it by running the following command (replacing *yourname* with your pseudonym):

```
yourname-test-ros2-snapstore.ros2-talker-listener
```

Note that here we use <snap name>.<command> because the application and snap have different names (we could have assigned the snap name to the application to be able to omit the second part of the command).

We should now see the "Hello world" messages:

```
[talker-1] [INFO] [1696952444.819483634] [talker]: Publishing: 'Hello World: 1'
[listener-2] [INFO] [1696952444.820526860] [listener]: I heard: [Hello World: 1]
[talker-1] [INFO] [1696952445.819339673] [talker]: Publishing: 'Hello World: 2'
[listener-2] [INFO] [1696952445.819848321] [listener]: I heard: [Hello World: 2]
```

Now we have tested our local snap with our unique name, it is time to register our unique name on the public Snap Store[123].

Let's remove the snap now, as we will now use the Snap Store:

```
sudo snap remove yourname-test-ros2-snapstore
```

If you had issues following any of the previous steps, please, review the developer guide *part 1* (page 2) again.

Register our snap

In order to upload our snap to the Snap Store, a developer account is required. This account is registered on Ubuntu One, which is a single sign-on service for Ubuntu and related projects. If you don't already have an account, head over to https://snapcraft.io/account and select the **"I don't have an Ubuntu One account"** option. More information on how to check your account ID and setup SSH keys can be found at https://snapcraft.io/docs/creating-your-developer-account.

CLI

Now that we have an account, we can proceed with the registration of our snap. First, we log in into our account via the CLI with the following command:

```
snapcraft login
```

Note that this must be run on the host and not in a container or a headless server.

We can find more information in the documentation about the snapcraft authentication[124].

[123] https://snapcraft.io/
[124] https://snapcraft.io/docs/snapcraft-authentication

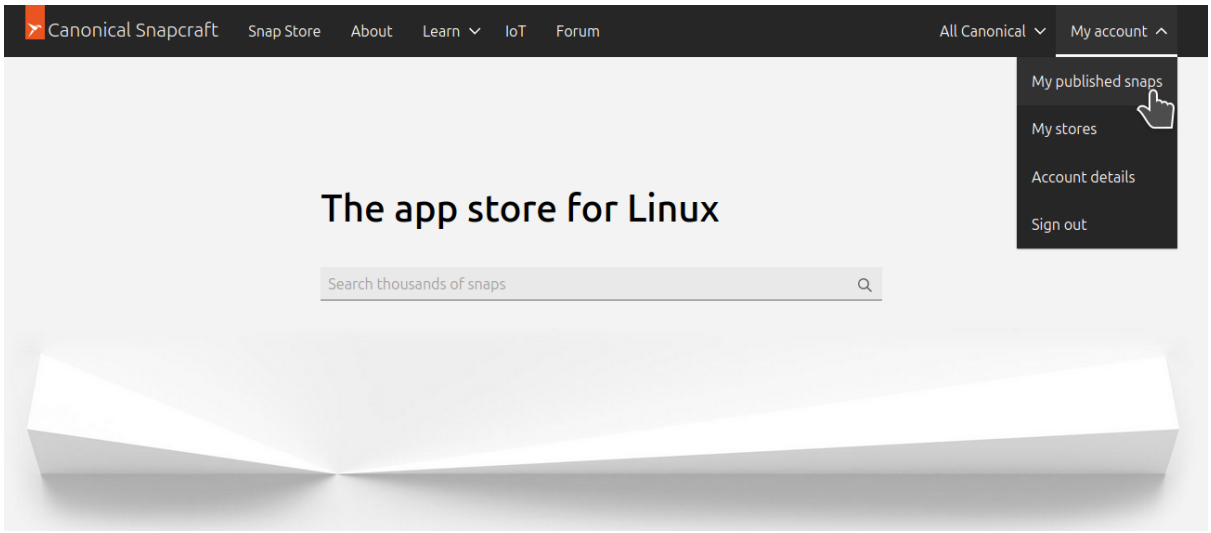
If the login is successful, we can now register our snap as follows:

```
snapcraft register yourname-test-ros2-snapstore
```

The output will require us to agree that most users will expect our snap name to represent the snap we're wanting to publish. Respond yes and then our snap will be correctly registered.

Web UI

It is also possible to register a snap directly from the snapcraft.io[125] website. To do so we can log into our developer account, select "My published snaps" in the account menu and then click *"Register a snap name"* on the top right of the page.



Upload and release our snap

After creating and registering our snap, the next step is to upload it on the Snap Store. Once on the Snap Store, our snap can be exposed with public, private or unlisted visibility[126].

Upload on the Store

Let's navigate to the folder in which our own built and tested snap is stored, and let's upload it to the store with the following command:

```
snapcraft upload yourname-test-ros2-snapstore_0.1_amd64.snap
```

The upload process triggers a series of automated reviews to inspect our snap. The inspection includes a security and integrity check to assure that our snap will run. Our application will be available for installation after these reviews have completed without errors

However, while the snap has been uploaded, it is not yet released to the public. Let's release it!

[125] https://snapcraft.io/
[126] https://snapcraft.io/docs/public-private-unlisted-snaps

Release the snap

The release of a snap sets its track and the risk level. Channels are an important snap concept. They define which release of a snap is installed and tracked for updates. For example, they allow developers to:

- Distribute software with different ROS versions under the same name.
- Distribute an application on stable and edge. The stable channel promises reliability, while the edge channel contains the latest features.
- Distribute fixes or temporary experimentation using branches that after 30 days with no further updates, will close automatically.

By default, all snaps are pushed to the `latest` track, which is typically used to manage different versions of the same application. A good example of the track usage with ROS application would be to release each ROS distro on different tracks.

For more information on snap tracks and channels, please read the documentation[127].

For our application, we will use the default track (latest) and the stable risk-level.

To release our snap *`yourname-test-ros2-snapstore`* revision 1 on `latest/stable` we will use:

```
snapcraft release yourname-test-ros2-snapstore 1 latest/stable
```

Additionally, we could have combined the upload and the release with the following command:

```
snapcraft upload --release=latest/stable yourname-test-ros2-snapstore_0.1_amd64.snap
```

Once our snap has been uploaded on the Snap Store, we can switch an uploaded snap between different channels using either the command line or the Snap Store website UI, all the information on release management can be found at https://snapcraft.io/docs/release-management.

Also note that the release process can be managed on the Snap Store UI on the `releases` tab.

We can now verify our snap is on the store by searching for it on https://snapcraft.io.

Since our snap won't be useful to others, let's change the visibility from public to unlisted. To do so, we must go to our snap list[128], select our snap. Then in the "Settings" tab, we can select "Unlisted".

While we are doing all this process manually here, all of this can be integrated into a CI workflow. We can find more information in the *ROS snap with GitHub Actions* (page 88) documentation.

[127] https://snapcraft.io/docs/channels
[128] https://snapcraft.io/snaps

Download our snap

Now that our snap has been uploaded and released, it can be installed from the Snap Store. To install our freshly released snap, run the following command:

```
sudo snap install yourname-test-ros2-snapstore
```

Notice how the `--dangerous` flag is no longer required. This change is a result of our snap being uploaded to the Store and undergoing the revision and signing process. With the snap now public and available in the Store, it becomes accessible to anyone worldwide!

The same command can be used on our robots, we can already start deploying our own snaps in our robotic fleet! If you want to create production grade images, we recommend using Ubuntu Core. For more information, please read the documentation about Creating Custom Images[129].

Update-our-snap

Once deployed, your application will need to be updated (e.g. to apply a fix, a new feature, a security patch). The Snap Store infrastructure can help you manage and deliver those updates to your devices, while we enjoy features like delta updates and over-the-air (OTA) updates.

Modify the snap

Let's now modify our snap to see how updates work in more detail. This may require source-code changes and/or `snapcraft.yaml` changes.

For this example, we are going to use a different launch file for our talker listener. The ROS 2 demo talker listener is also available via an XML launch file[130]. To simulate our update, we are going to replace the Python launch file with the XML one.

Change the launch file

To update the launch file that we use in our snap, we are going to change the file `snap/snapcraft.yaml` as follows:

```
name: ros2-talker-listener
-version: '0.1'
+version: '0.2'
[...]
apps:
  ros2-talker-listener:
-   command: opt/ros/humble/bin/ros2 launch demo_nodes_cpp talker_listener.launch.py
+   command: opt/ros/humble/bin/ros2 launch demo_nodes_cpp talker_listener.launch.xml
```

In addition to changing the launch file, we have also increased the snap version. This is not a snap requirement, but will be convenient for our users!

Now that we have modified our `snapcraft.yaml` we can rebuild it with the command:

[129] https://documentation.ubuntu.com/core/tutorials/build-your-first-image/
[130] https://github.com/ros2/demos/blob/humble/demo_nodes_cpp/launch/topics/talker_listener.launch.xml

```
snapcraft
```

A second snap file was generated this time and we can now distribute this update.

Upload the update

The freshly built snap file is called: `yourname-test-ros2-snapstore_0.2_amd64.snap`. We can see in the name that the new version was used. Let's upload and release the update with the following command:

```
snapcraft upload --release=latest/stable yourname-test-ros2-snapstore_0.2_amd64.snap
```

By uploading the update and releasing it on `latest/stable` we have replaced the version that people will download when installing the application. This means that if anyone now installs our snap from the `latest/stable` they will get the version 0.2.

Get the update

With the new version of our snap now available on the Snap Store, we can probe the Snap Store to see both the version we have installed and the update:

```
$ snap info yourname-test-ros2-snapstore

name: yourname-test-ros2-snapstore
summary: ROS 2 Talker/Listener Example
publisher: -
store-url: https://snapcraft.io/yourname-test-ros2-snapstore
license: unset
description: |
    This example launches a ROS 2 talker and listener.
commands:
    - yourname-test-ros2-snapstore.ros2-talker-listener
snapd-id:  YNWjzdWUV1aNIXBMt1hOAToArNWqTsJX
tracking:  latest/stable
refresh-date: today at 15:39 CEST
channels:
    latest/stable: 0.2 2023-10-24 (2) 69MB -
    latest/candidate: 
    latest/beta:  
    latest/edge:  
    installed:   0.1   (1) 69MB -
```

As we can see in bold, we are tracking the `latest/stable` channel and have the version 0.1 installed. We can also see that the channel `latest/stable` now has the version 0.2 available.

`Snapd` hasn't automatically updated our snap yet. By default, checks are automatically happening four times per day, with each check referred to as a "refresh". The timing and frequency of these updates can be customised using the `snap refresh` command.

Therefore, we could simply wait for `snapd` to automatically update in the background, or we can force a refresh of our snap with:

```
sudo snap refresh yourname-test-ros2-snapstore
```

Our snap is now updated, and we can verify it with the `snap info` command.

To verify that our freshly updated snap works well, we can type:

```
yourname-test-ros2-snapstore.ros2-talker-listener
```

See Managing updates documentation[131] to learn more about snap updates and how to monitor and manage them.

Conclusion

Our guide covered the basic steps required to release a robotics snap on the Snap Store. The final version of the snap can be found on the Snap Store[132]. We didn't cover features like air gap systems and proxies.

What we achieved

In this developer guide, we went through the first upload and release of a ROS snap. While covering the basics, we included links to more resources that explore the Snap Store topic in more depth in the Snapcraft documentation[133].

By uploading our first version and releasing it, we learnt that we can distribute our own application.

Similarly, we saw how to upload and distribute an update for our application, so our users and devices can benefit from it.

Make sure to check the official Snapcraft documentation[134] to learn more about the Snap Store, the updates, the branches and much more!

Part 4: Building ROS snaps with content sharing

How to use this developer guide

In *part 2* (page 18) of our developer guide series, we saw in fine detail how to create a single snap for a complete robot stack, including things like controllers, sensor drivers, but also functionalities such as autonomous navigation.

While *part 3* (page 48) of our series focused on the snap store and the snap release process, this developer guide, part 4, draws from the example shown in part 2 to exemplify one specific feature of snaps, content sharing[135]. We will therefore revisit the entire example in order to make use of this feature – after introducing it, of course.

This guide is meant for ROS snap beginners and advanced users looking for insights and guidelines on implementing ROS snap content sharing. This is not a quick tutorial, but

[131] https://snapcraft.io/docs/managing-updates
[132] https://snapcraft.io/yourname-ros2-talker-listener
[133] https://snapcraft.io/docs
[134] https://snapcraft.io/docs/releasing-your-app
[135] https://snapcraft.io/docs/content-interface

rather an in-depth guide. Furthermore, these steps assume that you have completed *part 2* (page 18) of this series.

Packaging complex robotics software with multiple snaps

We have established in *part 2* (page 18) the importance of packaging your applications for deployment. We also demonstrated this packaging by building a single snap that contains the entire software stack for the TurtleBot 3. While this monolithic solution is convenient, it also has several drawbacks such as its size and the resultant resources it consumes when installed and updated. More importantly, it also ties the whole stack to a specific robot model.

Ideally, parts of the entire software stack should be reusable on other robot models; the teleoperation application and the navigation stack are two such examples. This could be achieved by breaking down the monolithic design into several smaller snaps, each enabling a specific functionality.

For instance, we could design a single robot-specific snap, containing all of the robot-specific stack components such as drivers, controllers, robot model and so on. Alongside this snap we can add a plethora of other snaps which are robot agnostic, each enabling a given functionality. These snaps could then be used on a variety of different robots, potentially varying only slightly in their configuration.

This multi-snap design is exactly the theme of this guide. For a detailed comparison of the different architectures with snaps[136] you can check the documentation.

Rationalising a multi-snap deployment with content sharing

A snap being self-contained is a powerful feature. But sharing resources, such as the same ROS packages, can also be useful. Not only does sharing avoid duplication, it also helps to ensure related applications share the same version of the ROS stack.

Such design – sharing a common set of libraries among multiple snaps – is readily available through a feature called content sharing. However, using it is anything but straightforward. We must make sure that all dependencies are available at build time but not shipped in the final artefact. We must also make sure that the `ROS workspaces` are properly sourced, wherever they are and whenever they are needed. This can quickly become a headache.

Fortunately, new ROS extensions for snapcraft have been developed precisely to enable seamless content sharing. We'll be using those extensions in this guide.

What we will learn

Just like in *part 2* (page 18), we will be deploying a robotics software stack using snaps and snapcraft. Following the TurtleBot3[137] example, we will divide the monolithic snap presented in part 2 into four (4) standalone snaps, namely:

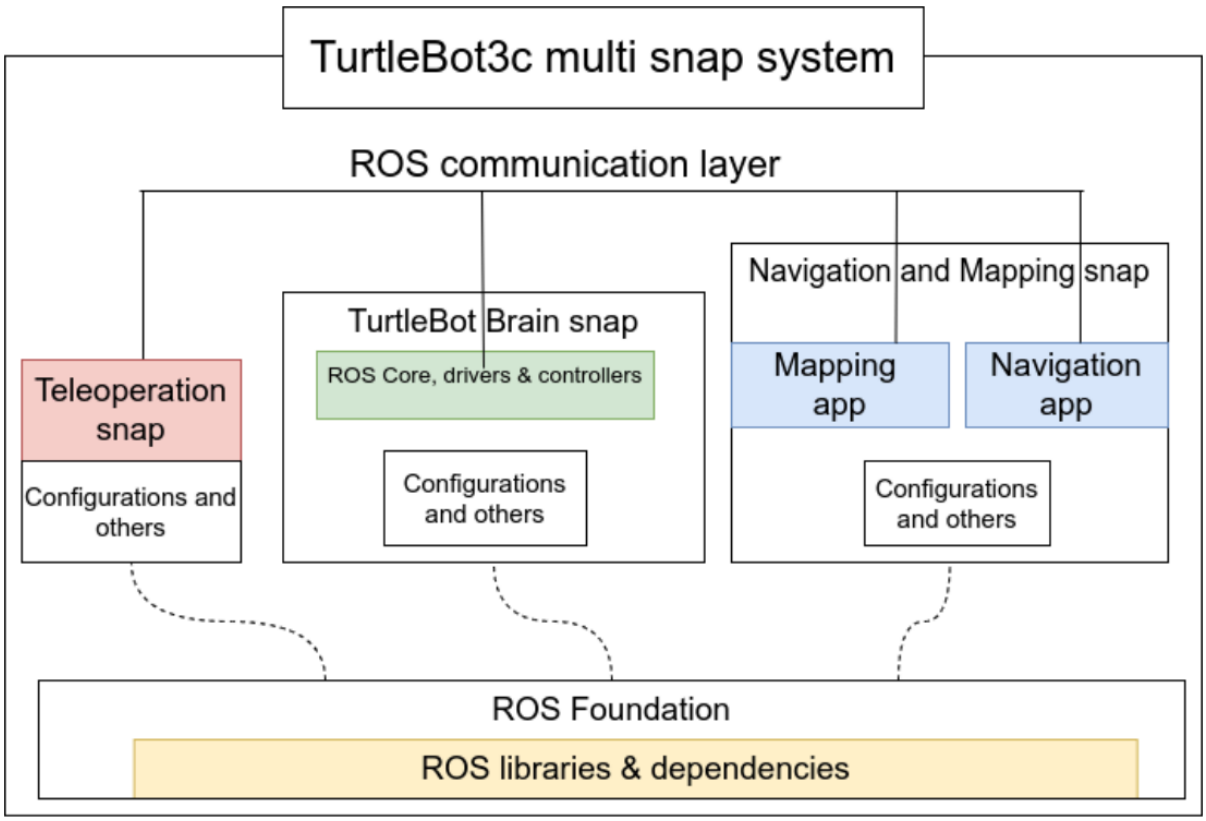
- turtlebot3c-core: The robot specific snap required to bring the robot up. We may also refer to it as the 'brain' snap.
- turtlebot3c-teleop: A teleoperation snap allowing us to drive our robot.

[136] https://ubuntu.com/robotics/docs/ros-architectures-with-snaps
[137] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

- turtlebot3c-nav: A snap that contains both the mapping and autonomous navigation functionality.
- A foundational snap.

The foundational snap is the basis for the content sharing design and, as its name suggests, it's a snap by itself. While we haven't yet dug into the inner workings of content sharing, we briefly suggested that the common bits live outside the application snaps. As a matter of fact, they live in a separate snap of their own, a snap that only contains libraries and a few executables (think ros2 run/launch) but exposes no applications.

We will design the four snaps so that they are standalone, yet loosely integrated to one another offering some ease of use for the final user. The image below shows a preliminary idea of what we will build in this tutorial.



In this fourth part, we will learn:

- How to draw a line and separate functionalities in their own snap.
- How to build a ROS snap using content sharing.
- New snap features and concepts that will be useful in a robotics context.

Requirements

The main requirement is that you have already completed *part 2* (page 18) of this guide. This implies that you are familiar with ROS, Linux and snap/snapcraft.

Setup

Since we are reusing part 2 examples, the setup is the same. Please *refer back* (page 18) for the details of the setup either on your machine or in a virtual machine.

The ROS content sharing feature we are going to employ later is only available on snapcraft version 8.x. Make sure that you have the right version of snapcraft installed with:

```
snapcraft version
```

Identification of robot components

When creating a monolithic snap for an entire robot stack, we end up placing many different applications and functionalities in the same blob. As we want to split that monolithic snap into smaller chunks that are as standalone as possible, we need to consider where to draw the line.

Fortunately, the job of identifying the high level functionalities was mostly done in *part 2* (page 22) where we identified the different applications to be exposed by the snap. Following the same dotted lines, we will consider the following functionalities:

1. BringUp, or the obvious need to bring the robot up to a functioning state. This includes things such as spawning controllers and drivers, and advertising the robot state and model.
2. Teleoperation, or the possibility to drive the robot from a keyboard, a `gamepad`, a virtual joystick or similar device.
3. Navigation and Mapping, or the ability for the robot to create a representation of its environment and autonomously navigate in it.

As a matter of fact, we will reuse the very same TurtleBot3c[138] project's source code. Turtlebot3c is a collection of launch files and configuration files organised so that the aforementioned functionalities can be launched individually. This is something that is not necessarily doable out-of-the-box from the official TurtleBot3[139] code base.

[138] https://github.com/canonical/turtlebot3c/tree/noetic-devel
[139] https://www.turtlebot.com/

Snapping the TurtleBot3

With the functionalities clearly identified, we shall now move on to creating the snaps.

We can follow the same template from our previous tutorial, change the names of the 3 snaps and include only the respective apps and daemons.

In the following sections we will clone the `snapcraft.yaml` defined in the developer guide part 2 and adapt it to generate 3 different snaps; “Bringup”, “Teleoperation”, and “Navigation and Mapping”.

BringUp

The full “BringUp” `snapcraft.yaml` can be found here https://github.com/canonical/turtlebot3c-snap/tree/feature/multi-snap/turtlebot3c-bringup-snap/snap.

First, change the name to `turtlebot3c-bringup` and adapt the summary and description based on the separation we described before.

```
name: turtlebot3c-bringup
summary: Turtlebot3c core snap
description: |
  This snap automatically spawn a roscore and the core components for the
  Turtlebot3 (controller, robot_state_publisher etc).
```

Next, we can simplify the workspace craft part because we only need to clone a specific sub-directory from a single repository.

So let’s use the `source-subdir` for cloning and focus only on the `turtlebot3c_bringup` folder.

```
parts:
  workspace:
    plugin: catkin
    source: https://github.com/ubuntu-robotics/turtlebot3c.git
    source-subdir: turtlebot3c_bringup
```

Remove everything related to the teleoperation, navigation and mapping from the original file. In fact, in the BringUp snap we want to keep the `core_launcher` app and nothing else.

```
apps:
  core:
    daemon: simple
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
      # LD_LIBRARY_PATH: "$LD_LIBRARY_PATH:$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/blas:
$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/lapack"
    command-chain: [usr/bin/ros_network.sh]
    command: usr/bin/core_launcher.sh
    plugs: [network, network-bind, raw-usb]
    extensions: [ros1-noetic]
```

Note that we have not enabled content sharing just yet. The snap works already by having all the required ROS dependencies inside. Let’s continue with the other 2 snaps and afterwards we will show how to enable content sharing.

Teleoperation

The full Teleoperation `snapcraft.yaml` can be found here https://github.com/canonical/turtlebot3c-snap/tree/feature/multi-snap/turtlebot3c-teleop-snap/snap.

Apply the snap rename and adapt the summary and the description:

```
name: turtlebot3c-teleop
summary: Turtlebot3c teleop snap
description: |
  This snap automatically contains a teleoperation app along with selectors for
controlling the Turtlebot3 with the keyboard or a joystick.
```

Adapt the `source-subdir`:

```
    source-subdir: turtlebot3c_teleop
```

We can now remove all the `navigation` and `mapping` sections, as well as the `core` which is now in the `bringup` snap.

This leaves us with the `teleop`, `key` and `joy` sections.

The final result looks like this:

```
apps:
  core:
    daemon: simple
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
      # LD_LIBRARY_PATH: "$LD_LIBRARY_PATH:$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/blas:
$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/lapack"
    command-chain: [usr/bin/ros_network.sh]
    command: roslaunch turtlebot3c_teleop turtlebot3c_teleop.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]

  key:
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
    command-chain: [usr/bin/ros_network.sh, usr/bin/mux_select_key_vel.sh]
    command: roslaunch turtlebot3c_teleop key.launch
    plugs: [network, network-bind]
    extensions: [ros1-noetic]

  joy:
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
    command-chain: [usr/bin/ros_network.sh, usr/bin/mux_select_joy_vel.sh]
    command: roslaunch turtlebot3c_teleop joy.launch
    plugs: [network, network-bind, joystick]
    extensions: [ros1-noetic]
```

Mapping and Navigation

Our third snap will contain the mapping and navigation bits. The steps are the same as before.

The final result is here https://github.com/canonical/turtlebot3c-snap/tree/feature/multi-snap/turtlebot3c-nav-snap/snap.

First change the name and the description:

```
name: turtlebot3c-nav
version: '0.1'
license: GPL-3.0
summary: Turtlebot3c mapping and navigation snap
description: |
  This snap provides apps to create a map and if a map is available run localization/
planning so that the  Turtlebot3 can navigate in the map.
```

Adapt the `source-subdir`:

```
    source-subdir: turtlebot3c_2dnav
```

Remove the unrelated apps and keep only the `mapping` and `navigation` apps:

```
apps:
  mapping:
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
    command-chain: [usr/bin/ros_network.sh, usr/bin/mux_select_key_vel.sh]
    command: usr/bin/mapping_launcher.sh
    daemon: simple
    install-mode: disable
    stop-command: usr/bin/save_map.sh
    post-stop-command: usr/bin/install_last_map.sh
    plugs: [network, network-bind]
    extensions: [ros1-noetic]

  navigation:
    environment:
      ROS_HOME: $SNAP_USER_DATA/ros
      # map server need pulseaudio
      # Defining this var here overwrite the extension definition,
      # thus we need to respecify the extension paths
      LD_LIBRARY_PATH: "$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/pulseaudio:$LD_LIBRARY_PATH"
    command-chain: [usr/bin/ros_network.sh, usr/bin/mux_select_nav_vel.sh]
    command: usr/bin/navigation_launcher.sh
    daemon: simple
    install-mode: disable
    plugs: [network, network-bind]
    extensions: [ros1-noetic]
```

Using content sharing

Up to this point in the tutorial, the snaps we have created are self-sufficient and are not using the ROS foundational snap.

The snaps are working and could be already used in a robot, but in order to gain the advantages of the content sharing we need to change the extensions from `ros1-noetic` to the `ros1-noetic-robot`. It's a single line of change, but it enables a different extension that will make sure all the ROS packages contained in the `ros-noetic-robot` meta-package are connected from the foundational snap into your newly created multiple snaps.

```
-   extensions: [ros1-noetic]
+   extensions: [ros1-noetic-robot]
```

The Ubuntu Robotics Community[140] maintains multiple ROS foundational variants based on REP-2001[141] such as `ros-core`, `ros-base`, `desktop` for multiple ROS distro (e.g. Noetic, Foxy, Humble).

For more details and all the possible extensions you can use as foundational please check the documentation[142] and search for extensions with a name including "Content Sharing" (e.g. "ROS2 Humble Content Sharing").

After changing the extension to make use of content sharing, we can re-create the snaps using snapcraft and install them. Once done, let us verify that the content sharing interface is connected. With the turtlebot3c-bringup snap as an example run,

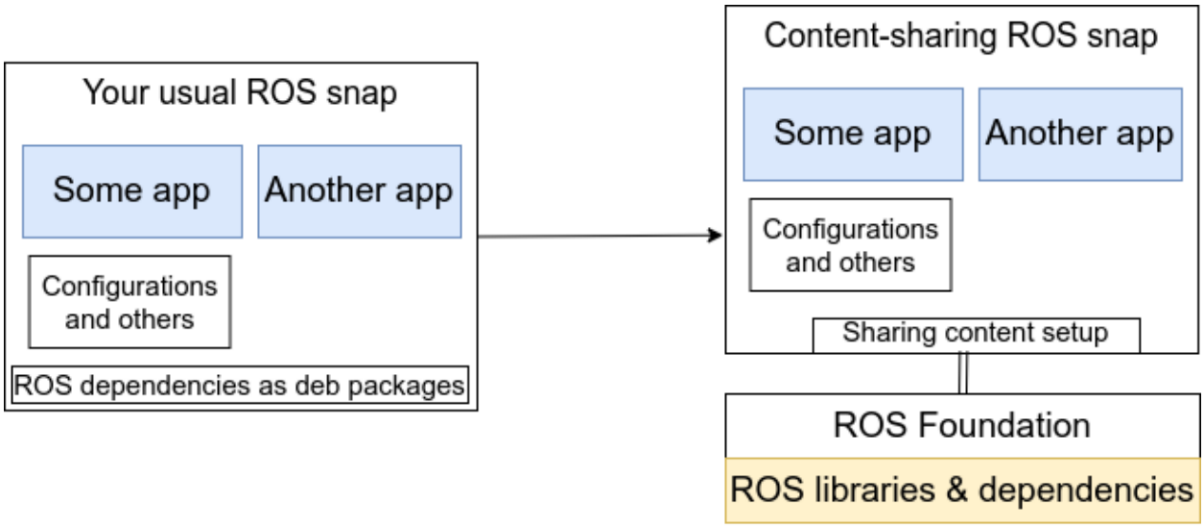
```
$ snap connections turtlebot3c-bringup
Interface            Plug                                Slot                           Notes
content[ros-noetic]  turtlebot3c-bringup:ros-noetic      ros-noetic-robot:ros-noetic    -
network              turtlebot3c-bringup:network         :network                       -
network-bind         turtlebot3c-bringup:network-bind    :network-bind                  -
raw-usb              turtlebot3c-bringup:raw-usb         -                              -
```

As we can see, the interface `content[ros-noetic]` is automatically connected. Our snap plug (`turtlebot3c-bringup:ros-noetic`) is connected to the slot (`ros-noetic-robot:ros-noetic`) of the `ros-noetic-robot` snap which provides the ROS Noetic stack.

[140] https://snapcraft.io/publisher/ubuntu-robotics-community
[141] https://ros.org/reps/rep-2001.html
[142] https://snapcraft.io/docs/supported-extensions

Your ROS snap with and without content sharing

It's worth looking more in detail at what happens when the content sharing method is used by the `ros1-noetic-robot` extension.



Remember that when using the `ros1-noetic` extension, all the dependencies of your ROS packages are read from the `package.xml` files and installed using `apt`. They will be shipped with your snap as usual.

When you switch to the new foundational snaps, such as `ros1-noetic-robot`, most of the basic ROS packages are already installed, so when reading the `package.xml` files the snapcraft process skips those and only installs the missing ones inside the snap.

When the snap is installed and started in your robot it will connect to the content from the foundational snap and get all the ROS packages in there, as if they were shipped with your own snap.

There are some very important advantages to this approach:

- The foundational snap is rarely updated, this reduces the bandwidth during OTA updates.
- Your snaps all fetch content from the same foundational snap, meaning there are no longer copies of the same packages across multiple snaps. This reduces disk usage and ensures the same versions are being used across snaps.

See below for a size comparison of the snaps before and after the use of content sharing that we described in this guide.

	Without using foundational snap	Using foundational snap	
Name	Size (Mb)	Size (Mb)	
turtlebot3c-bringup	205	5	
turtlebot3c-nav	296	61	
turtlebot3c-teleop	133	2	
ros-noetic-robot		161	
	634	229	**total**

For a deployment of the full navigation stack, a bringup and a teleop we have a size reduction of 64%.

Conclusion

We have reached the end of this developer guide that explores the use of content sharing for ROS snaps.

What have we achieved

What we achieved in the developer guide is no small feat. Starting from the example detailed in part 2 of the developer guide series, we have built a multi-snap schema to deploy the TurtleBot3 software stack. In doing so, we have practised how to split a stack into several independent, yet integrated, functionalities. This enables a clear separation of concerns and turns several of our snaps into robot agnostic functionalities that can be deployed on different platforms.

We have also learnt about the snap content sharing feature and experienced how this feature can be seamlessly employed for ROS snaps.

What's next?

This guide is part of a larger series covering different aspects of the snap ecosystem. While you should have gone through part 2 to end up here, you may have skipped part 3 which covers the Snap Store[143].

The next developer guide will cover Ubuntu Core[144], a secure, `application-centric` OS for embedded devices. We will explore how we can use Ubuntu Core to create production images with our application snaps to simplify the deployment of robots to market. It will also cover various aspects of the security, operation and management of snaps.

You can also visit the robotics documentation[145] to find out more about snaps and the other services offered by Canonical, such as ROS ESM[146]. If you have any questions or need help, you can visit and post your question on the snapcraft forum[147].

Part 5: Create an Ubuntu Core image for the TurtleBot3

When developing a robot, one might want to have an Ubuntu image ready to be flashed on a robot. With Ubuntu Core, developers can prepare an image embedding all the software necessary for a robot as well as configurations, so the robot is ready to be used from the first boot without manual intervention.

This tutorial will guide you through the steps required to create your own Ubuntu Core image for a TurtleBot3, with all the necessary snaps, and install it on a Raspberry Pi 4.

Requirements

For this tutorial, you will need:

- A developer account[148], and be logged in with snapcraft
- Some basic snap/snapcraft knowledge
- A TurtleBot3[149] with a RaspberryPi 4[150]
- An additional screen or, alternatively, a serial adaptor.
- Snapd[151] and snapcraft[152] installed

[143] https://snapcraft.io/store
[144] https://ubuntu.com/core
[145] https://ubuntu.com/robotics/docs
[146] https://ubuntu.com/robotics/ros-esm
[147] https://forum.snapcraft.io/
[148] https://snapcraft.io/docs/creating-your-developer-account
[149] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
[150] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
[151] https://snapcraft.io/snapd
[152] https://snapcraft.io/snapcraft

The goal

For this tutorial, our goal will be to create an Ubuntu Core image that is ready to be installed and used on a TurtleBot3 robot equipped with a Raspberry Pi 4.

The image will include all the TurtleBot3c snaps that we prepared in the Tutorial part 4: Building ROS snaps with content sharing[153]:

- Turtlebot3c-bringup[154]: The core components for the TurtleBot3
- Turtlebot3c-nav[155]: The navigation components for the TurtleBot3
- Turtlebot3c-teleop[156]: The teleoperation components of the Turtlebot3

Additionally, for convenience, the robot should create its own hotspot so that we can connect to the robot and teleoperate it from our laptop. For that matter, we will also embed:

- Wifi-hotspot-config[157]: Configure the network manager for the hotspot
- Network-manager[158]: The network manager
- Avahi[159]: The Avahi daemon so we can ping by hostname and not IP address.

With all these snaps installed and configured in our image, after booting the SD card in the Raspberry Pi, we will be able to teleoperate the robot from a computer connected to the hotspot.

If you are looking for a more generic tutorial about Ubuntu Core image creation, and additional information, please visit: ubuntu.com/core/docs/build-an-image[160].

Create the assertion model

When building an Ubuntu Core image, the very first step is to write an assertion model.

The model is a recipe that describes the components that comprise a complete image. An assertion is provided as JSON in a text file.

See more: Assertion model[161]

[161] https://ubuntu.com/core/docs/reference/assertions

The model contains:

- Identification information, such as the developer-id and model name.
- Which essential snaps[162] make up the device system.
- Other required or optional snaps that implement the device functionalities.

[153] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-4
[154] https://snapcraft.io/turtlebot3c-bringup
[155] https://snapcraft.io/turtlebot3c-nav
[156] https://snapcraft.io/turtlebot3c-teleop
[157] https://snapcraft.io/wifi-hotspot-config
[158] https://snapcraft.io/network-manager
[159] https://snapcraft.io/avahi
[160] https://ubuntu.com/core/docs/build-an-image
[162] https://documentation.ubuntu.com/core/explanation/core-elements/snaps-in-ubuntu-core/

In our case, we will build a Raspberry Pi 4 image, and we will pick core20 as the version of the Ubuntu Core image. While the Ubuntu Core version can be chosen independently of the base version of the snaps, it will still include the corresponding version of the base snap. In this example, all of our snaps are using core20 since we are using ROS Noetic. We thus pick the corresponding Ubuntu Core image, core20, to spare the installation of a different core snap.

Metadata

First, we get the reference model for the Raspberry Pi 4. Provided as a reference, it will serve as a good starting point for our own model:

```
wget -O turtlebot3c-model.json https://raw.githubusercontent.com/snapcore/models/master/
ubuntu-core-20-pi-arm64.json
```

Open the file `turtlebot3c-model.json` and change the `model` field with the name `turtlebot3c-pi-arm64`:

```
-    "model": "ubuntu-core-20-pi-arm64",
+    "model": "turtlebot3c-pi-arm64",
```

In the same file, change the `authority` and `brand-id` with your own ID. It can be retrieved with:

```
snapcraft whoami
```

List of snaps

From the snaps already listed in the model, we leave the Raspberry Pi `gadget`, `pi-kernel`, `snapd` and `core20` snaps.

When adding snaps to the model, an ID is requested. This is the unique ID provided by the Snap Store[163]. We use the command `snap info` to get ID:

```
$ snap info core20
name:   core20
summary:   Runtime environment based on Ubuntu 20.04
publisher: Canonical✓
store-url: https://snapcraft.io/core20
contact:   https://github.com/snapcore/core20/issues
license:   unset
description: |
  The base snap based on the Ubuntu 20.04 release.
type:      base
snap-id:   DLqre5XGLbDqg9jPtiAhRRjDuPVa5X1q
tracking:  latest/stable
[...]
```

Here, the snap ID is `DLqre5XGLbDqg9jPtiAhRRjDuPVa5X1q`.

[163] https://snapcraft.io/store

TurtleBot3c snaps

Let's add the TurtleBot3c snaps to our `turtlebot3c-model.json`:

```
    {
    "name": "ros-noetic-robot",
    "type": "app",
    "default-channel": "latest/stable",
    "id": "hRIlEoo1gAoRqfWCrTmdNeioKT1DwQrn"
    },
   {
   "name":"turtlebot3c-bringup",
   "type":"app",
   "default-channel":"latest/beta",
   "id":"9QxPQq7N15yTrRVuy8s4NFCWmkZRjtOa"
   },
   {
   "name":"turtlebot3c-nav",
   "type":"app",
   "default-channel":"latest/beta",
   "id":"LJzR5xevdtU54wLeciVWXdhlKy1L4TrX"
   },
   {
   "name":"turtlebot3c-teleop",
   "type":"app",
   "default-channel":"latest/beta",
   "id":"m37jVvixxcn5YVD1t4wxfBKgx6nX4esy"
   }
```

Note that we've also added the `ros-noetic-robot` snap. We need it since we are using the TurtleBot3c ROS snap content-sharing approach[164] that depends on `ros-noetic-robot` to provide the ROS libraries.

Networking snaps

Since we want the TurtleBot3 to create a hotspot, we will need the `network-manager` snap as well as our `wifi-hotspot-config`[165] snap that will talk to the `network-manager` to create the hotspot on boot.

Additionally, for convenience, we also add `avahi`. This way, we will be able to ping the robot by hostname and not have to look for its IP address.

We add the networking snaps to the `turtlebot3c-model.json`:

```
{ "name": "network-manager",
  "type": "app",
  "default-channel": "20/stable",
  "id": "RmBXKl6HO6YOC2DE4G2q1JzWImC04EUy"
},
{ "name": "wifi-hotspot-config",
  "type": "app",
```

(continues on next page)

[164] https://canonical-robotics.readthedocs-hosted.com/en/latest/explanations/snaps/ros-architectures-with-snaps/#multi-snaps-using-content-sharing

[165] https://snapcraft.io/wifi-hotspot-config

(continued from previous page)

```
  "default-channel": "latest/beta",
  "id": "9s6xjXubw4a2yA0TE1aIyK90kewEzhYf"
},
{ "name": "avahi",
  "type": "app",
"default-channel": "latest/stable"
  "id": "dVK2PZeOLKA7vf1WPCap9F8luxTk9Oll"
}
```

The model now contains all the TurtleBot3c snaps as well as all the networking snaps. With all the applications that we need listed in the model, our model is ready.

Sign the model

To certify our model and modify it later on, we need to sign it.

This ensures the model cannot be altered without the key, and also links the created image to both the signed version of the model and an Ubuntu One account.

See more: Sign a model assertion[166]

[166] https://ubuntu.com/core/docs/sign-model-assertion

First, we check whether we have a key or not:

```
snapcraft list-keys
```

If not, we generate one:

```
snapcraft create-key turtlebot3c-key
```

Then we register it to the Snap Store with the details of our Ubuntu One[167] account:

```
snapcraft register-key turtlebot3c-key
```

The key should now appear when entering:

```
$ snapcraft list-keys

The following keys are available on this system:
    Name              SHA3-384 fingerprint
*   turtlebot3c-key   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

With the key ready, we update the timestamp entry in the `turtlebot3c-model.json`. To generate a valid timestamp at the present moment, use the command:

```
date -Iseconds --utc
```

Note that we generated the timestamp after creating the key which is mandatory.

And report the change in the `turtlebot3c-model.json`:

[167] https://login.ubuntu.com/

```
-"timestamp": "2020-03-31T12:00:00.0Z",
+"timestamp": "2024-08-06T12:22:26+00:00",
```

We can now sign the model with our key `turtlebot3c-key`:

```
snap sign -k turtlebot3c-key turtlebot3c-model.json > turtlebot3c.model
```

We now have our signed model: `turtlebot3c.model` that we generated from our model `turtlebot3c-model.json` signed with our private `turtlebot3c-key`.

Create our gadget snap

The gadget snap[168] is responsible for defining and configuring system properties specific to our device.

The gadget content can define the layout of the volumes of the device storage, default configuration options for snaps or even interface connections.

Thus, we will customise snap parameters as well as snap interface connections in our gadget snap for the TurtleBot3c snaps.

Get the template

Let us retrieve the template of the `pi-gadget` snap to kickstart our own:

```
git clone https://github.com/snapcore/pi-gadget.git -b 20-arm64
```

We picked the branch `20-arm64` to get the `core20` version.

In the `snapcraft.yaml` of the pi-gadget snap that we've just cloned, change the name to "turtlebot3-pi":

```
-name: pi
+name: turtlebot3c-pi
```

Note that at the end of the `snapcraft.yaml`, all the slots available are declared (serial, `spi`, `i2c`, etc.). For security consideration, we should remove the one that we don't need. However, we leave this as an exercise to the reader.

We can now proceed with the gadget connections customisation.

Connections

In our snaps, some plugs are auto-connect. For those who are not, we manually connect them in the `gadget` snap.

Similarly to the model, we must specify the snaps by their ID.

Append the following connections at the end of the `gadget.yaml` file to connect the various interfaces.

[168] https://ubuntu.com/core/docs/gadget-snaps

```
volumes:
  [...]
+connections:
+  # turtlebot3c-bringup
+  - plug: 9QxPQq7N15yTrRVuy8s4NFCWmkZRjtOa:raw-usb

+  # turtlebot3c-teleop
+  - plug: m37jVvixxcn5YVD1t4wxfBKgx6nX4esy:joystick

+  # wifi-hotspot-config
+  - plug: 9s6xjXubw4a2yA0TE1aIyK90kewEzhYf:network-manager
+  slot: RmBXKl6HO6YOC2DE4G2q1JzWImC04EUy:service
```

Doing so, all the interfaces of our snap stack are automatically plugged for this specific Ubuntu Core image.

Configurations

The WiFi hotspot snap requires some configurations. More specifically, we want to configure a custom `SSID` and password, as well as the WiFi interface.

Similarly, we refer to the snap by their unique ID.

In the `gadget.yaml`, we configure the hotspot with the `SSID` “turtlebot3c” and the password “turtlebot3c”:

```
connections:
  [...]
+defaults:
+  9s6xjXubw4a2yA0TE1aIyK90kewEzhYf:
+    ssid: "turtlebot3c"
+    password: "turtlebot3c"
+    wifi-interface: "wlan0"
```

With this, the `wifi-hotspot-config` snap will create a hotspot called “turtlebot3c” so we can connect to it.

Build the gadget snap

Building a gadget snap is no different from any other snap.

We build the snap with snapcraft:

```
snapcraft
```

This gadget snap can be built for ARM64 on AMD64, so you can call it on your AMD64 host.

We now have our gadget snap: `turtlebot3c-pi_20-1_arm64.snap`. Let’s integrate it in our model!

In the `turtlebot3-model.json`, replace the previous gadget snap with our custom one:

```
-    {
-        "name": "pi",
-        "type": "gadget",
-        "default-channel": "20/stable",
-        "id": "YbGa9O3dAXl88YLI6Y1bGG74pwBxZyKg"
-    },
+    {
+        "name": "turtlebot3c-pi",
+        "type": "gadget"
+    },
```

Here we don't define the `id` since we are providing the gadget snap locally and not from the Snap Store[169]. Additionally, because we use a local gadget snap that is unsigned, we can only mark the grade in the `turtlebot3c-model.yaml` as "`dangerous`":

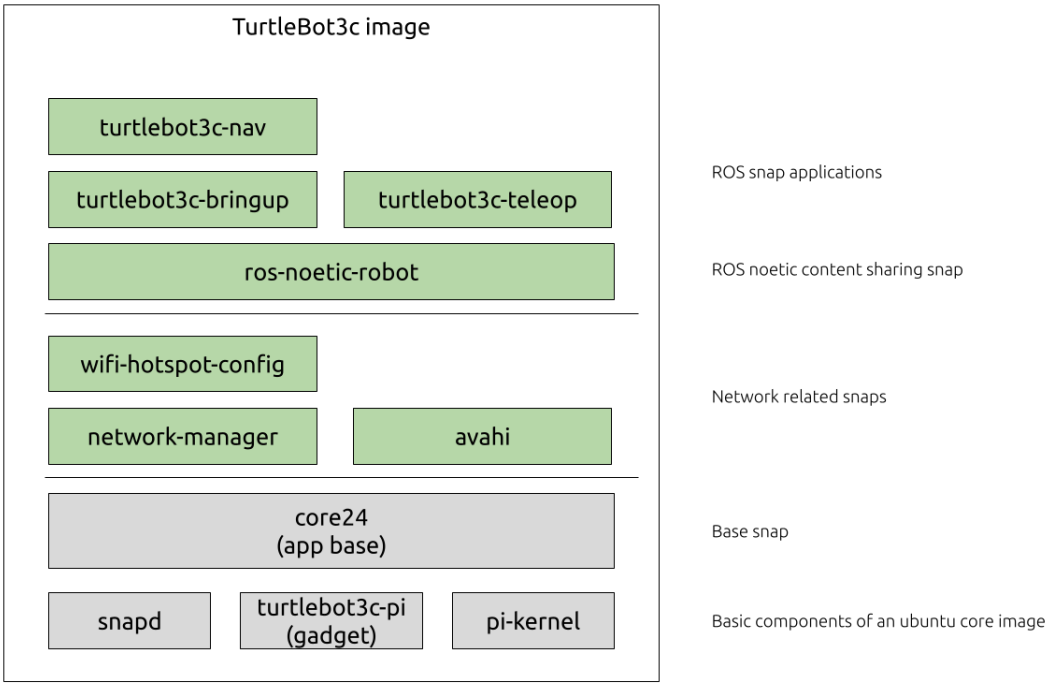
```
-grade: "strict"
+grade: "dangerous"
```

Since we modified the model, let's sign it again:

```
snap sign -k turtlebot3c-key turtlebot3c-model.json > turtlebot3c.model
```

The Ubuntu Core model is now fully customised for the TurtleBot3c.

As we see in the following diagram, we have in the image the basic `snapd`, `turltebot3c-pi` and `pi-kernel` snaps. Additionally, we have the `network-manager`, `avahi` and `wifi-hotspot-config` snaps to set up and manage the network. Finally, we have the `ros-noetic-robot` snap providing the ROS libraries to our `turtlebot3c-bringup`, `turtlebot3c-teleop` and `turtlebot3c-nav` snaps.



[169] https://snapcraft.io/store

Generating the image

To generate the image, we use the `ubuntu-image`[170] tool. It generates an image ready to be written on an SD card from the model that we prepared.

Install the tool as a snap:

```
snap install ubuntu-image
```

In our case, we provide the `turtlebot3c.model` that we signed as well as our local gadget snap:

```
ubuntu-image snap turtlebot3c.model --snap ./pi-gadget/turtlebot3c-pi_24-1_arm64.snap
```

This generates the `pi.img` file. The image file already contains all the snaps we specified in our model, as well as the configurations and connections from the gadget snap.

Write the image

To flash the image on the SD card, we use `rpi-imager`[171] which is conveniently distributed as a snap.

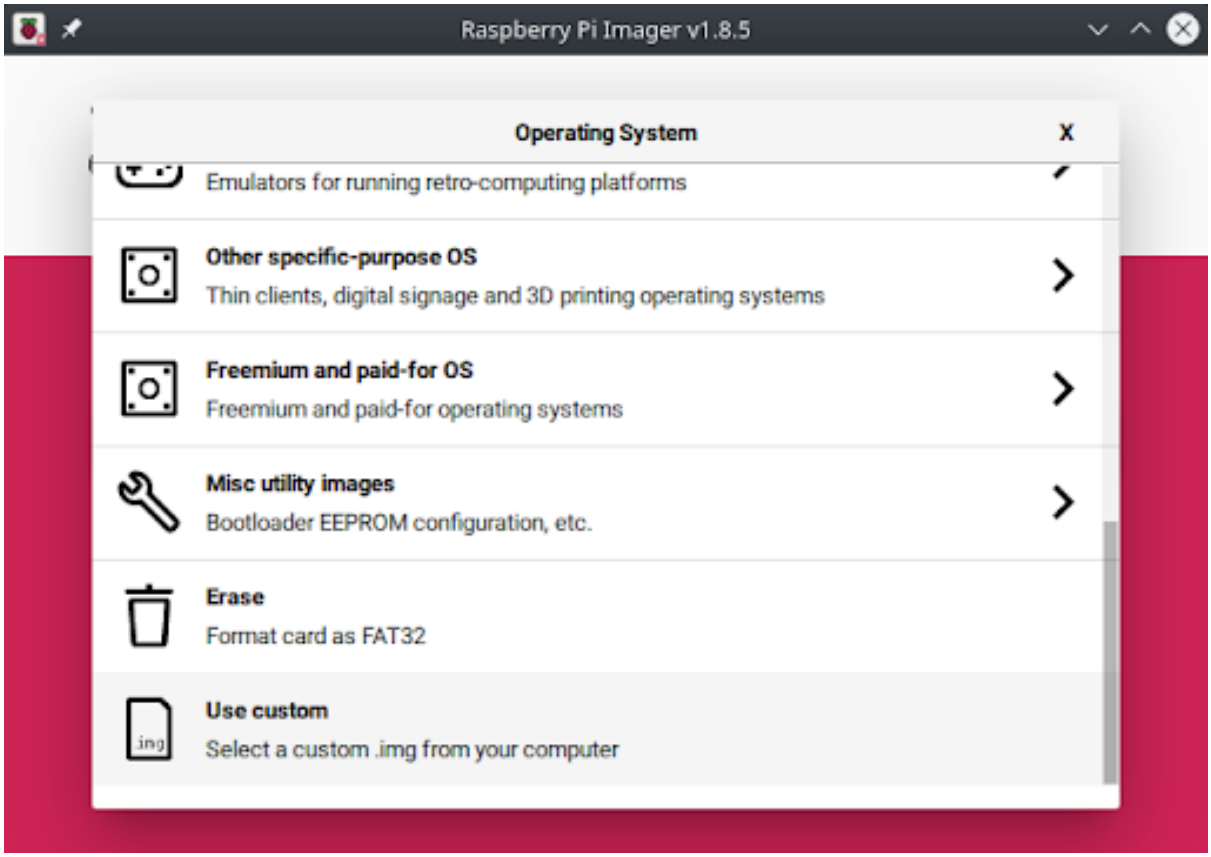
To install it:

```
snap install rpi-imager
```

Next, insert an SD card in your computer and start the `rpi-imager`.

We choose raspberry-pi 4 as the device. For the OS, go down the list and select “use custom”:

[170] https://snapcraft.io/ubuntu-image
[171] https://snapcraft.io/rpi-imager



Then select the `pi.img` file we've just created.

In the "choose storage", select your SD card.

Select no when asked if you want to apply custom OS settings, and yes when asked to clear the SD card.

This writing can take some time, and the `rpi-imager` will let you know once your SD card is ready.

First boot

Once the SD card is ready, we can insert it in the Raspberry Pi 4.

Ubuntu Core does not ship with a user by default and since we didn't create one explicitly, we won't be able to interact with the OS by plugging a screen and a keyboard, but we can still do some monitoring.

Monitoring of the first boot

In case we still want to monitor the logs and what's happening during the boot, we can connect a screen.

We can also connect a serial to USB adaptors for monitoring purposes.

To do so, connect the ground, `Rx` and `Tx` from your adaptor respectively to the pins 6, 8 and 10 of the Raspberry pi[172].

[172] https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#gpio

We use `picocom` to monitor the serial, we can install it with:

```
apt install picocom
```

You can now monitor the serial port with:

```
picocom /dev/ttyUSB0 -b 115200
```

Connect to the robot

The first boot creates the filesystem as well as decompresses the snaps and configures them. Meaning that the first boot can take up to 2 minutes.

Once finished, we can find the WiFi hotspot “turtlebot3c” available and connect to it with the password “turtlebot3c”.

Make sure to have `avahi-daemon` installed on your host machine and once connected to the WiFi hotspot, we can ping our robot with the command:

```
ping turtlebot3c.local
```

The TurtleBot3 is now ready to operate!

Teleoperate the robot

Now that we are connected to the robot, we can teleoperate it from our computer over the network.

First, we need to install `turtlebot3c-teleop` on our computer.

```
sudo snap install turtlebot3c-teleop
```

Make sure to disable the `core` application, since it’s already running on the robot.

```
sudo snap stop --disable turtlebot3c-teleop
```

We make our `ROS_MASTER_URI` point to the robot with:

```
export ROS_MASTER_URI=”http://turtlebot3c.local:11311”
```

after what, we can use the keyboard to teleoperate our robot with:

```
turtlebot3c-teleop.key
```

We have now tested our custom Ubuntu Core image with success.

Conclusion

When we booted the robot the first time no manual intervention was necessary, everything was planned in advance by the means of the gadget snap and the model. We have seen not only how to pre-install our application on an Ubuntu Core image, but also how to customise the image and our application for a specific use case. Ubuntu Core is meant for devices in production.

Thanks to the security and `sandboxing` features[173], Ubuntu Core is not only lightweight and customised but also secured, making it ready for an industrial application. Combined with landscape[174], it offers the perfect solution to deploy robot software at scale.

More generic documentation about Ubuntu Core can be found on ubuntu.com/core/docs[175].

[175] http://ubuntu.com/core/docs

- **Part 1: Packaging our first ROS application as a snap** **(page 2)**
  Learn to package an app as a snap. Across this tutorial, we will explore how to build snaps for a robotics application. Through different examples, we will cover the basics of snap creation for a ROS and ROS 2 application. By introducing the main concepts of a snap, we will see how to confine your robotics application and make it installable on dozens of Linux distributions.
- **Part 2: Packaging complex robotics software with snaps** **(page 18)**
  Learn to package a complete robot software stack as a snap. Across this tutorial, we will explore advanced snaps topics and tools that will show you how to structure, package, and test complex robotics applications. While covering the theoretical aspects of ROS snaps, we will apply all this knowledge to TurtleBot3 for a more interesting real-world scenario.
- **Part 3: Distribute ROS applications with the Snap Store** **(page 48)**
  Learn to distribute ROS applications to users or devices. Across this tutorial, we will explore the Snap Store to distribute robotics software and update like a global software vendor.
- **Part 4: Building ROS snaps with content sharing** **(page 56)**
  From the example shown in part 2 to exemplify one specific feature of snaps, content sharing. We will therefore revisit the entire example in order to make use of this feature.
- **Part 5: Create an Ubuntu Core image for the TurtleBot3** **(page 66)**
  From the example shown in part 2 we will learn to build a custom Ubuntu Core image for the TurtleBot3. We will go through all the steps to create a custom Ubuntu image from the snaps we've developed.

[173] https://ubuntu.com/core/docs/security-and-sandboxing
[174] https://documentation.ubuntu.com/landscape

See Also

- **Snap - Tutorials**[176]
  This section of documentation contains step-by-step tutorials to help outline what snap is capable of while helping you achieve specific aims, such as installing your favourite applications, taking a data snapshot, removing snaps.
- **Snapcraft - Tutorials**[177]
  This section of documentation contains step-by-step tutorials to help us learn how to build snap packages of our applications and services, for desktop, servers and embedded devices.
- **Ubuntu core - Tutorials**[178]
  This section of documentation contains step-by-step tutorials to help outline what Ubuntu Core is capable of while helping you achieve specific aims, such as installing Ubuntu Core or building a custom image for your device.

1.2. Monitor your robot fleet in the field

These tutorials detail how to **deploy and configure Canonical Observability Stack (COS) for Devices**.

1.2.1. Monitor your robot fleet in the field

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

These tutorials detail how to **deploy and configure Canonical Observability Stack (COS) for Devices**.

- *Deploy COS for robotics for Robotics server-side* (page 79)

  Deploy and set up the server-side of COS for robotics.

- *Deploy COS for robotics agent on your robot* (page 85)

  Install and configure COS for robotics agent on a robot.

[176] https://snapcraft.io/docs/snap-tutorials
[177] https://documentation.ubuntu.com/snapcraft/stable/tutorials/
[178] https://ubuntu.com/core/docs/tutorials

Deploy COS for robotics server in the cloud

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

Introduction

In this tutorial, we will walk through the process of deploying the **Canonical Observability Stack (COS) for robotics** on a cloud-based server. By the end of this tutorial, you will have a fully functional observability stack tailored for robotics, enabling you to monitor ROS devices efficiently.

COS for robotics is a lightweight, highly integrated observability stack designed to run on Kubernetes, offering a plug and play observability solution tailored for monitoring robotics devices. The server infrastructure integrates robotics-specific applications with the ones provided by COS-lite[179]. Moreover, it is designed with customization in mind. It offers the flexibility to add new applications in the form of charms or Open Container Initiative (`OCI`) images and enhance existing ones.

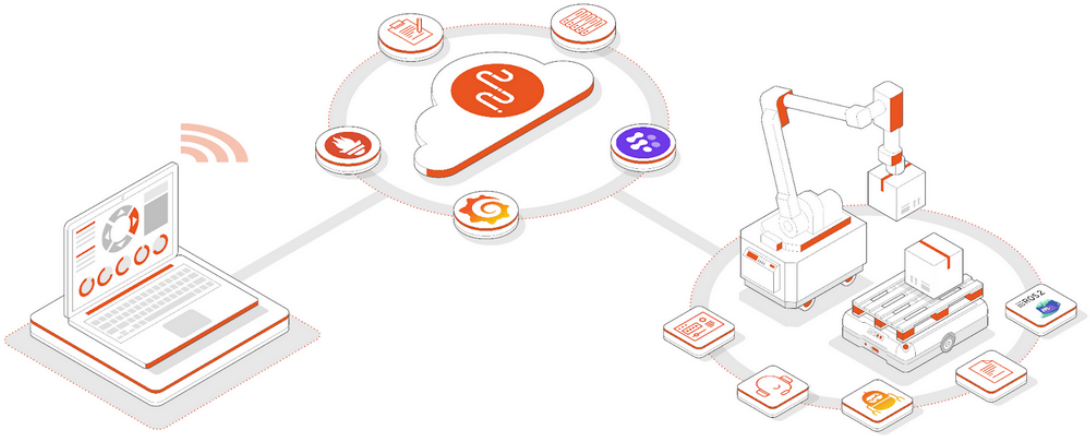
Note:

The server side is designed for the Edge and capable of running alongside `MicroK8s` and Juju with limited computing resources (around 8 GB of memory).

On the **server side**, COS for robotics runs a suite of observability tools that collect and process data from connected devices. On the **device side**, lightweight agents (packaged as **Snaps**) simplify device registration and enable real-time monitoring. This allows you to connect each robot in your fleet to the observability stack and immediately start collecting insights.

Each robot in your fleet can be set up with the snap agents, registered and observed, allowing for efficient management across an entire fleet.

[179] https://charmhub.io/topics/canonical-observability-stack/editions/lite



What you will learn

By following this tutorial, you will:

- Deploy COS for robotics on a cloud-based Kubernetes environment using **Juju** and **MicroK8s**.
- Register a ROS 2 device with the server.
- Begin monitoring robotic devices using **Prometheus, Grafana, Loki, and Foxglove Studio**.
- Understand how COS for robotics can be customized with additional applications and integrations.

Server side

The **COS for robotics** is a Juju-based observability stack running on **Kubernetes**. It includes the following key components:

- `Foxglove Studio`[180] – A visualization tool for robotics data.
- `COS-registration-server`[181] – Manages device registration.
- `Prometheus`[182] – Collects and stores metrics.
- `Loki`[183] – Handles logging for robotics devices.
- `Alert Manager`[184] – Manages alerts and notifications.
- `Blackbox exporter`[185] – Blackbox probing of endpoints.

[180] https://charmhub.io/foxglove-studio-k8s
[181] https://charmhub.io/cos-registration-server-k8s
[182] https://charmhub.io/prometheus-k8s
[183] https://charmhub.io/loki-k8s
[184] https://charmhub.io/alertmanager-k8s
[185] https://charmhub.io/blackbox-exporter-k8s

- `Grafana`[186] – Provides dashboards for visualization.

These components are at core of the stack and can be enhanced with additional functionalities and applications.

In the next section, we will go step by step through the deployment process.

Install prerequisites

Important:

To follow this tutorial, you will need a machine or a VM with at least 8 GB of memory, 4 CPUs and 50 GB of storage. A container won't be sufficient.

Let's proceed with the installation.

1. Install `MicroK8s`

Install the `microk8s` snap with:

```
sudo snap install microk8s --channel 1.35-strict
```

Add the user to the `microk8s` group for unprivileged access and give use permission to read the `~/.kube` director:

```
sudo adduser $USER snap_microk8s
sudo chown -f -R $USER ~/.kube
```

Wait for `microk8s` to finish initializing with:

```
sudo microk8s status --wait-ready
```

Enable the storage and `dns addons` which are required for the Juju controller:

```
sudo microk8s enable hostpath-storage dns
```

Finally, ensure your new group membership is apparent in the current terminal (not required once you have logged out and back in again):

```
newgrp snap_microk8s
```

[186] https://charmhub.io/grafana-k8s

2. Install Juju

Install the Juju snap with:

```
sudo snap install juju --channel 3.6/stable
```

Since the Juju package is strictly confined, you also need to manually create a path:

```
mkdir -p ~/.local/share
```

Now bootstrap a Juju controller into your `MicroK8s`

```
juju bootstrap microk8s cos-robotics-controller
```

If successful the terminal will show the following message:

```
Bootstrap complete, controller "cos-robotics-controller" is now available in namespace
"controller-cos-robotics-controller"
```

3. Configure and enable `Metallb`

The bundle comes with `Traefik` to provide ingress, for which the `metallb` add-on must be enabled. `Metallb`[187] provides load balancer functionality and requires the source IP address of the host system for outbound connections. Run the following command to retrieve the IP address:

```
sudo apt update && sudo apt install -y jq
IPADDR=$(ip -4 -j route get 2.2.2.2 | jq -r '.[] | .prefsrc')
```

Then, enable `metallb` with the following command:

```
sudo microk8s enable metallb:$IPADDR-$IPADDR
```

Deploy the COS for robotics bundle

The stack deployment relies on the infrastructure as code (IAC) tool Terraform[188]. It allows to easily set up complex infrastructure from a YAML based recipe, allowing for simplicity, re-usability & repeatability among (many) other things.

You can install Terraform from the Store[189] with:

```
sudo snap install terraform --classic
```

First, let us retrieve the Terraform plan:

```
git clone --branch track/0 https://github.com/canonical/rob-cos-overlay.git
cd rob-cos-overlay/terraform/rob-cos
```

Then we have to initialize the project:

[187] https://metallb.universe.tf/
[188] https://developer.hashicorp.com/terraform
[189] https://snapcraft.io/terraform

```
terraform init
```

In order to deploy COS for robotics, we create a dedicated model with the following:

```
juju add-model cos-robotics-model
juju switch cos-robotics-model
```

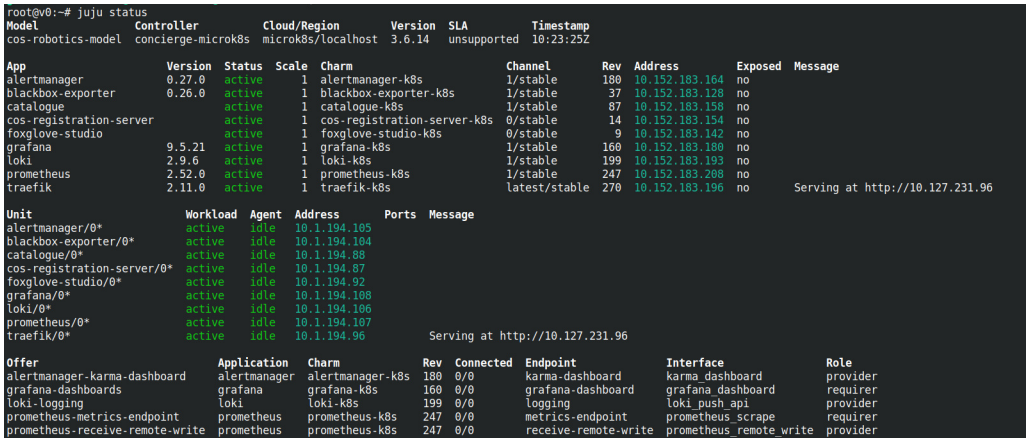
Finally, deploy it with:

```
terraform apply -var="model=cos-robotics-model"
```

When prompted, type “yes” to confirm. Now you can sit back and watch the deployment take place:

```
juju status --watch 5s --color --relations
```

COS will be ready to use when the `juju status` shows all the machines active and the agents idle as follow:



Now COS for robotics is good to go: you can register devices to it to begin the monitoring!

Verify the deployment

When all the charms are deployed, you can head over to browse their built-in web user interfaces. You can find out their addresses from the `show-proxied-endpoints`[190] `Traefik` action. In your terminal type:

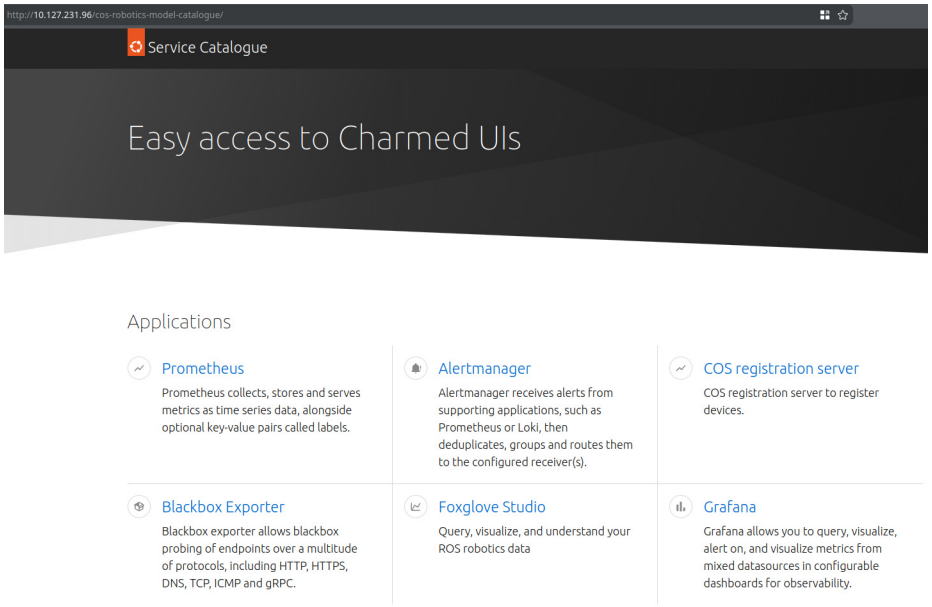
```
juju run traefik/0 show-proxied-endpoints
```

The catalogue endpoint can be visualized on your browser and it will list the catalogue of applications offered by COS for robotics. From the `proxied` endpoints, the catalogue URL should be similar to:

```
"catalogue":{"url": "http://<cos-robotics-server-ip>/cos-robotics-model-catalogue"}
```

Now by navigating to the catalogue URL in your browser, the catalogue of all the available application will be displayed:

[190] https://charmhub.io/traefik-k8s/actions



Grafana login

Clicking on the **Grafana** application will prompt you for username and password as follows:

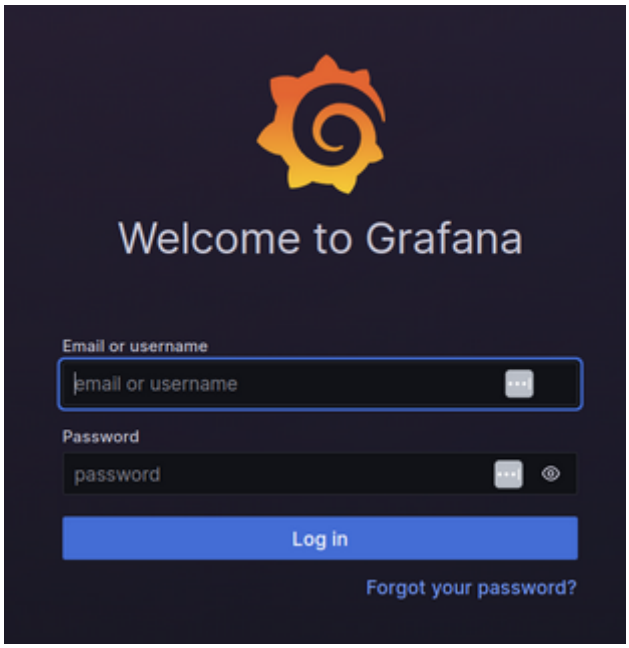


Fig. 1: Grafana login page

The default password for Grafana is automatically generated for every installation. To access `Grafana's` web interface, use the username `admin`, and the password obtained from the get-admin-password[191] action as follows:

```
juju run grafana/0 get-admin-password
```

[191] https://charmhub.io/grafana-k8s/actions

Next steps: device setup

Now that the server is set up, let's see how to deploy and register a device for monitoring.

> **Note**: The device setup is covered in a the next tutorial. You can find it at *Deploy COS for robotics agent on your robot* (page 85).

Deploy COS for robotics agent on your robot

> **Warning:**
>
> **Beta Notice**: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

Overview

Requirements: Before starting this tutorial, make sure to have the server side working *from the previous tutorial* (page 79)

In order for a device to register and interact with the COS registration server and its applications the following snaps have to be installed:

- `rob-cos-demo-configuration`[192]: contains the configuration of the robot.
- `cos-registration-agent`[193]: responsible for registering the robot on the `COS-registration-server`[194] as well as uploading robot specific data to the server (dashboard, foxglove layouts, `UID`, etc).
- `ros2-exporter-agent`[195]: responsible for recording data on the robot and syncing them to the `Ros2BagFileserver`[196].
- `grafana-agent`[197]: responsible for sending metrics, logs, and trace data to the Grafana charm.
- `foxglove-bridge`[198]: bridge to visualize live ROS data via the Foxglove websocket connection.
- `rob-cos-data-sharing`[199]: data sharing snap for on device cos robotics snaps.

[192] https://snapcraft.io/rob-cos-demo-configuration
[193] https://snapcraft.io/cos-registration-agent
[194] https://charmhub.io/cos-registration-server-k8s
[195] https://snapcraft.io/ros2-exporter-agent
[196] https://charmhub.io/ros2bag-fileserver-k8s
[197] https://snapcraft.io/grafana-agent
[198] https://snapcraft.io/foxglove-bridge
[199] https://snapcraft.io/rob-cos-data-sharing

Verify connectivity

Before diving into the device setup, let's ensure that the device can reach the server on the network. To connect devices across different networks, a VPN between the robots and the server could be used but is not mandatory. Let's do so by initiating a `curl` from the device to the server:

```
curl http://<cos-robotics-server-ip>/cos-robotics-model-cos-registration-server/api/v1/
health
```

If the command returns without errors, connectivity is correct.

Make sure the request works from the device to the server, otherwise the rest of the guide cannot be executed. Now it's time to set up and register the device!

Installation

A convenience script has been created to install all the required snaps on the device. Download the script as follows:

```
curl -L https://raw.githubusercontent.com/canonical/rob-cos-device-setup/track/0/setup-
robcos-device.sh -O
```

And run it with:

```
sudo bash setup-robcos-device.sh
```

The script will initiate prompts for the robot `UID` and the server URL. While the robot `UID` is optional, the URL is mandatory, serving as the designated address for the server where the device registration occurs. The queries and response will look as follows:

```
Please enter the device-uid:
robot_1
```

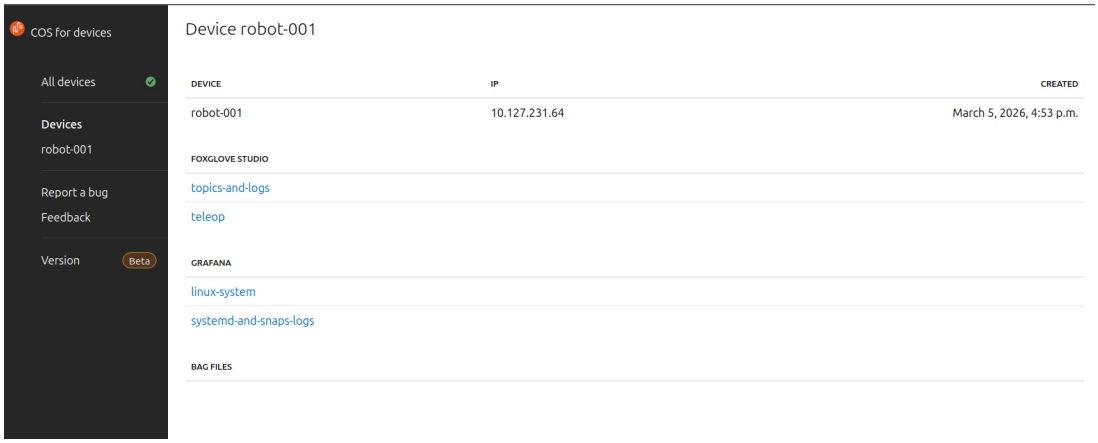
```
Please enter the rob-cos-server-url:
http://<cos-robotics-server-ip>/cos-robotics-model
```

The script will now proceed with the installation of all the required snaps. Upon completion, the device and its corresponding dashboards will be registered and available for visualization on the COS for robotics server.

Verify Installation

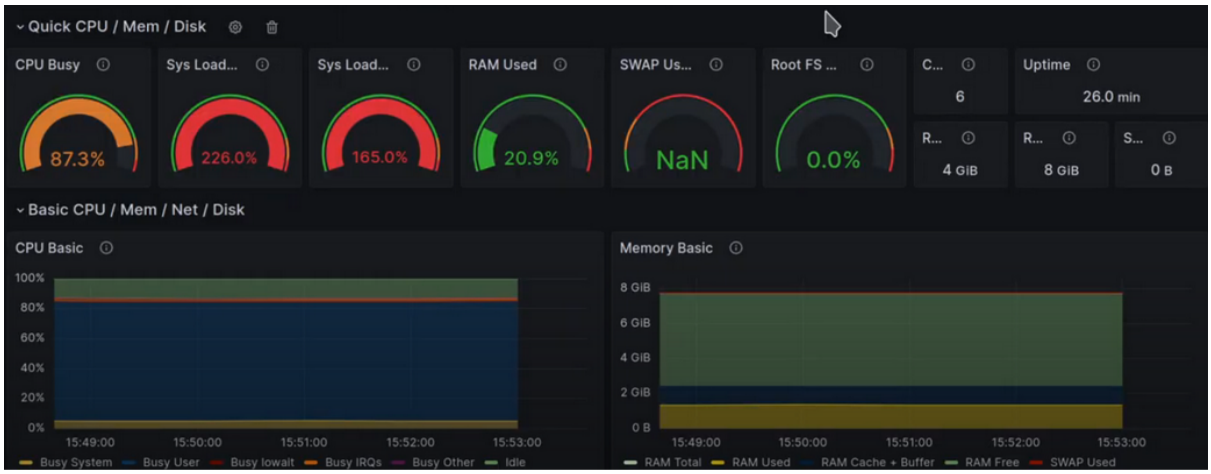
Now let's verify that the device has been correctly configured and registered. On the browser, access the catalogue and click on the COS-registration-server app. The registered robot should be now available in the devices list:

By clicking on the robot `UID`, a page will open, displaying all the links to the robot's data:



From this page, each link will redirect you to the corresponding dashboard for the specific data category and device, ensuring easy and intuitive visualization.

An example visualization of Grafana linux-system dashboard is provided below:



This is it, now your device is registered and being correctly monitored via COS for robotics!

If you want to start storing ROS 2 bags, check the following How-to guide:

- *Host a basic file server for your rosbags* (page 141)

2. How-to guides

How-to guides to achieve specific goals with Canonical's robotics stack.

2.1. Packaging

This section includes all guides related to packaging robotics applications using snaps.

2.1.1. Packaging

This section includes all guides related to packaging robotics applications using snaps.

Build and publish a snap with GitHub Actions

When deploying a robotics application with snap[200], keeping the deployment synchronised with development progress is a high priority. It's best accomplished with a CI/CD pipeline that will automatically deploy your latest developments to your devices.

In this post, we will explore how to automatically build and publish your ROS snap using GitHub Actions[201], so that your snap always stays up-to-date on the snapstore[202].

For this example, we will use a simple ROS2 Foxy application publishing some mock data, whose source code can be found on GitHub[203]. Of course, the same principle can be applied to a ROS package or another release of ROS 2.

Adding a Snap GitHub Action

Our example project is called *snapped_ros2_pkg* and it has a typical ROS 2 package structure:

```
.
├── CMakeLists.txt # Compilation instructions
├── config
│   └── config.yaml # Configuration file
├── launch
│   └── snapped.launch.py # Python ROS 2 launchfile
├── package.xml # Dependencies list
├── README.md
├── snap
│   └── snapcraft.yaml # Snapcraft entry point
└── src
    └── snapped_ros2_pkg_node.cpp # Source of the node publishing the mock data
```

GitHub's workflows are located in `.github/workflows/` in our project tree:

[200] https://ubuntu.com/robotics/docs
[201] https://docs.github.com/en/actions
[202] https://snapcraft.io/store
[203] https://github.com/ubuntu-robotics/ros_snap_github_action

```
├── .github
   └── workflows
      └── snap.yaml
```

The file `snap.yaml` will contain our GitHub workflow dedicated to snap. The GitHub workflow will first build the snap before testing it and publishing it to the snap store.

Here is the content of the `snap.yaml`. Don't worry, we'll break this down:

```yaml
name: snap
on:
  push:
    tags:
      - '*'
    branches:
      - main
  pull_request:
    branches:
      - main
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      snap-file: ${{ steps.build-snap.outputs.snap }}
    steps:
    - uses: actions/checkout@v3
    - uses: snapcore/action-build@v1
      id: build-snap

    # Make sure the snap is installable
    - run: |
        sudo snap install --dangerous ${{ steps.build-snap.outputs.snap }}
    # Do some testing with the snap
    - run: |
        gbeuzeboc-snapped-ros2-pkg.snapped-ros2-launch --print-description
    - uses: actions/upload-artifact@v3
      with:
        name: gbeuzeboc-snapped-ros2-pkg
        path: ${{ steps.build-snap.outputs.snap }}

  publish:
    if: github.ref == 'refs/heads/main' || startsWith(github.ref, 'refs/tags/')
    runs-on: ubuntu-latest
    needs: build
    steps:
    - uses: actions/download-artifact@v3
      with:
        name: gbeuzeboc-snapped-ros2-pkg
        path: .
    - uses: snapcore/action-publish@v1
      env:
        SNAPCRAFT_STORE_CREDENTIALS: ${{ secrets.STORE_LOGIN }}
      with:
        snap: ${{needs.build.outputs.snap-file}}
        release: ${{ startsWith(github.ref, 'refs/tags/') && 'candidate' || 'edge'}}
```

This workflow is already in use, and you can see the results at the ubuntu-robotics/ros_snap_github_action[204] repository.

Workflow conditions

A GitHub workflow file starts with its name and the run conditions.

In our case, we will only trigger the workflow on changes on the branch `main`, a new tag, as well as when a pull request is opened against `main`. We additionally added a condition (`workflow_dispatch`) to be able to trigger it manually.

```yaml
name: snap
on: # The GitHub Action will happen when:
  push:
    tags:
      - '*' # When a new tag is created
    branches:
      - main # When a commit was created on main
  pull_request:
    branches:
      - main # When a pull request is created on main
  Workflow_dispatch: # When we trigger it manually via the API or the web interface
```

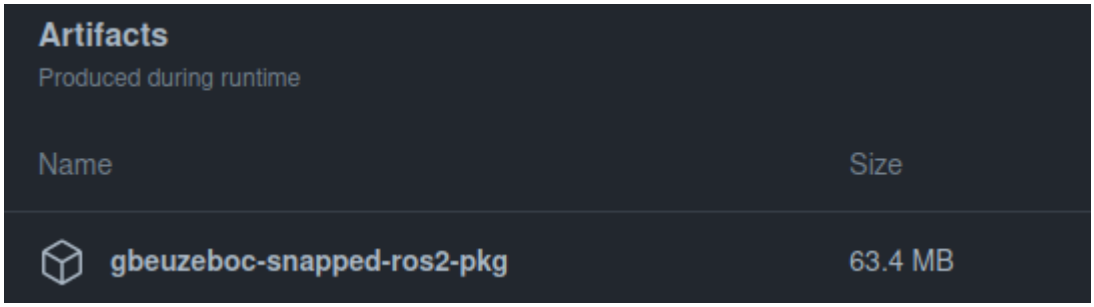
For more information about triggering a GitHub workflow, see, "on" documentation[205].

Workflow jobs

In our workflow, we will define two distinct jobs[206]. One for building the snap and another for publishing the snap. The idea is that we want to build the snap every time we run the action, but only publish under certain conditions.

Build job

Before listing the steps of our build, we will define the environment and the output of our build. The output will be used to transfer the result of the build job to the `publish` job. This is particularly helpful since you can download the artifact - the snap - from the build web page.



[204] https://github.com/ubuntu-robotics/ros_snap_github_action/actions
[205] https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#on
[206] https://docs.github.com/en/enterprise-server@3.3/actions/using-jobs/using-jobs-in-a-workflow

```
build:
  runs-on: ubuntu-latest
  outputs:
    snap-file: ${{ steps.build-snap.outputs.snap }}
```

Here we use the variable `steps.build-snap.outputs.snap` that will be defined by a step right after.

For more information about the type of machine to run the job on and outputs, see, `runs-on`[207] and `outputs`[208].

Checkout step

GitHub Action comes with reusable actions for CI/CD provided by GitHub and the community! We are going to use some common ones as well as some that are specific to snap.

The first step is to checkout our current code using the checkout workflow[209].

```
- uses: actions/checkout@v2
```

Build the snap

In order to build our snap, we will use the snapcore/action-build[210] action.

```
- uses: snapcore/action-build@v1
  id: build-snap
```

This will generate our `.snap` file and store its name in the variable `steps.build-snap.outputs.snap`.

Test our snap

Now we should make sure that our snap is installable and can run.

```
# Make sure the snap is installable
- run: |
    sudo snap install --dangerous ${{ steps.build-snap.outputs.snap }}
# Do some testing with the snap
- run: |
    gbeuzeboc-snapped-ros2-pkg.snapped-ros2-launch --print-description
```

Here we are reusing the variable `steps.build-snap.outputs.snap` to get our snap name and feed it to the snap install command. Then we simply run the `launchfile` accompanying our

[207] https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idruns-on
[208] https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idoutputs
[209] https://github.com/marketplace/actions/checkout
[210] https://github.com/snapcore/action-build

ROS 2 application with the `--print-description` option, which only prints the launch description to the console. Which gives us a good indication that the call to our application is fine. Of course, more extensive testing could and should be implemented.

Upload our artifact

Now that our snap is generated and tested, we will upload it as an artifact. This is useful, so that we can download it from the GitHub web page to test it locally. We will also make use of it for the second part of our workflow: publish.

```
- uses: actions/upload-artifact@v3
   with:
     name: gbeuzeboc-snapped-ros2-pkg
     path: ${{ steps.build-snap.outputs.snap }}
```

We are now using the actions/upload-artifact[211]. Note that before the steps, we defined snap-file: `${{ steps.build-snap.outputs.snap }}` in the output section. Later, we will refer to our uploaded artifact (our snap) as `snap-file`.

Publish job

This job will be executed only under certain conditions. We only want to publish our snap when there are changes on the `main` branch or when we create a new tag. Furthermore, this job can only run when the 'build' job is successfully done.

```
if: github.ref == 'refs/heads/main' || startsWith(github.ref, 'refs/tags/') # main branch
or tagged version
runs-on: ubuntu-latest
needs: build # Wait for build job to be done
```

By adding the `build` job in the needs, this job can only happen if the `build` job succeed. We will also be able to refer to the uploaded artifact of the `build` job.

Download the artifact

First, we need to get the previously uploaded artifact (our `.snap` file) with the actions/download-artifact[212] action.

```
- uses: actions/download-artifact@v3
    with:
      name: gbeuzeboc-snapped-ros2-pkg
      path: .
```

We explicitly specify `path: .` so that our artifact is stored at the root of our directory, without any extra directory.

[211] https://github.com/marketplace/actions/upload-a-build-artifact
[212] https://github.com/marketplace/actions/download-a-build-artifact

Publish the snap

Finally, we are publishing our snap with the snapcore/action-publish[213] action. You will have to identify to the snapstore[214] in order to publish your snap. Here we refer to the secret `STORE_LOGIN` (set in the next section). The snap to upload is referred to by the output of the build job.

Additionally, we want to release to different risk levels[215] depending on the situation. If it's a change on the `main` branch we would like to publish to the edge risk level, so edge always has the latest changes which may not be considered as stable. On the other hand, if we are on a tagged version, we would like to publish to the `candidate` risk level. After thorough testing, the maintainer can manually promote the snap to `beta/stable`.

```
- uses: snapcore/action-publish@v1
    env:
      SNAPCRAFT_STORE_CREDENTIALS: ${{ secrets.STORE_LOGIN }}
    with:
      snap: ${{needs.build.outputs.snap-file}}
      release: ${{ startsWith(github.ref, 'refs/tags/') && 'candidate' || 'edge'}}
```

Setting the secret

The `action-publish` is using the secret `STORE_LOGIN` so let us define it.

Generate the secret

Open your terminal and enter:

```
snapcraft export-login --snaps=gbeuzeboc-snapped-ros2-pkg --acls package_access,package_
push,package_update,package_release exported.txt
```

Make sure to adjust the `--snaps=` to your snap name.

This will prompt you to login and will then generate an `exported.txt` file. The content of this file is secret.

Add the secret to your GitHub repository

To add the secret to the project, we will need to:

- Go to the "Settings" tab of the repository
- Choose "Secrets" from the menu on the left and then "Actions"
- Click on "New repository secret".
- Setting the name of the secret as `STORE_LOGIN`, and paste the contents of `exported.txt` as the value.

[213] https://github.com/snapcore/action-publish
[214] https://snapcraft.io/store
[215] https://snapcraft.io/docs/channels

We are all good to go now. The actions can be checked on the Actions tab[216] of the project. And the published snap can be seen on the snapstore page[217].

You can of course create a more complex release behaviour, making use of all the channels[218] features of the snapstore.

A real world example

While the project presented was just an example, one can find real world use of a ROS snap CI/CD workflow. A great example is the ROS snap examples[219] repository. The ROS snap examples repository hosts multiple examples and CIs. With the snap GitHub workflow[220] updating the available snap[221], users can stay up to date on the latest version of the talker-listener.

Summary

We have explored how to build and publish a snap with GitHub Actions. This allows you to easily keep your deployment in sync with your development. With channels, providing simultaneously different versions of your software is made easy. Distributing your application with snap allows you to easily benefit from a production-grade infrastructure while offering a seamless experience to your users.

Debug Snap Applications

Important:

Before you start

1. This how-to guide builds on concepts from *Tutorial 2: Packaging complex robotics software with snaps* (page 18). While it's strongly advised to have followed this tutorial before starting, it is not mandatory to proceed with this guide.
2. This guide assumes you have a basic understanding of snaps. If you are new to snaps, you can learn more about them in the Snap documentation[222].

[222] https://snapcraft.io/docs

When developing a snap, things can go wrong. In this how-to guide, we will explore some common ways to debug a snap.

Starting from the results of *Tutorial 2: Packaging complex robotics software with snaps* (page 18), we will now use a modified source code with intentional errors to learn how to debug and fix issues in our snaps.

[216] https://github.com/ubuntu-robotics/ros_snap_github_action/actions
[217] https://snapcraft.io/gbeuzeboc-snapped-ros2-pkg
[218] https://snapcraft.io/docs/channels
[219] https://github.com/ubuntu-robotics/ros-snaps-examples
[220] https://github.com/ubuntu-robotics/ros-snaps-examples/actions/workflows/example_snap_github_action.yaml
[221] https://snapcraft.io/ros2-humble-talker-listener

The first step is to clone the specific branch containing the modified source code as described in the `setup` section below.

Since the build of the snap itself can go wrong. `Snapcraft` offers multiple ways to introspect the instance state and files. The Explanation Documentation: *Debug snap build*[223], is a great place to learn about this. When the build went right and this is the run-time that is causing issues, snap also offers multiple ways to debug it. The Explanation Documentation: *Debug a snap application*[224], is a great place to learn about this.

Setup

Clone the `debugging-tutorial` branch of the turtlebot3c-snap[225] repository. This branch contains the modified source code with intentional errors.

```
git clone -b debugging-tutorial https://github.com/canonical/turtlebot3c-snap.git
cd turtlebot3c-snap
```

Your folder should have the following structure:

```
.
├── README.md
├── renovate.json
├── snap
│   ├── hooks
│   │   ├── configure
│   │   └── install
│   ├── local
│   │   ├── core_launcher.sh
│   │   ├── install_last_map.sh
│   │   ├── mapping_launcher.sh
│   │   ├── mux_select_joy_vel.sh
│   │   ├── mux_select_key_vel.sh
│   │   ├── mux_select_nav_vel.sh
│   │   ├── navigation_launcher.sh
│   │   ├── ros_network.sh
│   │   └── save_map.sh
│   └── snapcraft.yaml
└── turtlebot3c.rosinstall
```

Debug the core application

Testing our TurtleBot3 snap will be the perfect opportunity to apply the different debugging approaches.

Before performing any test we need our actual snap. Although now it should be clear how to build our snap, let's see how.

[223] https://ubuntu.com/robotics/docs/debug-the-build-of-a-snap
[224] https://ubuntu.com/robotics/docs/debug-a-snap-application
[225] https://github.com/canonical/turtlebot3c-snap/tree/debugging-tutorial

First run of our snap

To build our snap, we must be located at the root of our directory next to our `snap/` folder. Then we simply run the command:

```
snapcraft
```

This will take some time but after some time we must get the file `turtlebot3c_*.snap`.

Install this snap with the following command:

```
sudo snap install turtlebot3c_*.snap --dangerous
```

And that's it. Our snap is installed and since our `core` and `teleop` apps are daemons, they must be already running! Let's inspect the logs to see what is going on.

```
sudo snap logs turtlebot3c.core -n 100
```

And we get the following output:

```
turtlebot3c.core[88370]: </font>RLException: while processing /snap/turtlebot3c/x1/opt/
ros/noetic/share/turtlebot3_bringup/launch/turtlebot3_robot.launch:
turtlebot3c.core[88370]: while processing /snap/turtlebot3c/x1/opt/ros/noetic/share/
turtlebot3_bringup/launch/turtlebot3_lidar.launch:
turtlebot3c.core[88370]: Invalid tag: environment variable 'LDS_MODEL' is not set.
turtlebot3c.core[88370]: Arg xml is
turtlebot3c.core[88370]: The traceback for the exception was written to the log file

turtlebot3c.core[88370]: ... logging to /root/snap/turtlebot3c/x1/ros/log/fdf740ce-b920-
11ed-a4a0-e5f11893ed73/roslaunch-workshop-part2-88370.log
turtlebot3c.core[88370]: Checking log directory for disk usage. This may take a while.
turtlebot3c.core[88370]: Press Ctrl-C to interrupt
turtlebot3c.core[88370]: Done checking log file disk usage. Usage is <1GB.
Turtlebot3c.core[88370]:
systemd[1]: snap.turtlebot3c.core.service: Main process exited, code=exited, status=1/
FAILURE
systemd[1]: snap.turtlebot3c.core.service: Failed with result 'exit-code'.
systemd[1]: snap.turtlebot3c.core.service: Scheduled restart job, restart counter is at 5.
systemd[1]: Stopped Service for snap application turtlebot3c.core.
systemd[1]: snap.turtlebot3c.core.service: Start request repeated too quickly.
systemd[1]: snap.turtlebot3c.core.service: Failed with result 'exit-code'.
systemd[1]: Failed to start Service for snap application turtlebot3c.core.
```

The `environment variable 'LDS_MODEL' is not set.` is the guilty part. The LDS model is the model of LIDAR that is used by the TurtleBot3. The TurtleBot3 can work with two different LIDAR models, `LDS-01` and `LDS-02`. The LIDAR model is read directly from an environment variable by the launch file[226]. So we will have to read the snap configuration and simply export it as an environment variable. We will need an `lds-model` configuration.

We already set two configurations so nothing new here. Let's add this additional configuration.

[226] https://github.com/ROBOTIS-GIT/turtlebot3/blob/noetic/turtlebot3_bringup/launch/turtlebot3_lidar.launch#L3

LIDAR configuration

We will first add our configuration to our different hooks, and then use the configuration in our `core_launcher.sh` script.

Hooks

First, let's define the default value for our configuration `lds-model`. Let's add our configuration to the `snap/hooks/install` file:

```
snapctl set simulation=false
+# set default lidar model for real robot
+snapctl set lds-model=LDS-01
```

Similarly to the simulation configuration we will also complete the `snap/hooks/configure`:

```
+LDS_MODEL="$(snapctl get lds-model)"
+case "$LDS_MODEL" in
+  "LDS-01") ;;
+  "LDS-02") ;;
+  *)
+    >&2 echo "'$LDS_MODEL' is not a supported value for lds_model." \ "Possible values
are LDS-01 and LDS-02"
+    return 1
+    ;;
+esac
# restart core and teleop on new config
snapctl stop "$SNAP_INSTANCE_NAME.core"
snapctl stop "$SNAP_INSTANCE_NAME.teleop"
```

Now, our hooks are correctly handling the `lds-model` configuration. Nothing new here; we applied the same method as for the `turtlebot3-model` configuration.

Use the configuration

Now let's use the configuration within our TurtleBot3 snap.

Let's do so by modifying the file `snap/local/core_launcher.sh`:

```
#!/usr/bin/bash
[...]
TURTLEBOT3_MODEL="$(snapctl get turtlebot3-model)"
+LDS_MODEL="$(snapctl get lds-model)"
[...]
+export LDS_MODEL
${SNAP}/opt/ros/noetic/bin/roslaunch turtlebot3c_bringup turtlebot3c_bringup.launch
simulation:=$SIMULATION
```

Now, when the `core` daemon is going to start it will use the `lds-model` configuration in the launch file.

Since we already installed our snap, we must set a value manually here:

```
sudo snap set turtlebot3c lds-model=LDS-02
```

We can now rebuild the snap and reinstall it. Once reinstalled we can now set the proper configuration:

```
sudo snap set turtlebot3c lds-model=LDS-01
```

`Roscore` tend to stay alive sometime, so make sure no `roscore` is running when you have stopped the snap. `roscore` zombies create unsupported behaviour.

Teleoperate the Turtlebot3

We just fixed the previous issue regarding the `lds-model`. It's now time to see if the `core` daemon is now running properly.

Let's check again the logs now that we fixed the LIDAR model issue:

```
systemd[1]: Started Service for snap application turtlebot3c.core.
turtlebot3c.core[101438]: [ERROR] [1677843362.232445761]: An exception was thrown: open:
No such file or directory
turtlebot3c.core[101437]: [ERROR] [1677843362.654190]: Error opening serial: [Errno 2]
could not open port /dev/ttyACM0: [Errno 2] No such file or directory: '/dev/ttyACM0'
```

Our application cannot access the USB port. There are multiple reasons for that.

First, we declared the `raw-usb` interface but didn't connect it. We can verify that with:

```
$ snap connections turtlebot3c

Interface  Plug          Slot   Notes
Joystick  turtlebot3c:joystick    -    -
network   turtlebot3c:network    :network   -
network-bind  turtlebot3c:network-bind  :network-bind -
raw-usb      turtlebot3c:raw-usb    -       -
```

The `raw-usb` plug has no slot assigned. This is because it's not connected. `raw-usb` is not auto-connect, so it has to be connected manually. We can connect it with:

```
sudo snap connect turtlebot3c:raw-usb
```

Not that once you have connected a `plug` on a device for a given snap you won't have to do it again (even over the updates). And we can check the result with the `snap connections` command:

```
$ snap connections turtlebot3c

Interface  Plug          Slot     Notes
Joystick  turtlebot3c:joystick    -      -
network   turtlebot3c:network    :network     -
network-bind  turtlebot3c:network-bind  :network-bind   -
raw-usb      turtlebot3c:raw-usb       :raw-usb    manual
```

We can restart our application with the `sudo snap restart turtlebot3c.core` command and check the logs. Unfortunately, we will see the error again, because we are not running on the real robot. We must change the snap configuration simulation to run the simulation mode.

```
sudo snap set turtlebot3c simulation=true
```

Now the logs look better:

```
systemd[1]: Started Service for snap application turtlebot3c.core.
turtlebot3c.core[102917]: xacro: in-order processing became default in ROS Melodic. You
can drop the option.
```

We can now launch the simulation with:

```
TURTLEBOT3_MODEL=waffle_pi roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

And in another terminal, we can teleoperate our robot with:

```
turtlebot3c.key
```

Alternatively, if we have a joystick we can test the `turtelbot3c.joy`.

Since our key application is already selecting the right topic we should be able to fully control our robot with the keyboard. This means that our `core`, `teleop`, and at least `key` applications are working fine.

Now we can focus on our remaining applications. Let's see if our `mapping` is working fine.

Save the map in the correct location

Our `mapping` daemon is a rather complex snap application. Indeed, we used various features of snaps to manage the saving of the map. It's time to verify if everything works as expected. With our simulation running and our `core` and `teleop` daemons running in the background we will simply need to run `turtlebot3c.key` to control our robot, but also we will need to start the mapping service. To do so we can start the service (without enabling it) with:

```
sudo snap start turtlebot3c.mapping
```

We can then follow the logs of our application with the command:

```
sudo snap logs turtlebot3c.mapping -f
```

The logs should be similar to this:

```
turtlebot3c.mapping[104495]: Laser Pose= -1.65701 -0.499409 -3.13914
turtlebot3c.mapping[104495]: m_count 9
turtlebot3c.mapping[104495]: Average Scan Matching Score=303.733
turtlebot3c.mapping[104495]: neff= 100
turtlebot3c.mapping[104495]: Registering Scans:Done
```

This is what the normal log of the mapping should look like. In case it's not the case we must stop the simulation and the snap to make sure there is no `roscore` zombies. We can then use our `key` application to move around and create the map!

Once we moved around, the SLAM should have enough data to generate the map. We can test our "smart map saving" feature by simply stopping the map and seeing if it's created.

We can stop the map with:

```
sudo snap stop turtlebot3c.mapping
```

We should see at the end of our mapping logs something similar to:

```
turtlebot3c.mapping[104749]: Waiting for the map
turtlebot3c.mapping[104749]: Received a 384 X 384 map @ 0.050 m/pix
turtlebot3c.mapping[104749]: Writing map occupancy data to /root/snap/turtlebot3c/common/
map/new_map.pgm
turtlebot3c.mapping[104749]: Writing map occupancy data to /root/snap/turtlebot3c/common/
map/new_map.yaml
turtlebot3c.mapping[104749]: Done
```

What we see in the logs is rather positive since the logs report that the map was saved.

Recall that we decided to save the map in the directory `${SNAP_USER_COMMON}/map`. In this case, our user is `ubuntu`, and the `SNAP_USER_COMMON` should be pointing to `/home/ubuntu/snap/turtlebot3c/common`. But here we can see that the map was saved in the directory: `/root/snap/turtlebot3c/common/`.

Let's check why that is.

Where is my map?

When we call a snap application command, the command is executed with the permission of our current user. In case the user is `ubuntu`, calling the command `MY_SNAP` will run the command with the `ubuntu` permissions. This also means that if we call sudo `MY_SNAP` we will run our command with the `root` permissions. From that point of view, nothing is different with snaps. One thing to know is that snap daemons are running as `root`. This will obviously have some impact. Let's jump inside our snap environment and verify some things. We can start a shell in the snap daemon environment with the following command:

```
sudo snap run --shell turtlebot3c.mapping
```

We will then enter a shell.

Let's go to the root of our snap system.

```
cd $SNAP
```

Simply by running the `ls` command, we might recognize a typical Linux system:

```
etc/ lib/ meta/ opt/ snap/ usr/ var/
```

We can even see our ROS install inside `opt/ros/noetic/`.

If we check the content of the `$HOME` environment variable we will see:

```
$ echo $HOME

/root/snap/turtlebot3c/x
```

So the home of our snap when run as root is this specific directory.

We can run:

```
echo $SNAP_USER_COMMON
```

As we can guess it will give the value:

```
/root/snap/turtlebot3c/common
```

And this is where our maps are saved. This then explains the location of our map. It's simply saved in our `root` user home since we are running a daemon. We can exit this terminal by typing `exit`.

Back to our machine, we can verify that our maps are present with the command:

```
ls -l /root/snap/turtlebot3c/common/map
```

We can see our map files as well as our symbolic link.

Our `mapping` application was then perfectly working. In the future we might want to be careful regarding the `$SNAP_*` variable since they might point to different directories depending on the user or if it's a daemon or not. Let's now see if the `navigation` works!

Fix the navigation application

The `mapping` application successfully created map files. Let's see if our navigation application starts and loads the correct files. We will still need the simulation up and running. Let's keep the logs in a dedicated terminal by typing:

```
sudo snap logs turtlebot3c -f
```

We can then start our navigation service with:

```
sudo snap start turtlebot3c.navigation
```

Then we can see that something is going wrong. We get this log:

```
systemd[1]: Started Service for snap application turtlebot3c.navigation.
turtlebot3c.navigation[106426]: prev_topic: "/nav_vel"
turtlebot3c.navigation[106455]: /snap/turtlebot3c/x2/opt/ros/noetic/lib/map_server/map_
server: error while loading shared libraries: libpulsecommon-13.99.so: cannot open shared
object file: No such file or directory
```

A shared library is missing. It's `libpulsecommon`. A missing library in a snap is a very common problem when building a snap. Let's stop our application since anyway it's crashing:

```
sudo snap stop turtlebot3c.navigation
```

Let's debug our application to give an overview of how to debug such cases.

Debug the missing libpulsecommon library

Our application is crashing because of a missing shared library. This is a very common problem in snaps since the confined environment only contains what was explicitly declared. Let's quickly jump into our application environment to verify what is going on:

```
sudo snap run --shell turtlebot3c.navigation
```

We are now in the exact same condition that our snap is right before calling `navigation_launcher.sh`. Let's see if our `map_server` application is missing a library. We can do so by typing:

```
ldd /snap/turtlebot3c/current/opt/ros/noetic/lib/map_server/map_server
```

Here we are using `ldd` to print the shared object dependencies.

We can then see that in the middle of all the different links and found libraries we have:

```
libpulsecommon-13.99.so => not found
```

So our `map_server` is depending on a library `libpulsecommon` that it cannot find. Let's see if we can find this missing library inside our snap. Type the following command:

```
find $SNAP -type f -name "libpulsecommon-13.99.so"
```

It will search for the library file that our `map_server` is not finding inside our snap. Surprisingly, we find it in our snap:

```
/snap/turtlebot3c/x2/usr/lib/x86_64-linux-gnu/pulseaudio/libpulsecommon-13.99.so
```

This means that our library is installed already in our snap, just that our application is not finding it.

Now we must verify why our library is not found while being present in our snap. On Linux, like in a snap, there are several mechanisms used to search for a dynamic library. The most common is the `$LD_LIBRARY_PATH` environment variable. If we run:

```
echo $LD_LIBRARY_PATH
```

we will see that `$SNAP/usr/lib/x86_64-linux-gnu/` is listed but not with the subdirectory `pulseaudio`.

We can now exit this snap shell by typing `exit`.

The simplest solution would be to extend the `$LD_LIBRARY_PATH` environment variable with the additional directory.

To do that we must modify our `snapcraft.yaml`:

```
  navigation:
+   environment:
+     # map server need pulseaudio
+     LD_LIBRARY_PATH: "$LD_LIBRARY_PATH:$SNAP/usr/lib/x86_64-linux-gnu/pulseaudio"
    command-chain: [usr/bin/mux_select_nav_vel.sh]
```

The `environment` keyword lets us define environment variables for our application. Here we extend the `$LD_LIBRARY_PATH` with the additional `pulseaudio` path relative to our snap.

We can now rebuild and reinstall the snap!

At the end of the build we will see this warning:

```
CVE-2020-27348: A potentially empty LD_LIBRARY_PATH has been set for environment in
'navigation'. The current working directory will be added to the library path if empty.
This can cause unexpected libraries to be loaded.
```

In our case, we can ignore this warning since we set the `$LD_LIBRARY_PATH` to a non-empty value.

We can now retest the `navigation` application.

Navigation test

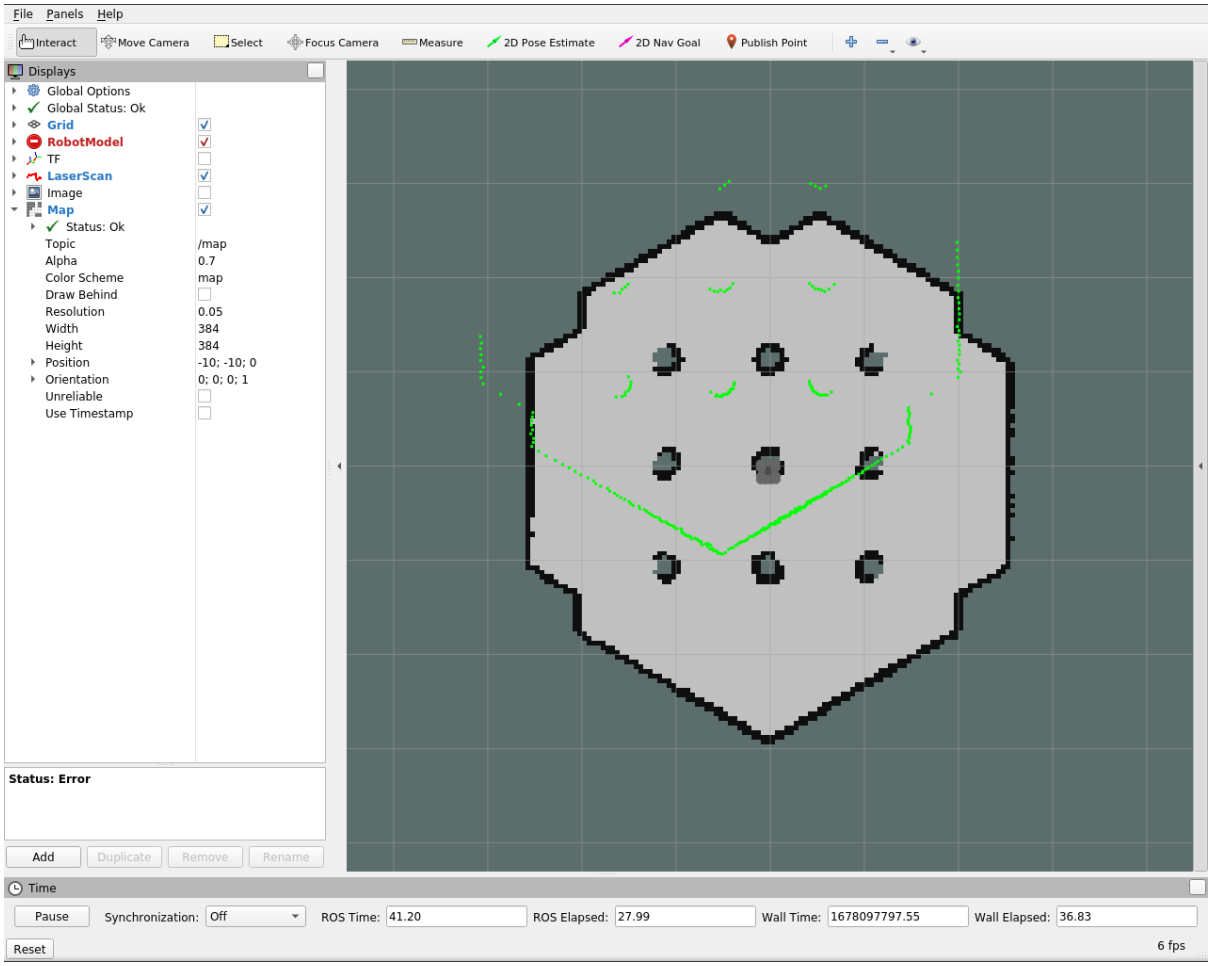
Once our snap is properly rebuilt and reinstalled we can test our `navigation` app. With our last modification, our application should now find every library it needs. We must make sure to restart the simulation after reinstalling the snap. Once up and running, let's run the navigation again:

```
sudo snap start turtlebot3c.navigation
```

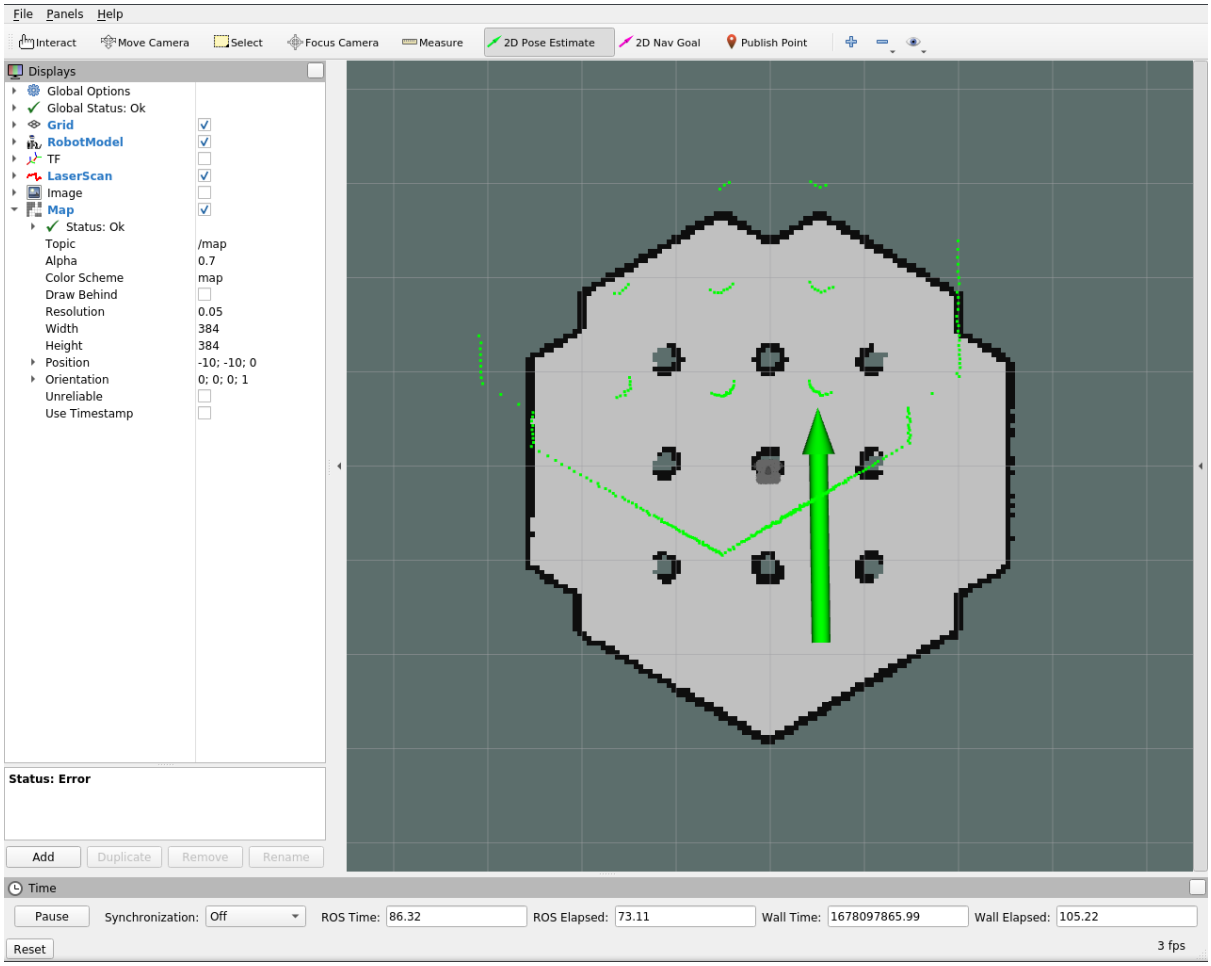
This time we will see that the logs look fine. We can then run `rViz` and see if we can navigate. RViz is a debug/visualization tool, so it doesn't have to be included inside our snap. We can run `rViz` with the TurtleBot3 configuration with the following command:

```
rviz -d /opt/ros/noetic/share/turtlebot3_slam/rviz/turtlebot3_gmapping.rviz
```

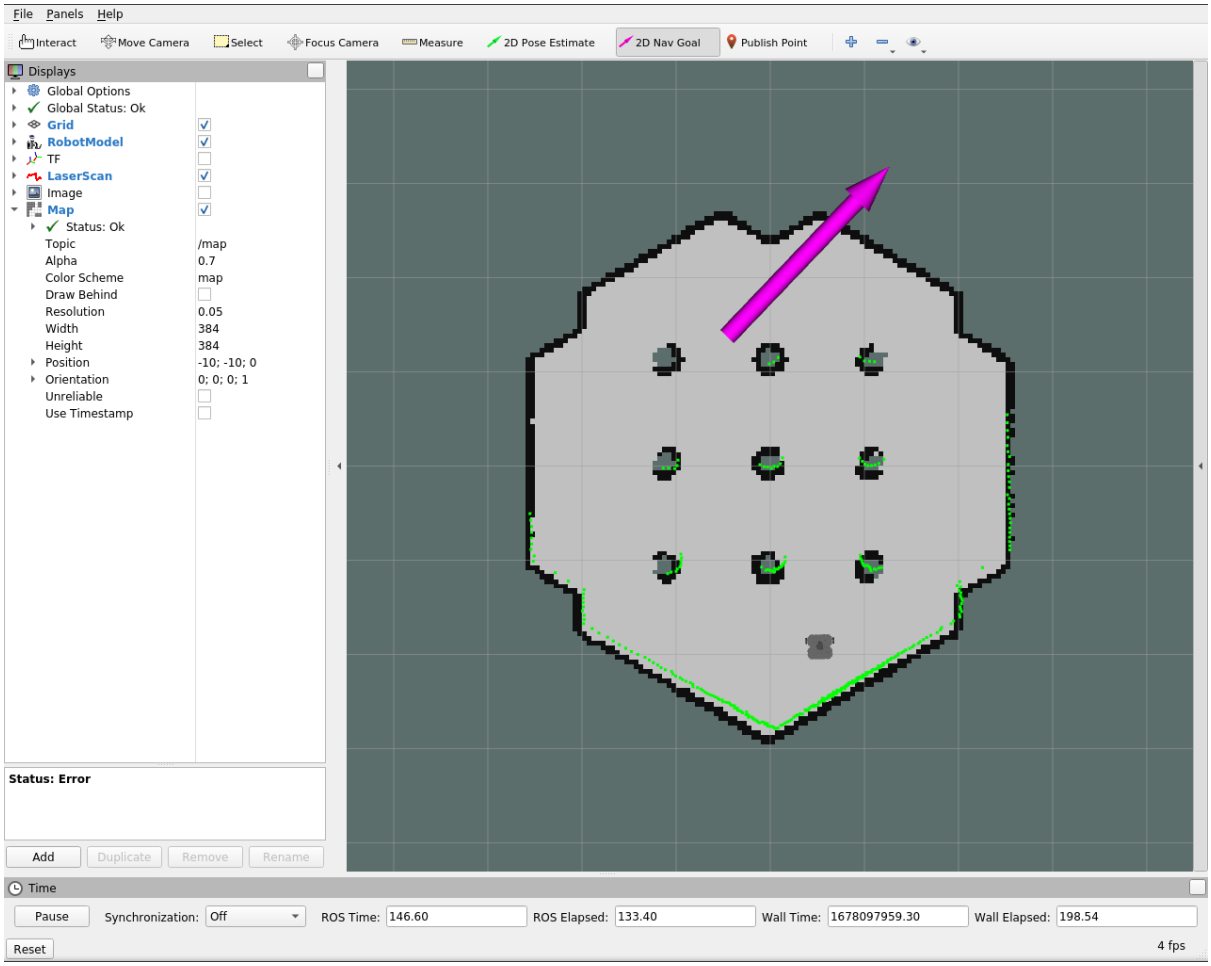
We should see something like this:



Let's select the "2D Pose Estimate" and select the actual position:



We can now check with the “2D Nav Goal” to send a goal to our navigation:



Our robot is now navigating. This means that the `navigation` daemon is running properly but also that our `core` and `teleop` daemons are also working fine.

Now our whole software stack is running!

Final detail

We have now tested our whole stack and every feature was properly working. There is still a final detail to look at before we run a final test.

During the test we could have noticed this warning in the `mapping` and `navigation` logs:

```
[rospack] Warning: cannot create rospack cache directory /root/.ros:
boost::filesystem::create_directory: Permission denied: "/root/.ros"
```

Rospack[227] doesn't use the `$HOME` environment variable but uses the password structure to get the home directory[228]. By default, snaps make sure that the environment variable `$HOME` is pointing to a writable directory, but unfortunately, `rospack` is not using it. The good news is that rospack first tries to read the `$ROS_HOME`[229]. We could make the error disappear by defining the `$ROS_HOME` to `$SNAP_USER_DATA/ros`. Since ROS logging is also using `$ROS_HOME` too, to

[227] http://wiki.ros.org/rospack
[228] https://github.com/ros/rospack/blob/ad85a874575bbed74124b722b42b545537cc6aa3/src/rospack.cpp#L1951
[229] https://github.com/ros/rospack/blob/ad85a874575bbed74124b722b42b545537cc6aa3/src/rospack.cpp#L1935

have the logs of every app and daemon in the same directory we will apply the environment variable change to every app and daemon. We can do that by modifying the `snapcraft.yaml`:

```
  core:
+   environment:
+     ROS_HOME: $SNAP_USER_DATA/ros
[...]
  teleop:
+   environment:
+     ROS_HOME: $SNAP_USER_DATA/ros
[...]
  joy:
+   environment:
+     ROS_HOME: $SNAP_USER_DATA/ros
[...]
  key:
+   environment:
+     ROS_HOME: $SNAP_USER_DATA/ros
[...]
  mapping:
+   environment:
+     ROS_HOME: $SNAP_USER_DATA/ros
[...]
  navigation:
    environment:
+   ROS_HOME: $SNAP_USER_DATA/ros
```

We can then rebuild our snap!

By solving this last little detail, we made sure that all the logs are written properly and in the same location. Snaps rely on the standard usage of Linux. Any solution not following the standard might need adjustment.

Get started with ROS 2 snaps

When creating a snap for a ROS 2 application (or any snap for that matter), the very first step is to create a `snapcraft.yaml` file for the project and file it up with some boilerplate before we can actually get to the specifics of the project at hand. In the case of ROS 2-based applications, we can actually make use of a template to get us started faster.

Note:

If you are new to the whole topic of creating snaps for ROS 2 applications, I'd encourage you to start with the tutorial series: *From zero to hero: deploy a robot with snaps and Ubuntu Core* (page 2). as this particular how-to touches on one single specific aspect: creating the `snapcraft.yaml` file from a template.

To get started with ROS 2 snaps and create a `snapcraft.yaml` from the template, use the command:

```
snapcraft init --name my-ros2-project --profile ros2
```

This command will generate the `snapcraft.yaml` file inside a `snap` folder in the current directory:

```
$ tree
.
└── snap
    └── snapcraft.yaml
```

This is nice and all, but simply creating a file is not all that interesting. The real value lies in that said file is a functioning snap recipe for an actual ROS 2 demo.

Let us see what it contains:

```
# The name of the snap.
name: my-ros2-project
# Just for humans, typically '1.2+git' or '1.3.2'
version: "0.0.1"
# 79 char long summary
summary: Single-line elevator pitch for your amazing snap
description: |
  This is my-ros2-project's description. You have a paragraph or two to tell the
  most important story about your snap. Keep it under 100 words though,
  so that it looks good in the snap store.

# The base snap is the runtime environment for this snap.
# Each ROS 2 LTS distribution has a corresponding base in the core** series.
# View the compatible bases at:
# https://documentation.ubuntu.com/snapcraft/stable/reference/extensions/ros-2-extensions
base: core24

# use 'strict' once you have the right plugs and slots
confinement: devmode
# must be 'stable' to release into candidate/stable channels
grade: devel

# The applications exposed by the snap.
apps:
  ros2-talker-listener:
    command: ros2 launch demo_nodes_cpp talker_listener.launch.py
    # The ROS extensions establish common settings for all ROS snaps.
    # Learn more about it at https://canonical-robotics.readthedocs-hosted.com/en/latest/
references/snapcraft/extensions/
    extensions: [ros2-jazzy-ros-core]

# The parts to build the snap.
parts:
  ros-demos:
    # The colcon plugin builds parts for ROS 2.
    # Learn more about the plugin at https://documentation.ubuntu.com/snapcraft/stable/
reference/plugins/colcon_plugin
    plugin: colcon
    source: https://github.com/ros2/demos.git
    source-branch: jazzy
    source-subdir: demo_nodes_cpp
```

The recipe comes with most expected directives pre-filed with default values that are informative and commented. Note that the template packages a talker-listener demo from the upstream `ros2/demos` GitHub repository. It retrieves the source code from a specific branch and packages only a sub-directory of this large collection of demos. It then invokes a plain `ros2 launch` command to start the talker-listener demo. The last point to which I would like

to draw your attention to is that this template uses `core24` and ROS 2 Jazzy. Should you be using a different ROS 2 distribution, you will find links to the documentation right at your fingertip to help you in your endeavor.

Migrate from docker to snap

Docker has greatly facilitated robotics software development by providing a way to package applications and their dependencies into portable containers. However, due to its cloud-oriented design, Docker poses some difficulties for developers when it comes to deploying software on a robotic device.

In this document, we are going to see when and how to migrate a ROS application currently deployed with Docker to a Snap.

When to migrate

The software life cycle for a robotics application typically consists of four stages: development, testing, deployment, and maintenance.

When transitioning from development and testing to deployment and maintenance, Docker's limitations in embedded devices become apparent. Docker lacks dedicated high-level interfaces for accessing low-level hardware, a robust update system, state transactionality and are not integrated in terms of network. All of these require the user to implement workarounds that can be challenging and expose your application to security issues. This is where developers start their migration to snaps.

Snaps offer a better solution for the deployment and maintenance of the software life cycle.

They provide robust security features to ensure the safety and integrity of software applications and are cryptographically-signed. Additionally, automatic over-the-air updates ensure that snaps are always up-to-date, while delta-binary downloads minimize bandwidth costs. Snaps also provide atomic install and removal functionality which ensures that the overall system is never in a broken state.

From Docker to Snap

Docker images are built using Dockerfiles, which are essentially scripts containing a series of commands executed in sequence to define the image of your Docker container. Snaps are built based on a recipe declared in a YAML file[230]. This similarity will prove convenient while converting from Docker to Snap as you will see hereafter.

A generic *snapcraft.yaml* file is defined by four main blocks:

- snap's metadata
- build environment
- parts definition
- apps definition

[230] https://snapcraft.io/docs/build-configuration

More detailed information about creating a snapcraft yaml file can be found in Creating snapcraft.yaml[231].

Snap metadata

This is the data that describes your snap, including your snap's name, version, icon location and summary.

For help completing these details, see global metadata[232].

Build environment

Docker allows users to create images based on existing parent images that provide the required environment for the application. This is useful to avoid having to set up the same core libraries every time. Similarly, snaps provide base snaps[233].

Both base snaps and parent Docker images serve as the foundation for building our application. For example, for a ROS Noetic application, in Docker you would include the Ubuntu 20.04 LTS image with the following command:

```
FROM ubuntu:20.04
```

In a similar way, in snapcraft you will select the core20 base for the Snap as follows:

```
base: core20
```

In Docker there are all sorts of images, even ones with ROS already installed. However, base snaps have the goal of guaranteeing a minimal, stable, maintained and secure environment. ROS is thus installed on top of this in the parts as we will see hereafter.

Visit the base snaps documentation[234] for more information.

Building the application

Once the build environment is defined you can proceed to building your application.

When writing a Dockerfile, you think of the bash commands that would run on the host to install an application and all of its dependencies correctly. For a generic ROS application these commands would install ROS, install the project dependencies using rosdep, compile your ROS package and source it.

[231] https://snapcraft.io/docs/creating-snapcraft-yaml
[232] https://snapcraft.io/docs/adding-global-metadata
[233] https://snapcraft.io/docs/base-snaps
[234] https://snapcraft.io/docs/base-snaps

Building ROS

In Dockerfile, ROS can be made available in two ways. Either by writing the bash commands required to install ROS as outlined in the ROS documentation[235] or by using a Docker parent image with ROS preinstalled. In both cases, ROS installation requires the following steps:

- adding the ros package repositories
- setting up the GPG keys
- installing the ROS debian package.

In Snaps, adding the required package repository and setting up the keys is done using the Snapcraft package repository keyword[236]. For example:

```
package-repositories:
- components: [main] # Apt repository components to enable
  formats: [deb] # List of deb types to enable
  key-id: C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654 # 40 character GPG key identifier
  key-server: keyserver.ubuntu.com # Key server to fetch key
  suites: [focal] # Repository suites to enable
  type: apt # Specifies type of package-repository
  url: http://repo.ros2.org/ubuntu/main # Repository URL
```

The ROS debian packages are installed by defining a Snap part[237]. Snap parts are recipes to build a piece of software and are driven via plugins[238]. When defining a part, you can search the snapcraft plugins[239] page to select the required plugin. If we solely want to install some ROS packages, no source code involved, we can use the nil plugin as follows:

```
parts:
  ros2-humble-extension:
    plugin: nil
```

To add the packages that are needed to build our ROS project you can use the build-packages keyword, which allows for the installation of build-time dependencies. You can learn more about Build and staging dependencies[240] in the documentation.

For a ROS project, the bare minimum packages required are those setting up a ROS workspace, hence they are added to the list of build-packages as follows:

```
parts:
  ros2-humble-extension:
  plugin: nil # Plugin for parts with no source to import
  build-packages:
    - ros-humble-ros-environment
    - ros-humble-ros-workspace
    - ros-humble-ament-index-cpp
    - ros-humble-ament-index-python
```

It's important to emphasize that `build-packages` are only used for building and won't be packaged in the final Snap.

[235] http://wiki.ros.org/noetic/Installation/Ubuntu
[236] https://snapcraft.io/docs/package-repositories
[237] https://snapcraft.io/docs/snapcraft-yaml-schema
[238] https://snapcraft.io/docs/snapcraft-plugins
[239] https://snapcraft.io/docs/supported-plugins
[240] https://snapcraft.io/docs/build-and-staging-dependencies

To ease sourcing the ROS workspace for an application, Snapcraft provides a script[241] to do so. You can pull the script from the Snapcraft source and install it. Following the example above the `ros2-humble-extension` will look like this:

```
parts:
  ros2-humble-extension:
  plugin: nil # Plugin for parts with no source to import
  build-packages:
    - ros-humble-ros-environment
    - ros-humble-ros-workspace
    - ros-humble-ament-index-cpp
    - ros-humble-ament-index-python
  source: $SNAPCRAFT_EXTENSIONS_DIR/ros2
  override-build: install -D -m 0755 launch ${SNAPCRAFT_PART_INSTALL}/snap/command-chain/
ros2-launch # Install the ros2-launch script responsible of sourcing your ROS environment
```

Read more about overriding the build step[242] in the documentation.

This is the process to set up ROS in a snap, and the process is the same for every ROS distribution.

Building your ROS package

After setting up ROS, you can proceed building the ROS package. In a Dockerfile the application's source code is copied into the image file system, its dependencies installed and the package compiled. Something along the line of:

```
COPY ./my-ros-application ./src/my-ros-application

RUN source /opt/ros/$ROS_DISTRO/setup.bash && \
    rosdep update --rosdistro $ROS_DISTRO && \
    rosdep install -i --from-path src --rosdistro $ROS_DISTRO -y && \
    cd /ros_ws && \
    catkin_make
```

In Snap, this is all handled automatically with the catkin plugin. Add it to your part as follows:

```
parts:
  my-ros-application-part:
    plugin: catkin
```

Then you need to provide the plugin with information on the source it has to build.

The source can be a folder on your host or the link to a git repository. You can learn more about parts, how to set up a specific branch or a subfolder in the Snapcraft parts metadata[243] documentation. Let's add an example source in the previous example:

```
parts:
  my-ros-application-part:
    plugin: catkin
    source: https://github.com/my-company/my-ros-application.git
    source-branch: testing-branch
```

[241] https://github.com/snapcore/snapcraft/blob/main/extensions/ros2/launch
[242] https://documentation.ubuntu.com/snapcraft/stable/how-to/crafting/override-the-default-build/
[243] https://snapcraft.io/docs/snapcraft-yaml-schema

The plugin will take care of installing the dependencies defined in the *package.xml* file.

Remember to add any dependency that is required by the application and is not included in rosdep or installing instructions via the `build-packages` and `stage-packages` keywords.

By including these keywords in your part, you can ensure that all necessary packages and dependencies are properly installed.

Running the application

When deploying a ROS application in either Docker or Snapcraft, you can identify three main components that must be defined:

- command; launch file or node to be run
- enabling access to the necessary host resources (such as cameras, GPIO pins, network connections, and drivers), defining the launch file or rosnode to run
- sourcing ROS and the workspace

Command

Docker offers various means of defining a command at container runtime (e.g. you can define a docker_compose.yaml file or use the CMD keyword in the Dockerfile). Also the entrypoint allows to specify bash commands that should be run when the container is started.

Snaps effectively allows you to define and isolate the pieces of your application that you want to expose to the rest of the system via the apps[244] tag.

After having identified the command that launch your application you can add it with the command keyword as follows:

```
apps:
  my-awesome-ros-app:
    command: opt/ros/noetic/bin/roslaunch my-awesome-ros-app app.launch
```

Access to host resources

In order to allow access to host resources via Docker it is necessary to define ports and volumes in the Dockerfile or when running the container. For example, Docker uses a virtual network to manage communication between containers. In order for the network to use the host's network namespace, you have to run your container with `--network=host` option. Other examples can include using the `--device` flag to access a usb port, or various flags required for X11 forwarding to render GUI applications.

By default snap applications are confined and are not allowed to access any of the host resources. Interfaces and plugs[245] allow the user to define the resources on the host that the application will have access to. You can have a look at the list of supported interfaces[246].

[244] https://snapcraft.io/docs/snapcraft-yaml-schema
[245] https://snapcraft.io/docs/interface-management
[246] https://snapcraft.io/docs/supported-interfaces

For a generic ROS application that communicates with other ROS components via topics, you will need the the `network` plug to grant the Snap access to the host's network, and also the `network-bind` plug, which provides the Snap with the ability to bind to a specific IP address and port as required for ROS communication. For instance, if you needed X11 forwarding for a GUI you would use the x11 interface[247] and so on. By adding the plugs, my app would look like this:

```
apps:
  my-awesome-ros-app:
    command: opt/ros/noetic/bin/roslaunch my-awesome-ros-app app.launch
      plugs: [network, network-bind, x11]
```

Sourcing

In Docker, sourcing of the ROS workspace is usually handled in the Dockerfile in the following way:

```
RUN echo "source /opt/ros/$ROS_DISTRO/setup.bash" >> ~/.bashrc && \
    echo "source /ros_ws/devel/setup.bash" >> ~/.bashrc
```

Sourcing can also be done using the Docker entrypoint script.

In Snapcraft this is achieved using the script installed earlier and executing it before launching our application through the `command-chain` tag as shown below:

```
apps:
  my-awesome-ros-app:
    command: opt/ros/noetic/bin/roslaunch my-awesome-ros-app app.launch
    plugs: [network, network-bind]
    command-chain: [snap/command-chain/ros1-launch]
```

The command-chain keyword contains a list of commands to be executed prior to the app command. Because of it, we are relieved from the concern of having to source the ROS environment before launching our ROS application.

ROS extensions

To further simplify the deployment of snaps for robotic applications, ros-extensions have been implemented. ROS extensions automatically set up a fair share of what we've just detailed here, such as adding the ROS apt package repository, building ROS and defining the build environment. This means that developers can focus on building their application without worrying about the underlying ROS integration to Snapcraft.

The ros-extensions currently available are:

- ros1-noetic-extension[248]
- ros2-foxy-extension[249]
- ros2-humble-extension[250]

[247] https://snapcraft.io/docs/x11-interface
[248] https://snapcraft.io/docs/ros-noetic
[249] https://snapcraft.io/docs/ros2-foxy-extension
[250] https://snapcraft.io/docs/ros2-humble-extension

You can read more about snap extensions[251] in the official documentation.

Learn more

In this post you have seen how to Snap an application by gathering information from its Dockerfile. To learn more about the core snap concepts look at the next resources:

- base snaps[252]
- parts[253]
- apps[254]
- extensions[255]
- plugs and interfaces[256]
- How to deploy robotic applications with snaps[257]

Use ROS 2 shared memory in snaps

Strictly confined ROS 2 snaps shows an access error regarding shared memory. If you see something similar to:

```
[RTPS_TRANSPORT_SHM Error] Failed to create segment 86bb3c83d0835208: Permission denied ->
Function compute_per_allocation_extra_size
[RTPS_MSG_OUT Error] Permission denied -> Function init
```

ROS 2 communication library is trying to use the shared memory mechanism. But don't worry, even if you see this error, the messages are going to be transmitted (just not through shared memory).

Here, we present solutions to make the error message disappear by activating snap shared memory interface or simply disabling ROS 2 shared memory.

If you want a detail explanation about ROS 2 shared memory and why it doesn't work right away in snaps please visit the blog: how-to-use-ros-2-shared-memory-in-snaps[258].

Note that everything discussed hereafter is exemplified on GitHub[259].

[251] https://snapcraft.io/docs/snapcraft-extensions
[252] https://snapcraft.io/docs/base-snaps
[253] https://snapcraft.io/docs/adding-parts
[254] https://snapcraft.io/docs/snapcraft-yaml-schema
[255] https://snapcraft.io/docs/snapcraft-extensions
[256] https://snapcraft.io/docs/interface-management
[257] https://snapcraft.io/docs/robotics
[258] https://canonical.com/blog/how-to-use-ros-2-shared-memory-in-snaps
[259] https://github.com/ubuntu-robotics/ros-snaps-examples/tree/main/shared_memory_foxy_core20

Public shared memory interface for ROS 2

Snap is providing an interface called shared-memory[260]. This interface allows different snaps to have access (read and/or write) to a specified path. This is meant to share resources in `/dev/shm` across snaps. To do so, we have to declare both a `slot` as well as `plug`[261]. Before running the snap, we will have to manually connect these slot & plug, possibly connecting several other snaps to the same slot.

The slot is the one defining the shared memory paths to be accessed. With ROS 2, we actually need access to everything within the `/dev/shm` directory (due to the semaphore temporary files). An example of the slot and plug is as follows:

```
slots:
  shmem-slot:
    interface: shared-memory
    write: ['*'] # paths are relative to /dev/shm
    private: false
plugs:
  shmem-plug:
    interface: shared-memory
    shared-memory: shmem-slot
    private: false
```

Then both, the plug and the slot are added to the apps:

```
apps:
  my-ros-2-app:
    [...]
    plugs: [network, network-bind, shmem-plug]
    slots: [shmem-slot]
```

Once the snap is built and installed, we can connect the shared memory with the command

```
sudo snap connect my_snap_name:shmem-plug my_snap_name:shmem-slot
```

Additional snaps will simply have to create their own plug and connect it to the very same `slot`.

Caution:

As specified in the FastDDS documentation[262], using the ROS 2 shared memory and different users might lead to communication problems. Note that snap daemons are running as `root` while snap CLI applications might be running with a different user that would cause a communication problem.

[262] https://github.com/eProsima/Fast-DDS-docs/blob/master/docs/fastdds/transport/shared_memory/shared_memory.rst?plain=1#L71:L78

This solution has an important overhead, since one must define slots and plugs, possibly across multiple snaps. But it is the de-facto way to enable the use of the shared memory feature in strictly confined snaps. Note that `snapd 2.56.2` (or above) is necessary.

[260] https://snapcraft.io/docs/shared-memory-interface
[261] https://snapcraft.io/docs/interface-management

You can find a complete example of a ROS 2 snap using the public shared memory interface on GitHub[263].

Private shared memory interface

Snap is providing another interface called private shared memory[264] that vastly simplifies shared memory support with snap. The private shared memory[265] is a subset of the shared memory interface[266]. Without modifying your software, all calls to `/dev/shm` are going to be bound to `/dev/shm/snap.SNAP_NAME` automatically. It can be activated by simply adding the plug to the `snapcraft.yaml`:

```
plugs:
  shared-memory:
    private: true
```

In a ROS 2 context, adding the private shared memory[267] allows the shared memory to work within a given snap, hence benefiting from better performance.

Everything is fine within the snap, but the application is mute to the outside. ROS 2 applications running on the host, but outside the snap, will not be able to see the topics from the snap; let alone to subscribe to them.

This being said, it is important to note that if you access these topics published from the snap from another computer, it will work seamlessly as the data are shared via UDP.

You can find a complete example of a ROS 2 snap using the private shared memory interface on GitHub[268].

Disabling shared memory for ROS 2

FastDDS offers two options to totally disable the shared memory feature; either at compile time or at run time. We are detailing both options hereafter.

At compile-time

FastDDS offers an option to compile without the shared memory feature by simply specifying a CMake variable: `-DSHM_TRANSPORT_DEFAULT=OFF`. With this, no shared memory nor any associated files – ciao the error message. Adding the following FastDDS part to your *snapcraft.yaml* will disable FastDDS at compile time:

```
  fastdds:
    plugin: colcon
    source: https://github.com/eProsima/Fast-DDS.git
```

(continues on next page)

[263] https://github.com/ubuntu-robotics/ros-snaps-examples/tree/main/shared_memory_foxy_core20/public-shared-memory
[264] https://snapcraft.io/blog/private-shared-memory-support-for-snaps
[265] https://snapcraft.io/blog/private-shared-memory-support-for-snaps
[266] https://snapcraft.io/docs/shared-memory-interface
[267] https://snapcraft.io/blog/private-shared-memory-support-for-snaps
[268] https://github.com/ubuntu-robotics/ros-snaps-examples/tree/main/shared_memory_foxy_core20/private-shared-memory

(continued from previous page)

```
    source-tag: "v2.1.1" # Use tag according to your ROS version
    colcon-cmake-args:
            - -DCMAKE_BUILD_TYPE=Release
            - -DSHM_TRANSPORT_DEFAULT=OFF
    build-packages:
            - libasio-dev
            - ros-foxy-fastcdr # Replace by used ROS 2 release
            - nlohmann-json3-dev
            - ros-foxy-tinyxml2-vendor  # Replace by used ROS 2 release
    stage-packages:
            - ros-foxy-fastcdr  # Replace by used ROS 2 release
            - ros-foxy-tinyxml2-vendor  # Replace by used ROS 2 release
```

Of course, the main drawback of this approach is that we have to recompile FastDDS with every snap.

You can find a complete example of a ROS 2 snap using the FastDDS with shared memory disable at compile time on GitHub[269].

Disabling shared memory at run-time

FastDDS also allows for providing a configuration XML file[270] at runtime in order to customize several aspects of the middleware. Such as, forcing the transport to use UDPv4. The XML profile is passed through an environment variable[271]: Under your `snap/local` directory, create the file `fastdds_no_shared_memory.xml` with the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>CustomUdpTransport</transport_id>
                <type>UDPv4</type>
            </transport_descriptor>
        </transport_descriptors>

        <participant profile_name="participant_profile" is_default_profile="true">
            <rtps>
                <userTransports>
                    <transport_id>CustomUdpTransport</transport_id>
                </userTransports>

                <useBuiltinTransports>false</useBuiltinTransports>
            </rtps>
        </participant>
    </profiles>
```

And then, you can add the proper `part` to place your profile and set the environment variable to your app in your *snapcraft.yaml*:

[269] https://github.com/ubuntu-robotics/ros-snaps-examples/tree/main/shared_memory_foxy_core20/disable-shared-memory-compile-time
[270] https://fast-dds.docs.eprosima.com/en/latest/fastdds/xml_configuration/xml_configuration.html
[271] https://fast-dds.docs.eprosima.com/en/latest/fastdds/env_vars/env_vars.html

```
parts:
  [...]
  config:
    plugin: dump
    source: snap/local/
    organize:
      'fastdds_no_shared_memory.xml': usr/share/
apps:
  my-ros-2-app:
    [...]
    environment:
        FASTRTPS_DEFAULT_PROFILES_FILE: ${SNAP}/usr/share/fastdds_no_shared_memory.xml
```

This is much easier to set up and to change in subsequent releases of a snap.

You can find a complete example of a ROS 2 snap using the FastDDS with shared memory disable at run time on GitHub[272].

Snap ROS distributions with no extensions

The snapcraft ROS extensions help you snap ROS applications for the different ROS distributions. However, this does not mean that you cannot build a snap for a ROS distribution that does not have a dedicated extension. In this document, you will see how to do that.

How to snap a ROS application without extension?

The ROS extensions facilitate the deployment of ROS applications with snaps. What does a ROS extension do?

ROS extensions, as all other snapcraft extensions, factorize some of the YAML entries found in the *snapcraft.yaml* file that are common and necessary for building ROS applications. For ROS extensions these entries are,

- The ROS APT package repository and its GPG key.
- The following build-packages:
  - ros-ROS-DISTRO-ros-environment
  - ros-ROS-DISTRO-ros-workspace
  - ros-ROS-DISTRO-ament-index-cpp
  - ros-ROS-DISTRO-ament-index-python
- The build-environment variables:
  - ROS_VERSION = ros-version
  - ROS_DISTRO = ros-distro
- The command-chain command script which takes care of sourcing the ROS environment prior to launching your application.

[272] https://github.com/ubuntu-robotics/ros-snaps-examples/tree/main/shared_memory_foxy_core20/disable-shared-memory-run-time

All of these additional YAML entries can be revealed from your *snapcraft.yaml* file with the command `snapcraft expand-extensions`.

With this in mind, let us see how you can replicate what the extension is doing for other ROS distributions.

Important:

Core18 does not support Snapcraft Extensions. If you are developing a ROS snap based on ROS Melodic distro, then all the extensions entries are handled by the catkin plugin[273]. To check an example see the core18 example here[274].

[273] https://snapcraft.io/docs/catkin-plugin#heading--core18
[274] https://snapcraft.io/docs/ros-applications

Writing the snap

As an example, the process of creating a simple talker-listener snap based on ROS 2 Galactic will be shown hereafter.

Building ROS

ROS installation requires the following steps:

- adding the ROS package repositories
- setting up the GPG keys
- installing the ROS Debian package
- source ROS workspace

Let's see how to achieve each step with Snaps.

Adding the required package repository and setting up the keys is done by using the Snapcraft package repository keyword[275] as follows:

```
# Add ROS 2 repository
package-repositories:
  - components: [main]
    formats: [deb]
    key-id: C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
    key-server: keyserver.ubuntu.com
    suites: [focal]
    type: apt
    url: http://repo.ros2.org/ubuntu/main
```

Now, let's proceed to install the ROS Debian packages. The ROS Debian packages are installed by defining a snap part[276]. Snap parts are recipes to build a piece of software and are

[275] https://snapcraft.io/docs/package-repositories
[276] https://snapcraft.io/docs/snapcraft-yaml-schema

driven via plugins[277]. For ROS, you solely want to install the ROS Debian package, no source code involved, you can use the nil plugin[278] as follows:

```
parts:
  ros2-galactic-extension:
    plugin: nil
```

To add the packages that are needed to build our ROS project you can use the build-packages keyword, which allows for the installation of build-time dependencies. You can learn more about build and staging dependencies[279] in the documentation. For a ROS project, the bare minimum packages required are those setting up a ROS workspace, hence they are added to the list of build-packages as follows:

```
parts:
  ros2-galactic-extension:
    plugin: nil
    build-packages:
      - ros-galactic-ros-environment
      - ros-galactic-ros-workspace
      - ros-galactic-ament-index-cpp
      - ros-galactic-ament-index-python
```

It's important to emphasize that build-packages are only used for building and won't be packaged in the final snap.

Finally, the ROS workspace has to be sourced for ROS applications to run. To ease this process, snapcraft provides a script[280] to do so. You can pull the script from the snapcraft source and install it. Following the example above, the ros2-galactic-extension will look like this:

```
parts:
  ros2-galactic-extension:
    plugin: nil
      build-packages:
        - ros-galactic-ros-environment
        - ros-galactic-ros-workspace
        - ros-galactic-ament-index-cpp
        - ros-galactic-ament-index-python
      override-build: |
        install -D -m 0755 $SNAP/share/snapcraft/extensions/ros2/launch ${SNAPCRAFT_PART_
INSTALL}/snap/command-chain/ros2-launch
```

Read more about overriding the build step[281] in the documentation. This is the process to set up ROS in a snap, and the process is the same for every ROS distribution.

[277] https://snapcraft.io/docs/snapcraft-plugins
[278] https://snapcraft.io/docs/nil-plugin
[279] https://snapcraft.io/docs/build-and-staging-dependencies
[280] https://github.com/snapcore/snapcraft/blob/main/extensions/ros2/launch
[281] https://documentation.ubuntu.com/snapcraft/stable/how-to/crafting/override-the-default-build/

Building the demo application

Now that ROS has been dealt with, let's proceed with building the ROS application source code. This is done by creating a new snap part. ROS 2 provides some demos in the demos GitHub repository[282]. ROS packages are built by cloning the source code, installing its dependencies via rosdep and compiled with colcon. In snapcraft, all of this is handled via the colcon plugin[283] which you can add to the part as follows:

```
ros-demos:
  after: [ros2-galactic-extension]
  plugin: colcon
  source: https://github.com/ros2/demos.git
  source-branch: galactic
  source-subdir: demo_nodes_cpp
```

Our application requires the `roslaunch` package as a run dependency however this is not included as a `run_dependency` in the *package.xml* file of our example. Therefore, you need to also include it in the part by using the stage-packages keyword as follows:

```
ros-demos:
  after: [ros2-galactic-extension]
  plugin: colcon
  source: https://github.com/ros2/demos.git
  source-branch: galactic
  source-subdir: demo_nodes_cpp
  stage-packages: [ros-galactic-ros2launch]
```

Finally, for the part to install all the correct dependencies versions and build, it is necessary to define the ROS version and distro. This is done by defining the `build-environment` variables as follows:

```
# Define the ROS 2 environment variable necessary for install and build time
build-environment:
  - ROS_VERSION: '2'
  - ROS_DISTRO: galactic
```

This is it, you can now proceed in defining the application that will be launched by your snap.

Running the application

When deploying a ROS application you can identify three main components that must be defined:

- command; launch file or node to be run
- enabling access to the necessary host resources (such as cameras, GPIO pins, network connections, and drivers), defining the launch file or rosnode to run
- sourcing ROS and the workspace

Snaps effectively allows you to define and isolate the pieces of your application that you want to expose to the rest of the system via the apps[284] tag.

[282] https://github.com/ros2/demos.git
[283] https://snapcraft.io/docs/colcon-plugin
[284] https://snapcraft.io/docs/snapcraft-yaml-schema

After having identified the command that launch your application you can add it with the command keyword as follows:

```
apps:
  ros2-talker-listener:
  command: opt/ros/galactic/bin/ros2 launch demo_nodes_cpp talker_listener.launch.py
```

By default, snap applications are confined and are not allowed to access any of the host resources. Interfaces and plugs[285] allow the user to define the resources on the host that the application will have access to. You can have a look at the list of supported interfaces[286].

For a generic ROS application that communicates with other ROS components via topics, you will need the `network` plug to grant the snap access to the host's network, and also the `network-bind` plug, which provides the snap with the ability to bind to a specific IP address and port as required for ROS communication. You can add those to the application as follows:

```
apps:
  ros2-talker-listener:
    command: opt/ros/galactic/bin/ros2 launch demo_nodes_cpp talker_listener.launch.py
    plugs: [network, network-bind]
```

In order to source the ROS environment, you can use the command-chain keyword, which allows us to list commands to be executed before our main command. In this case, you will execute the script that was pulled in the Snap in the ROS part as follows:

```
apps:
  ros2-talker-listener:
    command-chain: [snap/command-chain/ros2-launch]
    command: opt/ros/galactic/bin/ros2 launch demo_nodes_cpp talker_listener.launch.py
    plugs: [network, network-bind]
```

Finally, to run the application it is necessary to source the ROS environment and define the necessary ROS variables such as the `PYTHONPATH`, `ROS_DISTRO`, `ROS_VERSION`:

```
environment:
  PYTHONPATH: $SNAP/opt/ros/galactic/lib/python3.8/site-packages:$SNAP/usr/lib/python3/
dist-packages:${PYTHONPATH}
  ROS_DISTRO: galactic
  ROS_VERSION: '2'
```

This is it, now you can run your ROS application with snap. You can look at the full *snapcraft.yaml* file described in this document: non_lts_galactic/snap/snapcraft.yaml[287].

[285] https://snapcraft.io/docs/interface-management
[286] https://snapcraft.io/docs/supported-interfaces
[287] https://github.com/ubuntu-robotics/ros-snaps-examples/blob/main/non_lts_galactic/snap/snapcraft.yaml

See also

- Snapcraft ROS Noetic extension[288]: The Snapcraft extension to snap ROS Noetic applications.
- Snapcraft ROS Foxy extension[289]: The Snapcraft extension to snap ROS Foxy applications.
- Snapcraft ROS Humble extension[290]: The Snapcraft extension to snap ROS Humble applications.
- Snapcraft supported extensions[291]: Complete list of Snapcraft extensions available to developers.

Configure a snap: pull a configuration from a server

When a robotics application is snapped, one might want to use it on multiple different robots.

Reusing the same snap means that we must be able to configure the snap to the specificity of a robot once installed on it.

We present in this guide the steps to host a configuration on a server and use it for our snap.

Our snap will get its configuration not from the snapped package but from a web server. This allows us to use the same snap on multiple devices with different configurations hosted remotely, as well as updating the configuration without having to update the snap itself.

For this how-to-guide, we use the example ubuntu-robotics/snap_configuration[292].

The repository consists of the `snapcraft.yaml` file from which the snap is built, as well as a launcher script.

The repository contains a standard snap package providing the `key_teleop`[293] application from the teleop_tool[294] ROS 2 package. The goal here is to be able to configure the application without having to update the snap. The `key_teleop` node can be configured for its `forward_rate`, `backward_rate` and `rotational_rate` parameters. They are the parameters we will be configuring from the server.

Requirements

This how-to-guide is assuming that we are familiar with robotics snaps. Please refer to our tutorials[295] to learn more about robotics snaps.

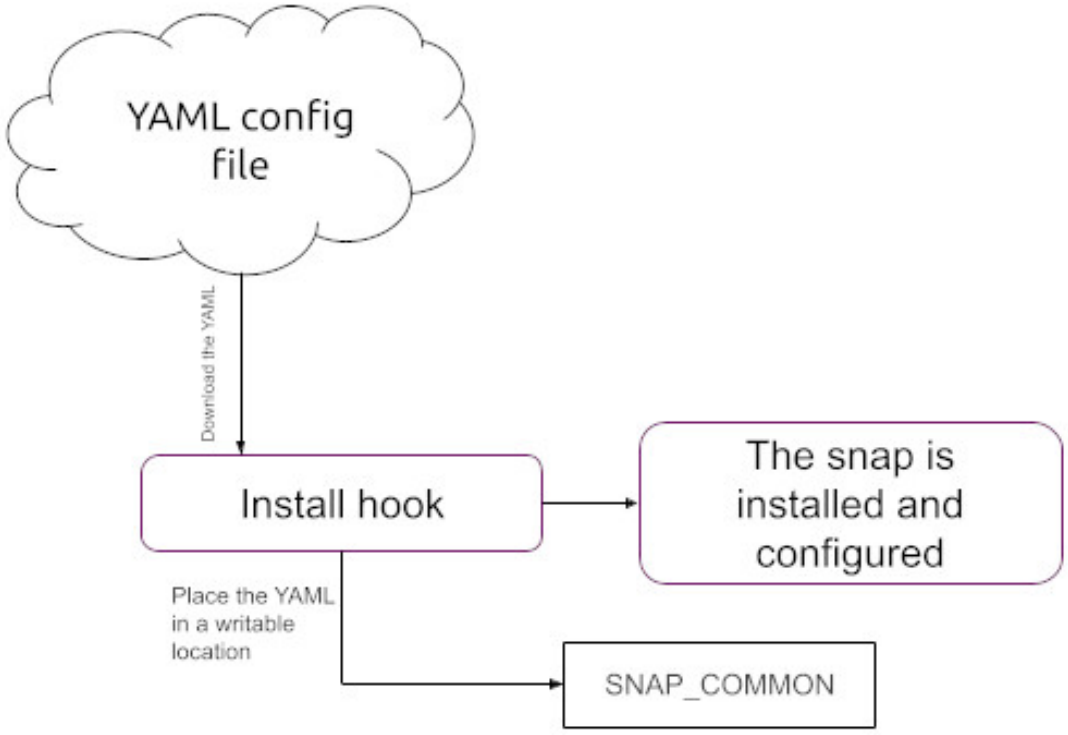
Additionally, this how-to-guide will require a GitHub account[296] and the associated ssh key setup[297] (this is necessary to follow this how-to-guide but not for the approach).

[288] https://snapcraft.io/docs/ros-noetic
[289] https://snapcraft.io/docs/ros2-foxy-extension
[290] https://snapcraft.io/docs/ros2-humble-extension
[291] https://snapcraft.io/docs/supported-extensions
[292] https://github.com/ubuntu-robotics/snap_configuration
[293] https://github.com/ros-teleop/teleop_tools/tree/master/key_teleop
[294] https://github.com/ros-teleop/teleop_tools/tree/master
[295] https://ubuntu.com/robotics/docs
[296] https://github.com/
[297] https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

An up and running Ubuntu (minimum 20.04) with snapcraft[298] installed is also required.

Download the initial configuration

To get the initial configuration, we will use the snap `install` hook[299]. The following diagram shows how the `install` hook will pull the configuration from a server and place it in `$SNAP_COMMON`[300]. Once this is done, our snap will be installed and configured.



Host a configuration file

Since we are retrieving the configuration from a remote location, it has to be hosted somewhere. While any file server would do, we will use GitHub as a simple and easy solution.

In order to reproduce the how-to-guide, we should fork the ubuntu-robotics/snap_configuration[301].

We will place the configuration file in the same git repository as the snap for the sake of simplicity.

First, we clone our fork:

```
git clone git@github.com:YOUR_GH_USERNAME/snap_configuration.git
```

[298] https://snapcraft.io/snapcraft
[299] https://snapcraft.io/docs/supported-snap-hooks#heading--install
[300] https://ubuntu.com/robotics/docs/snap-data-and-file-storage
[301] https://github.com/ubuntu-robotics/snap_configuration/fork

At the root of our git repository, we will create a file called `key_teleop.yaml` with the following ROS 2 configuration content:

```
key_teleop:
  ros__parameters:
    forward_rate : 1.234 # our custom value
```

This configuration changed the default `forward_rate` to 1.234.

Now we can add, commit and push this change to our forked repository:

```
git add key_teleop.yaml
git commit -m "add configuration"
git push
```

Our configuration file is now available online. By looking for it on GitHub and selecting the "raw" view[302], we will get the URL of the configuration file to download later.

With the original repository, the URL is:

https://raw.githubusercontent.com/ubuntu-robotics/snap_configuration/howto/pull_configuration_from_a_server/key_teleop.yaml

Download and place the file

Now that we have a configuration file available online, we can write a script – that will be invoked by the `install` hook[303] – to download it and place it appropriately in our snap. For instance, in `$SNAP_COMMON`[304]. Since the `install` hook runs as root, we cannot use any user-specific snap writable environment. The `$SNAP_COMMON`[305] directory is the same for `root` or any user and will be read-accessible from any user.

First, let us write the script to download the configuration and place it.

We create the file `snap/local/download_config.bash` with the following content:

```
#!/usr/bin/bash

URL="https://raw.githubusercontent.com/ubuntu-robotics/snap_configuration/howto/pull_configuration_from_a_server/key_teleop.yaml"
curl $URL -o $SNAP_COMMON/up-to-date-config.yaml
```

And make it executable:

```
chmod +x snap/local/download_config.bash
```

`Curl`[306] now being a dependency, we must make it available at runtime in our snap. Let's modify the `snapcraft.yaml`:

[302] https://docs.github.com/en/enterprise-cloud@latest/repositories/working-with-files/using-files/viewing-a-file
[303] https://snapcraft.io/docs/supported-snap-hooks#heading--install
[304] https://ubuntu.com/robotics/docs/snap-data-and-file-storage
[305] https://ubuntu.com/robotics/docs/snap-data-and-file-storage
[306] https://linux.die.net/man/1/curl

```
local-files:
  plugin: dump
  source: snap/local/
+ stage-packages: [curl]
  organize:
    '*.bash': bin/
```

The `download_config` script will be called from the `install` hook[307]. Let's create the hook:

```
mkdir snap/hooks
```

Then, we create the file `snap/hooks/install` with the following content:

```
#!/usr/bin/bash

$SNAP/bin/download_config.bash
```

And make it executable:

```
chmod +x snap/hooks/install
```

Finally, we must grant network access to our hooks so it can reach our server. We can do so in the `snapcraft.yaml`:

```
 grade: devel
 confinement: strict
+hooks:
+  install:
+    plugs: [network]
```

Use the file

We are just missing a small detail: using the configuration file. We can use the downloaded configuration file by modifying the launcher `snap/local/teleop_launcher.bash`:

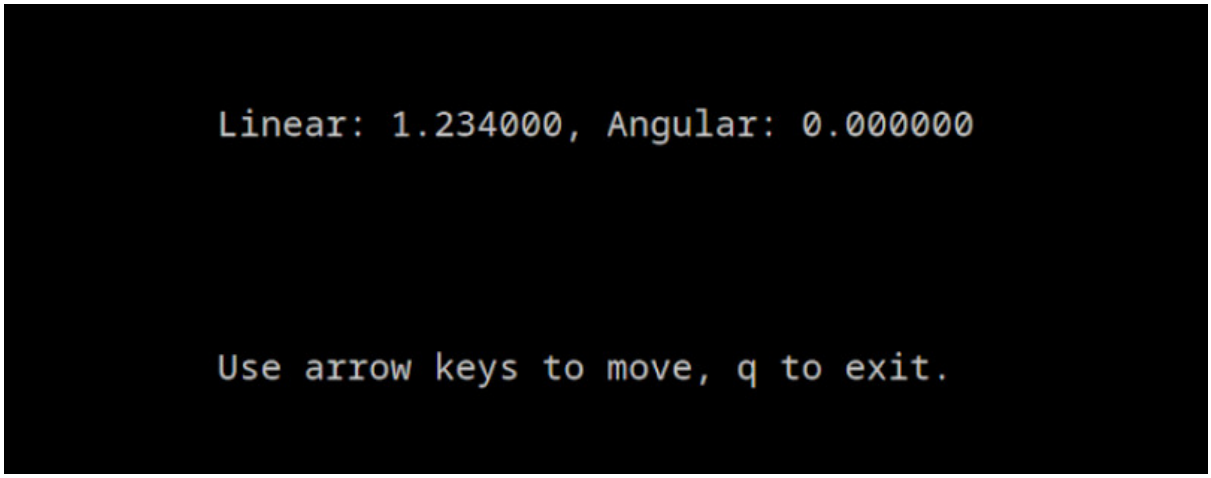
```
-ros2 run key_teleop key_teleop
+ros2 run key_teleop key_teleop --ros-args --params-file $SNAP_COMMON/up-to-date-config.
yaml
```

We can now build the snap and install it!

```
snapcraft
sudo snap install my-ros2-teleop-test_*.snap --dangerous
```

When launching our application with the command `my-ros2-teleop-test` we can see that when using the "up" arrow, our custom value 1.234 is showing!

[307] https://snapcraft.io/docs/supported-snap-hooks#heading--install



Keeping the configuration up to date

Now that our snap is pulling its configuration on install, let's keep the configuration up to date over time.

Automatic configuration update

Since we already have our script to download and place a configuration from a server, we can reuse it to call it on a regular basis to keep our configuration up to date.

To do so, we add a daemon[308] to our snap called by a timer. The daemon will be called every day at midnight so that it executes the `download_config.bash` to update our configuration. The timer syntax is described in the documentation[309]. We must change the `snapcraft.yaml` as following:

```
apps:
+ auto-update-config:
+   command: bin/download_config.bash
+   daemon: simple
+   timer: "00:00" # every day
+   plugs: [network]
```

Let's build and install our updated snap:

```
snapcraft
sudo snap install my-ros2-teleop-test_*.snap --dangerous
```

Note that this time, the `install` hook won't be called. Indeed, the snap is already installed and this only updates it.

To verify that our auto update works properly, we can change the configuration file on our server. Let's modify our configuration file `key_teleop.yaml`:

```
forward_rate : 1.234
+backward_rate: 4.321
```

[308] https://snapcraft.io/docs/services-and-daemons
[309] https://snapcraft.io/docs/timer-string-format

And upload it:

```
git add key_teleop.yaml
git commit -m "update configuration"
git push
```

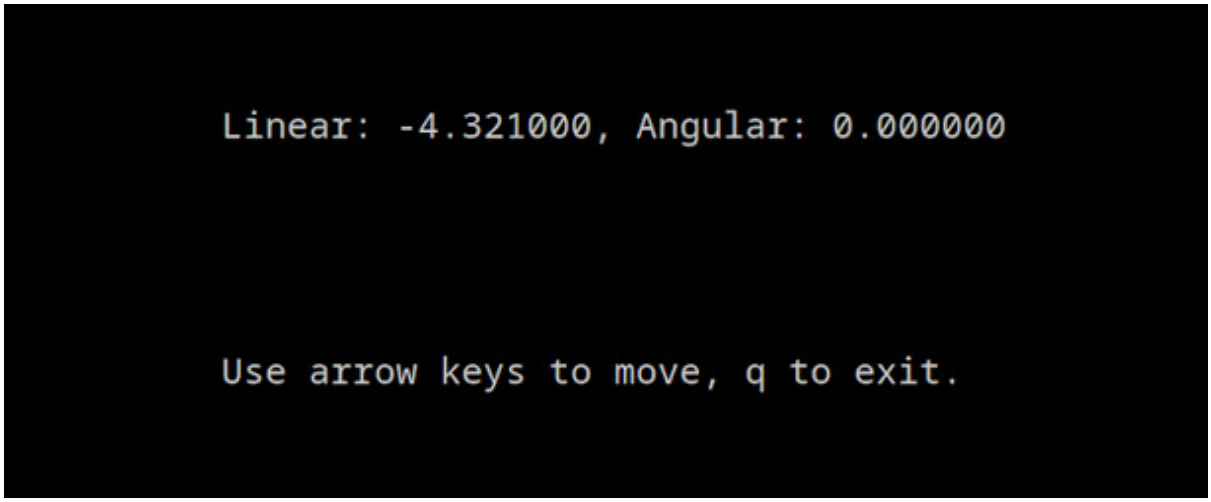
Since we won't wait until midnight to verify that our auto update works, we can trigger the daemon manually with the following command:

```
sudo snap run my-ros2-teleop-test.auto-update-config
```

And test it:

```
my-ros2-teleop-test
```

By pressing the "bottom" arrow key on the keyboard, we can see that the latest configuration was used!



Caution:

Mind that GitHub pages are cached for 5 minutes, we thus have to wait 5 minutes before being able to pull the new configuration. This is a GitHub limitation.

Our snap is not only retrieving a configuration from a server on install, but also getting the configuration updated automatically!

One may notice that the configuration will be the same for every snap installed. Or that the application won't even work if the configuration file download fails. Indeed, the configuration file's URL is hard-coded and no default file is provided. We are providing here a basic example, and the reader can customize it to their own needs or limitations. Contact us if you want more help on your robotics application!

We can find the completed example of this how-to-guide on the branch how-to/pull_configuration_from_a_server[310] of the repository.

[310] https://github.com/ubuntu-robotics/snap_configuration/tree/howto/pull_configuration_from_a_server

Configure a snap: using a content snap

When a robotics application is snapped, one might want to use it on multiple different robots.

Reusing the same snap means that we must be able to configure the snap to the specificity of a robot once installed on it.

For the rest of this guide, we will refer to the snap distributing the configuration as the configuration snap, and the one running the teleoperation application as the application snap.

We present in this guide the steps to distribute configurations with a separate, dedicated configuration snap and use it to distribute configuration files to another application snap.

Our application snap will get its configuration, not from the snapped package but from a configuration snap. This allows us to use the same snap on multiple devices with different configurations, as well as updating the configuration without having to update the application snap.

For this how-to-guide, we use the example ubuntu-robotics/snap_configuration[311].

The repository consists of the `snapcraft.yaml` file from which the snap is built, as well as a launcher script.

The repository contains a standard snap package providing the `key_teleop`[312] application from the teleop_tool[313] ROS 2 package. The goal here is to be able to configure the application using content-sharing, thus without having to update the application snap. The `key_teleop` node can be configured for its `forward_rate`, `backward_rate` and `rotational_rate` parameters. These are the parameters we will be configuring from the configuration snap.

Requirements

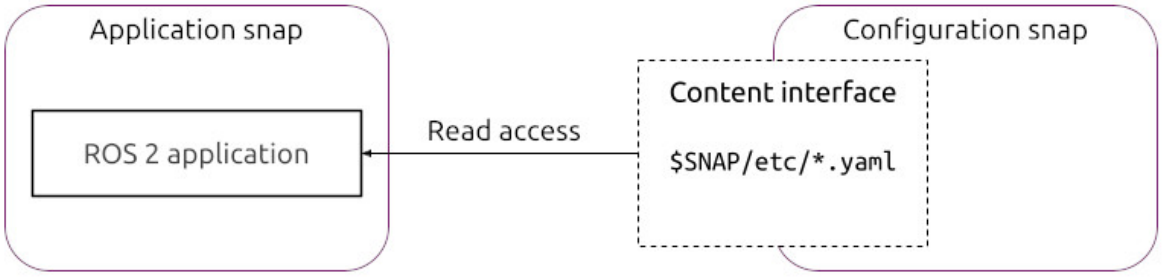
This how-to-guide is assuming that we are familiar with robotics snaps. Please refer to our tutorials[314] to learn more about robotics snaps.

An up and running Ubuntu (minimum 20.04) with snapcraft[315] installed is also required.

Configuration snap

In addition to an application snap containing our teleoperation, we will also distribute an independent configuration snap, only responsible for configuring our application. The configuration snap will simply contain the YAML file and make it available for the application snap via a content interface[316].

[311] https://github.com/ubuntu-robotics/snap_configuration
[312] https://github.com/ros-teleop/teleop_tools/tree/master/key_teleop
[313] https://github.com/ros-teleop/teleop_tools/tree/master
[314] https://ubuntu.com/robotics/docs
[315] https://snapcraft.io/snapcraft
[316] https://snapcraft.io/docs/content-interface



Create the configuration snap

The first step is to create a snap to distribute our configuration file. Let's create a new directory for the configuration snap:

```
mkdir snap
```

And then we can create the file snap/snapcraft.yaml with the following content:

```
name: my-configuration-snap
base: core22
version: '0.1'
summary: A snap to configure them all
description: |
  This snap is sharing a configuration via content-sharing.
  This way, my-ros2-teleop-test is getting configured.

grade: devel
confinement: strict

parts:
  configuration:
    plugin: dump
    source: snap/local/
    organize:
      '*.yaml': etc/
```

The snap only contains one part, dumping local YAML files inside the snap. There is no application defined, since we don't need one.

Define the configuration

Now let's create the configuration file that our snap will be sharing.

First, we create a directory:

```
mkdir snap/local
```

We then create the file `snap/local/up-to-date-config.yaml` containing:

```
key_teleop:
  ros__parameters:
    forward_rate : 1.234 # our custom value
```

This configuration file only overwrites one value from the default one. This is the configuration that our application snap will use.

Declare the content slot

The configuration snap needs now to define the content interface[317]. On the configuration snap side, we declare the `slot`[318] part of the interface. Please refer to the online documentation for the explanation of slots and plugs[319].

We modify the `snapcraft.yaml` to declare the content slot:

```
grade: devel
confinement: strict

+slots:
+  configuration:
+    interface: content
+    source:
+      read:
+        - $SNAP/etc
```

The configuration is placed in `$SNAP/etc` here since `etc/` is a standard directory for system configurations on Linux, but we could have used another directory such as `$SNAP/teleop_config`. The configuration snap is now completed, we can build it and install it:

```
snapcraft
sudo snap install my-configuration_*.snap --dangerous
```

Our configuration snap is installed and ready to provide configurations!

Application snap

At the beginning of this guide, we introduced the ubuntu-robotics/snap_configuration[320]. We start from this snap and modify it so that it uses our shared configuration. First, we clone the repository:

```
git clone https://github.com/ubuntu-robotics/snap_configuration.git
```

This repository already contains a snap package for the `key_teleop` package. There is a launcher script in `snap/local/teleop_launcher.bash` and the `snap/snapcraft.yaml`.

[317] https://snapcraft.io/docs/content-interface
[318] https://snapcraft.io/docs/interface-management#heading--slots-plugs
[319] https://snapcraft.io/docs/interfaces
[320] https://github.com/ubuntu-robotics/snap_configuration

Declare the content plug

The configuration snap is exposing a content slot containing the configuration. To access this configuration, we must declare the content `plug`.

Let's add the plug to the `snapcraft.yaml`:

```
grade: devel
confinement: strict

+plugs:
+  configuration:
+    interface: content
+    target: $SNAP/configuration
```

Additionally, we must declare that our application is using this `plug`:

```
command: bin/teleop_launcher.bash
-plugs: [network, network-bind]
+plugs: [network, network-bind, configuration]
```

Our application has now enough privilege to access the shared YAML file.

Before rebuilding our snap, we still need to actually use this file in our launcher. Let's see how to do that.

Use the content shared configuration

Even if we can access the shared configuration file, our launcher is still using the default configuration.

We will modify the launcher to make sure the shared configuration file is present and load it.

We modify the file `snap/teleop_launcher.bash` as follows:

```
#!/usr/bin/bash

+CONFIG_FILE_PATH="$SNAP/configuration/etc/up-to-date-config.yaml"

+if [[ -f $CONFIG_FILE_PATH ]]; then
-ros2 run key_teleop key_teleop --ros-args --params-file $CONFIG_FILE_PATH
+ ros2 run key_teleop key_teleop --ros-args --params-file $CONFIG_FILE_PATH
+else
+ echo "No configuration found!"
+ exit 1
+fi
```

As we can see, the content shared was entirely mounted in our target location.

Our application is now loading the YAML from our configuration snap!

Ensure content-interface connection between snaps

The content interface[321] is auto-connect[322] only when connecting two snaps from the same publisher, in other cases we really should verify the connection of the plug before launching our application.

Again, we modify our `snap/teleop_launcher.yaml` and prepend the following:

```
#!/usr/bin/bash

+if ! snapctl is-connected configuration; then
+  >&2 echo "Plug 'configuration' isn't connected, \
+  please run: snap connect ${SNAP_NAME}:configuration PROVIDER_SNAP:configuration"
+  exit 1
+fi

CONFIG_FILE_PATH="$SNAP/configuration/etc/up-to-date-config.yaml"
```

This way, the error message will be clear to the user in case the plug is not connected.

We are now ready to test, we can build and install the application snap:

```
snapcraft
sudo snap install my-ros2-teleop-test_*.snap --dangerous
```

We are now all set!

Test the snaps together

Both snaps are now installed. It's time to make sure everything works fine.

We can verify the connections state of our application snap with:

```
$ snap connections my-ros2-teleop-test
Interface     Plug                                Slot            Notes
content       my-ros2-teleop-test:configuration   -               -
network       my-ros2-teleop-test:network         :network        -
network-bind  my-ros2-teleop-test:network-bind    :network-bind   -
```

Since the content interface is not connected, we can manually connect it with the following command:

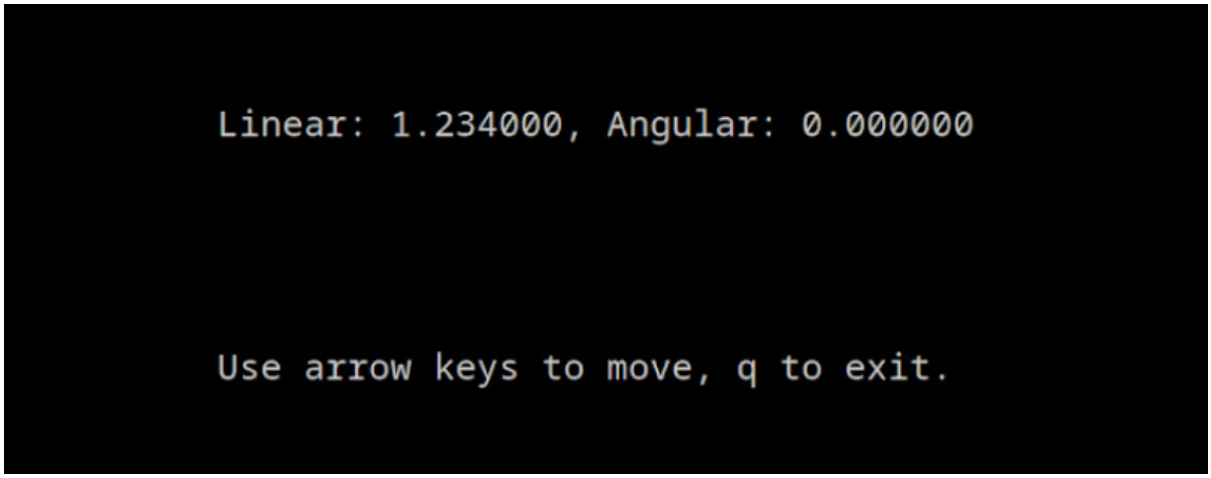
```
sudo snap connect my-ros2-teleop-test:configuration my-configuration-snap:configuration
```

We can now launch the application with:

```
my-ros2-teleop-test
```

By pressing the "up" arrow, we can observe that the value from our configuration snap (1.234) is used!

[321] https://snapcraft.io/docs/content-interface
[322] https://snapcraft.io/docs/auto-connection-mechanism



With the application snap properly using the YAML provided by our configuration snap, we can now update the configuration easily and independently of the application snap.

We can use the same application snap on multiple robots and provide a different configuration snap to different robots. We can also ship many configuration files in the configurations snap serving multiple application snaps!

We can find the complete example of this how-to-guide (application and configuration snap) on the branch how-to/content_sharing_configuration_snap[323] of the repository.

Configure a snap: Make the snap configuration overwritable

When a robotics application is snapped, one might want to use it on multiple different robots.

Reusing the same snap means that we must be able to configure the snap to the specificity of a robot once installed on it.

We present in this guide the steps to distribute a snap with a configuration that is easily overwritable on the robot.

Our snap will be distributed with the default configuration, and will copy this default configuration into a location that is common across multiple revisions[324] of the snap. This allows the user to manually modify the configuration file and keep it across updates.

For this how-to-guide, we use the example ubuntu-robotics/snap_configuration[325].

The repository consists of the *snapcraft.yaml* file from which the snap is built, as well as a launcher script.

The repository contains a standard snap package providing the `key_teleop`[326] application from the teleop_tool[327] ROS 2 package. The goal here is to be able to configure the application without having to update the snap. The `key_teleop` node can be configured for its `forward_rate`, `backward_rate` and `rotational_rate` parameters. They are the parameters present in the configuration file that are overwritable.

[323] https://github.com/ubuntu-robotics/snap_configuration/tree/howto/content_sharing_configuration_snap
[324] https://snapcraft.io/docs/glossary#heading--revision
[325] https://github.com/ubuntu-robotics/snap_configuration
[326] https://github.com/ros-teleop/teleop_tools/tree/master/key_teleop
[327] https://github.com/ros-teleop/teleop_tools/tree/master

Requirements

This how-to-guide is assuming that we are familiar with robotics snaps. Please refer to our tutorials[328] to learn more about robotics snaps.

An up and running Ubuntu (minimum 20.04) with snapcraft[329] installed is also required.

Distribute a default configuration

At the beginning of this guide, we introduced the ubuntu-robotics/snap_configuration[330]. We start from this snap and modify it so that it uses our shared configuration. First, we clone the repository:

```
git clone https://github.com/ubuntu-robotics/snap_configuration.git
```

This repository already contains a snap package for the `key_teleop` package. There is a launcher script in `snap/local/teleop_launcher.bash` and the `snap/snapcraft.yaml`.

Import the default configuration

By default, our application doesn't use any configuration file. It simply uses the default parameters hard-coded in the `key_teleop.py`[331]. We then add our YAML file and import it in our snap!

First, we must create our default YAML file `snap/local/up-to-date-config.yaml` with the following content:

```
key_teleop:
  ros__parameters:
    forward_rate : 1.0
    backward_rate: 0.5
    rotation_rate: 1.0
```

Then we modify our `snapcraft.yaml` so it also imports the configuration file into our snap:

```
local-files:
  plugin: dump
    source: snap/local/
    organize:
    '*.bash': bin/
+   '*.yaml': etc/
```

Our default configuration file will now be available in the snap.

[328] https://ubuntu.com/robotics/docs
[329] https://snapcraft.io/snapcraft
[330] https://github.com/ubuntu-robotics/snap_configuration
[331] https://github.com/ros-teleop/teleop_tools/blob/master/key_teleop/key_teleop/key_teleop.py

Use the default configuration

The `teleop_launcher.bash` is currently not using any configuration file. Let's modify the launcher so it uses the default configuration file that we added to our snap:

```
#!/usr/bin/bash

+CONFIG_FILE_PATH="$SNAP/etc/up-to-date-config.yaml"

+echo "Using config file: $CONFIG_FILE_PATH."

-ros2 run key_teleop key_teleop
+ros2 run key_teleop key_teleop --ros-args --params-file $CONFIG_FILE_PATH
```

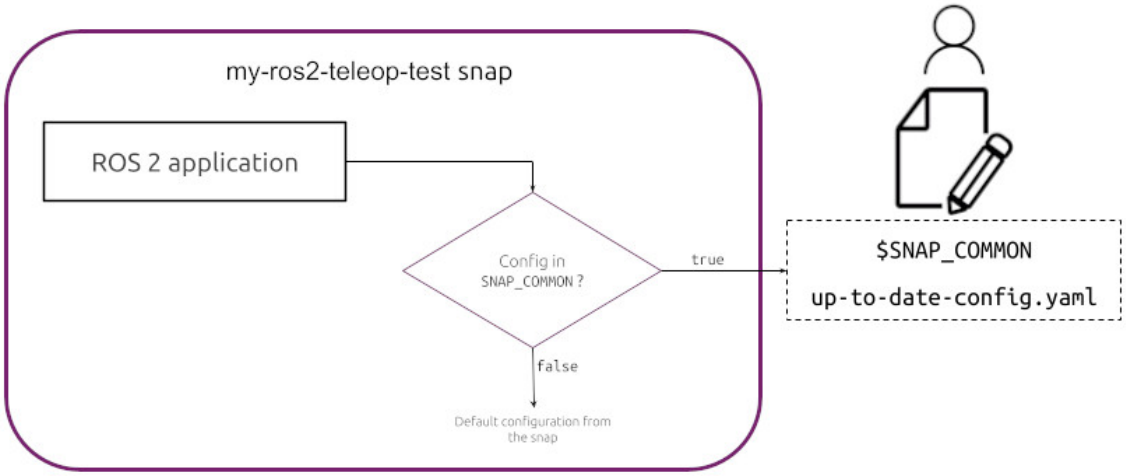
Our `key_teleop` ROS application is now using the default configuration file.

In the next section, we will see how to let the user customize this configuration file.

Add the overwritable configuration file

For the user to be able to customize the configuration file, it must be in a location that the snap can access and where a user can edit this configuration. We use `$SNAP_COMMON`[332] to store the editable YAML file.

The following diagram illustrates the configuration file workflow. Before starting the application, our snap checks if a configuration file is available in `$SNAP_COMMON`. If not, we simply use the default configuration.



Let's implement this workflow with our snap!

[332] https://ubuntu.com/robotics/docs/snap-data-and-file-storage

Provide the customizable configuration file

Launcher script

The first step is to implement the launching logic for the configuration file. We must check if the custom file exists and launch our application with the custom file if possible. To do so, we modify the `teleop_launcher.sh`:

```
#!/usr/bin/bash

+CUSTOM_CONFIG_FILE_PATH="$SNAP_COMMON/up-to-date-config.yaml"
CONFIG_FILE_PATH="$SNAP/etc/up-to-date-config.yaml"

+if [[ -f $CUSTOM_CONFIG_FILE_PATH ]]; then
+ CONFIG_FILE_PATH=$CUSTOM_CONFIG_FILE_PATH
+fi

echo "Using config file: $CONFIG_FILE_PATH."

ros2 run key_teleop key_teleop --ros-args --params-file $CONFIG_FILE_PATH
```

Our launcher script is now picking the customised configuration when available.

Install the customizable configuration at install

To provide a good starting point to the user, we add a script, so the default configuration is copied and placed in the editable area at install.

Note that the `$SNAP_COMMON` requires root privilege from the user for editing.

We create the script `snap/local/reset-overwritable-configuration.bash` with the following content:

```
#!/usr/bin/bash -e

echo "Make sure to run this application with enough privilege."

cp $SNAP/etc/up-to-date-config.yaml $SNAP_COMMON/

echo "The configuration can now be edited in the file $SNAP_COMMON/up-to-date-config.yaml.
"
```

We make the script executable:

```
chmod +x snap/local/reset-overwritable-configuration.bash
```

Now that we have a script, we simply call it from the `install` hook[333]. This way, on the installation of the snap a default configuration is placed, so the user can directly edit the YAML. We create the `install` hook:

```
mkdir snap/hooks
```

We can then add the file `snap/hooks/install`:

[333] https://snapcraft.io/docs/supported-snap-hooks#heading--install

```
#!/usr/bin/bash

$SNAP/bin/reset-overwritable-configuration.bash
```

We make the hook executable:

```
chmod +x snap/hooks/install
```

The installation part of our snap is now done. Before testing, we add a last feature to our snap in case something goes wrong with our config.

Add an application to reset the configuration

By modifying the configuration, over time, one might want to retrieve the default behaviour. To do so, we can add another application to our snap to reset the customizable configuration file. For this application, we reuse the `reset-overwritable-configuration.bash` already present in the snap. To add the application, we only modify the `snapcraft.yaml`:

```
apps:
  my-ros2-teleop-test:
    command: bin/teleop_launcher.bash
    plugs: [network, network-bind]
    extensions: [ros2-humble]
    environment:
     "LD_LIBRARY_PATH": "$LD_LIBRARY_PATH:$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/blas:
$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/lapack"

+ reset-overwritable-configuration:
+   command: bin/reset-overwritable-configuration.bash
```

This application requires sudo if called by a non-privileged user.

We are now all set to test this snap!

Testing

We can now test the overwritable configuration. First, we build and install the snap:

```
snapcraft
sudo snap install my-ros2-teleop-test_*.snap --dangerous
```

The snap being installed, we can make sure that the customizable YAML was correctly placed:

```
cat /var/snap/my-ros2-teleop-test/common/up-to-date-config.yaml
```

We then see the default content of the file:

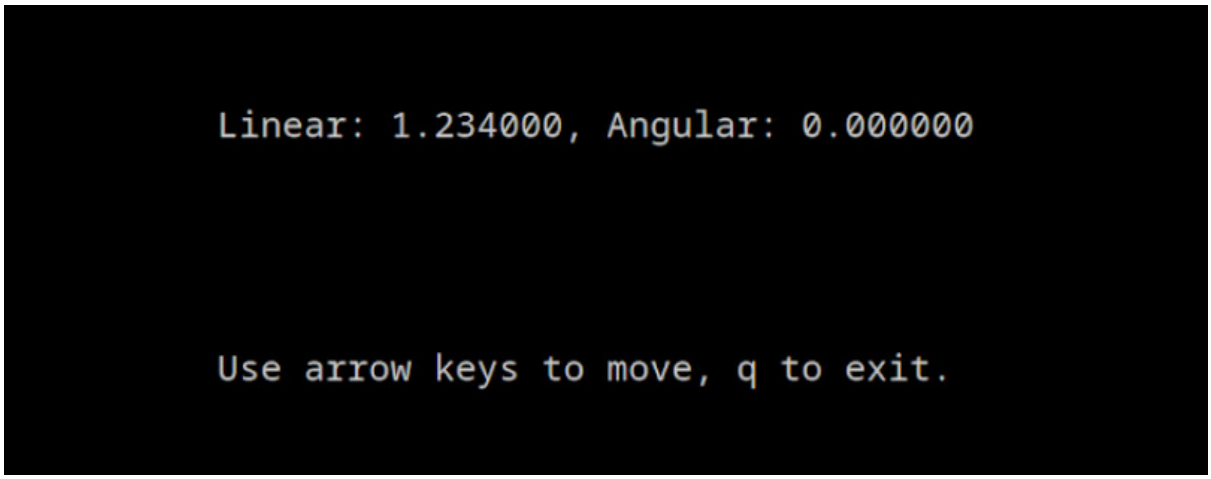
```
key_teleop:
  ros__parameters:
    forward_rate : 1.0
    backward_rate: 0.5
    rotation_rate: 1.0
```

We can edit (with root privileges) this file with a custom value (1.234 for `forward_rate`).

Now we launch the application:

```
my-ros2-teleop-test
```

And by pressing the “up” arrow key, we can see that the custom configuration was used:



Additionally, we can reset to the default configuration with the following command:

```
sudo my-ros2-teleop-test.setup-overwritable-configuration
```

With the application, snap properly using the custom YAML edited by the user. We can now update the configuration easily and independently of the snap.

We can now use the same snap on multiple robots and use a different configuration file for every device.

We can find the completed example of this how-to-guide on the branch how-to/overwritable_configuration[334] of the repository.

2.2. Operation

This section includes all guides related to daily operations for deployed robots.

2.2.1. Operation

This section includes all guides related to daily operations for deployed robots.

Warning:

Beta Notice: COS for robotics is currently in *beta*. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

[334] https://github.com/ubuntu-robotics/snap_configuration/tree/howto/overwritable_configuration

Deploy a Caddy file server for COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

In this how-to, we will deploy a simple Caddy file server in the **Canonical Observability Stack (COS) for robotics**. We therefore assume that a COS for robotics stack is up and running. You may refer to the tutorial *'Deploy COS for robotics server in the cloud'* (page 79). to do so.

By the end of this guide, We will have a cloud storage available in COS for robotics so that devices can push rosbags for later use.

Deploy Caddy

To deploy Caddy, hit the following command:

```
juju deploy ros2bag-fileserver-k8s --resource caddy-fileserver-image=ghcr.io/ubuntu-robotics/ros2bag-fileserver:dev --storage database:=10G --config ssh-port=2222
```

Once deployed, we can integrate the file storage with COS for robotics:

```
juju relate ros2bag-fileserver-k8s:ingress-tcp traefik:ingress-per-unit
juju relate ros2bag-fileserver-k8s:ingress-http traefik:ingress
juju relate ros2bag-fileserver-k8s:catalogue catalogue:catalogue
juju relate cos-registration-server:auth-devices-keys ros2bag-fileserver-k8s:auth-devices-keys
```

We can monitor the deployment, including the relations with:

```
juju status --watch 5s --color --relations
```

Once everything is green, the storage is ready to receive rosbags from the devices.

Next steps: device setup

Now that the storage is set up, let's see how to configure a device to upload rosbags: *'Send rosbags to COS for robotics'* (page 142).

Send rosbags to COS for robotics

In this how-to, we detail how to set up and configure a robot to send rosbags to the COS for robotics server.

We thus assume that a COS for robotics server is up and running. You may refer to the tutorial *'Deploy COS for robotics server in the cloud'* (page 79). to do so.

In addition, a cloud storage must be deployed and integrated with COS for robotics. Pick the tab below corresponding to the storage solution you deployed.

The following assumes that a file server is deployed and integrated with COS for robotics. You may refer to the how-to *'Deploy a file server for COS for robotics'* (page 141). to do so.

Setting up the agent

With the server side ready to receive rosbags, we shall now set up the device side.

The setup is as simple as installing the snap on the robot:

```
sudo snap install ros2-exporter-agent --channel=latest/beta
```

Once installed, we shall verify that its interfaces are connected:

```
$ snap connections ros2-exporter-agent
Interface     Plug                                    Slot
           Notes
content       ros2-exporter-agent:configuration-read   rob-cos-demo-
configuration:configuration-read  -
content       ros2-exporter-agent:rob-cos-common-read  rob-cos-data-sharing:rob-cos-
common-read       -
```

and that its services have automatically started:

```
$ snap services ros2-exporter-agent
Service                                Startup   Current   Notes
ros2-exporter-agent.auto-clean         enabled   active    timer-activated
ros2-exporter-agent.daily-rotation     enabled   active    timer-activated
ros2-exporter-agent.read-configuration  enabled   active    -
ros2-exporter-agent.recorder           enabled   active    -
ros2-exporter-agent.synchronization    enabled   active    timer-activated
```

In case the interfaces are not connected, or if you are using your own configuration snap, you can connect them with:

```
sudo snap connect ros2-exporter-agent:rob-cos-common-read <my-data-sharing>:rob-cos-
common-read
```

And:

```
sudo snap connect ros2-exporter-agent:configuration-read <my-demo-configuration>
:configuration-read
```

Once both interfaces are connected, the exporter agent will automatically start recording rosbags and send them to the file server on COS for robotics.

They can be start/stop with their respective command:

```
sudo snap stop ros2-exporter-agent.recorder
```

and:

```
sudo snap stop ros2-exporter-agent.synchronization
```

Write a configuration snap for COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

COS for robotics is composed of various snaps. These snaps must be configured for your robots, your needs and your setup.

In this how-to guide, you will learn the different steps to define your applications configuration and how to deploy them all in one snap.

This way all your COS for robotics snap services are configured with one snap.

For the rest of the guide, the presented configuration has been inspired from `rob-cos-demo-configuration`[335].

The configuration in COS for robotics

For COS for robotics snaps, the configuration is done by sharing files with a `content` interface[336].

In this how-to, you will deploy all configurations in one snap and share them with one interface. This way the various snaps will pick the configurations they need.

Snaps to configure

Before starting, let's present the snaps that are using the content interface:

[335] https://github.com/canonical/rob-cos-demo-configuration/tree/main/snap/local/configuration
[336] https://snapcraft.io/docs/content-interface

`cos-registration-agent`[337]

The snap expects a `device.yaml` file to specify device specific information to the server. This file requires the UID of the device as well as the URL of the server. Details about this file can be found on GitHub[338].

Additionally, the agent can upload application-specific data to the server:

- Grafana dashboards[339]
- Foxglove layouts[340]
- Prometheus alerts rule files[341]
- Loki alerts rule files[342]

`rob-cos-grafana-agent`[343]

This snap is wrapping the `grafana-agent`[344] in flow mode[345], hence a single `river`[346] file is necessary under the name `grafana-agent.river`.

`ros2-exporter-agent`[347]

This snap expects a `ros2-data-exporter.yaml` file to specify the ros2bags recording configurations. The YAML file is a one to one match from the snap parameters available[348].

`foxglove-bridge`[349]

This snap expects a `foxglove-bridge.yaml` file to specify the bridge configurations. The YAML file is a one to one match from the `foxglove-bridge` configuration[350].

[337] https://snapcraft.io/cos-registration-agent
[338] https://github.com/canonical/cos-registration-agent?tab=readme-ov-file#config
[339] https://grafana.com/grafana/dashboards/
[340] https://docs.foxglove.dev/docs/visualization/layouts
[341] https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/
[342] https://grafana.com/docs/loki/latest/alert/
[343] https://snapcraft.io/rob-cos-grafana-agent
[344] https://grafana.com/docs/agent/latest/
[345] https://grafana.com/docs/agent/latest/flow/
[346] https://grafana.com/docs/agent/latest/flow/concepts/config-language/
[347] https://snapcraft.io/ros2-exporter-agent
[348] https://github.com/canonical/ros2-exporter-agent/?tab=readme-ov-file#snap-parameters
[349] https://snapcraft.io/foxglove-bridge
[350] https://github.com/foxglove/ros-foxglove-bridge?tab=readme-ov-file#configuration

The content sharing interface

In order to have a match between what the different snaps are expecting and your configurations, you must follow the format from the content sharing interface.

Expose a content interface slot[351] with all the configuration placed at the root of the slot's content, as shown below:

```
slots:
  configuration-read:
    interface: content
    read:
      - path/to/configurations
```

Configuration files layout

Once the configuration is accessible to a snap, the various services are expecting the configuration files in specific locations.

All the configuration files are relative to the root of the configuration.

cos-registration-agent

```
.
├── device.yaml
├── foxglove_layouts
│   ├── layout1.json
│   └── layout2.json
├── grafana_dashboards
│   ├── dashboard1.json
│   └── dashboard1.json
├── loki_alert_rules
│   ├── alert1.rules
│   └── alert2.rules
└── prometheus_alert_rules
    ├── alert1.rules
    └── alert2.rules
```

rob-cos-grafana-agent

```
.
└──  grafana-agent.river
```

[351] https://snapcraft.io/docs/interface-management

ros2-exporter-agent

```
.
└── ros2-data-exporter.yaml
```

foxglove-bridge

```
.
└── foxglove-bridge.yaml
```

Write the snap

Now, with a clear view of the content interface as well as the various configurations, you can start creating the snap.

Define the `snapcraft.yaml`

In a new directory, start by creating a `snapcraft.yaml` with the following content:

```
name: my-rob-cos-configuration
base: core24
version: git
summary: A snap for my custom configuration of the rob cos snaps.
description: |
  A snap for my configuration of the rob cos snaps on the device.

  This snap offers a custom configuration for a rob cos device.
  It also offers a content sharing interface, to allow snaps on a device that is meant
  to work with the rob cos ecosystem to easily get configured.

  It offers a slot called configuration-read that allows plugged snaps to read data
  stored in $SNAP/etc/configuration.

  Usage:

  Connect as follows:
  sudo snap connect rob-cos-snap:configuration-read my-rob-cos-
configuration:configuration-read

grade: stable
confinement: strict
```

Add the configurations to the snap

Under the folder `snap/local/configuration`, add all the configuration files.

Next, append the following part to your `snapcraft.yaml`:

```
parts:
  configuration:
    plugin: dump
    source: snap/local/configuration/
    organize:
      '*': /etc/configuration/
```

The configurations files will now be added to your snap.

Expose the configuration with the content interface

Expose the configurations shipped in the snap by adding the content interface.

To do so, add the following to your `snapcraft.yaml`:

```
slots:
  configuration-read:
    interface: content
    read:
      - $SNAP/etc/configuration
```

Use the configuration snap

Before using the configuration snap, build and install it:

```
snapcraft
snap install my-rob-cos-configuration*.snap --dangerous
```

It can now be connected to the various applications you want to configure. As an example, connect it to the `cos-registration-agent` with:

```
snap connect cos-registration-agent:configuration-read my-rob-cos-
configuration:configuration-read
```

This connection will automatically trigger the registration on the `cos-registration-agent` side.

You can apply the same connections to all the snaps.

Additionally, you could add more features to your configuration snap to make it smarter.

A complete example of such snap can be found on GitHub: github.com/canonical/rob-cos-demo-configuration[352].

You can also find a collection of example configurations for the various COS for robotics server applications on GitHub: github.com/canonical/ROB-COS-configurations[353].

[352] https://github.com/canonical/rob-cos-demo-configuration/tree/main
[353] https://github.com/canonical/ROB-COS-configurations/

Configure Alertmanager to send email alerts

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

Once an alert is triggered by Prometheus or Loki, the alert is received and distributed by Alertmanager.

Alertmanager can be configured to send notifications on multiple receivers[354]

In this how-to guide, you will configure the Alertmanager application to send emails on alerts.

Alertmanager configuration

The configuration of Alertmanager is done by a YAML file later loaded in the application. The YAML file follows the scheme defined in the official documentation[355].

Get the app password

Before configuring the Alertmanager, you must get a token called “app password”, to let Alertmanager send emails from your account.

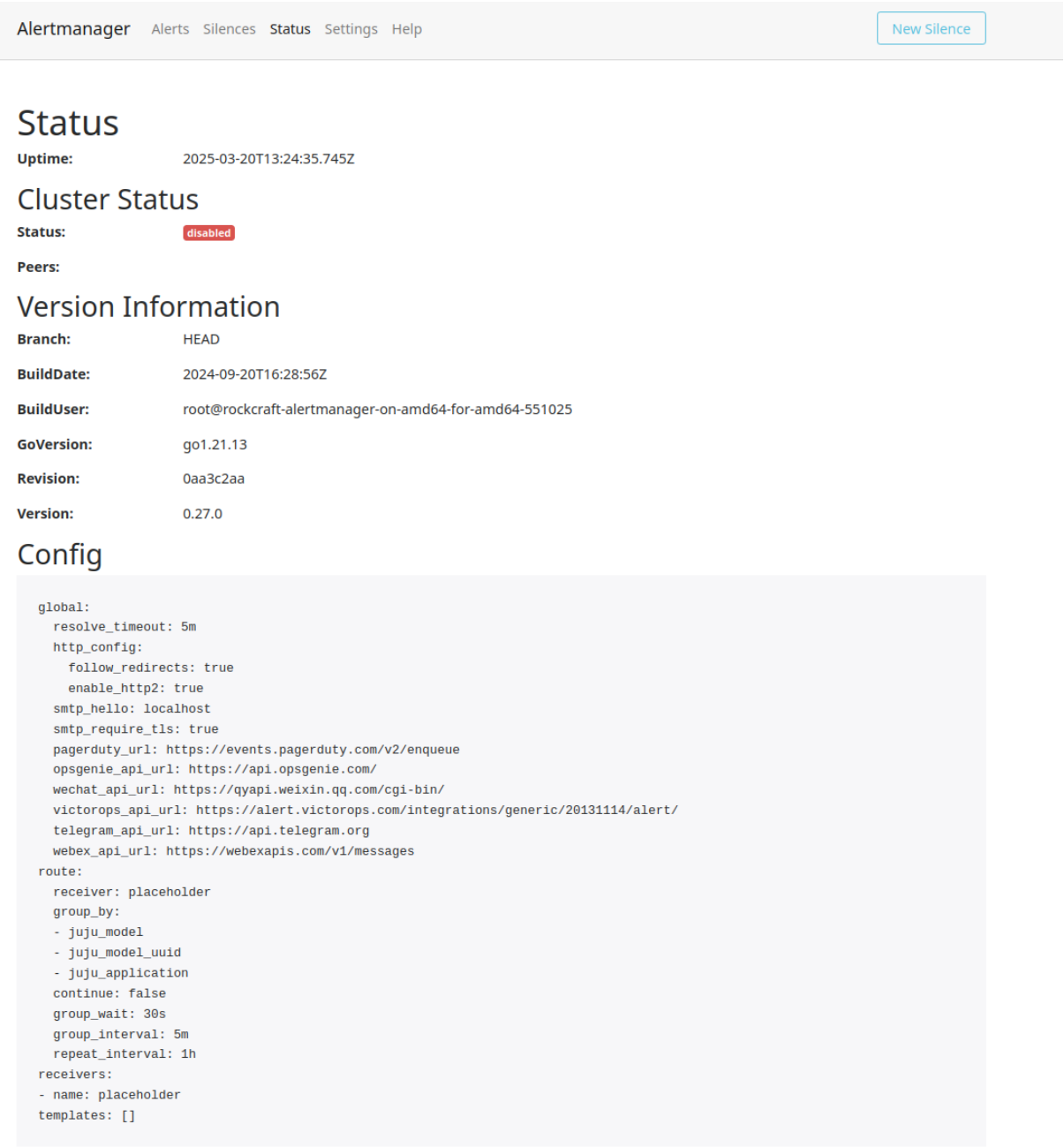
While logged into your Gmail account, go to the App passwords page[356] and generate a new app password. visit the Gmail app password page[357], and create a new app password.

Make sure to copy the created code, and place it in the `alert-manager.yaml` configuration file.

Write the configurations file for Alertmanager

Alertmanager is configured via a YAML file. To get a starting point for the file, you can get the actual `alert-manager.yaml` by going on your Alertmanager instance. In the “status” tab, you will find at the bottom, the actual config to use as a starting point:

[354] https://prometheus.io/docs/alerting/latest/configuration/#general-receiver-related-settings
[355] https://prometheus.io/docs/alerting/latest/configuration/#file-layout-and-global-settings
[356] https://myaccount.google.com/apppasswords
[357] https://myaccount.google.com/apppasswords



In addition to the default configuration, you will add two entry: `route` and `receivers`.

Route

The route attribute[358] defines where and how the alerts are dispatched. Here, the idea is to group alerts not only from devices but also from Juju applications. The alerts are sent in batches in order to prevent continuous notifications.

Additionally, the route defines to which receiver the alerts are then forwarded. In this case, it will be emails.

The route is defined as follows:

[358] https://prometheus.io/docs/alerting/latest/configuration/#route-related-settings

```
route:
  receiver: 'email'
  group_by:
    - juju_model_uuid
    - juju_application
    - juju_model
  continue: false
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 1h
  routes:
    - receiver: 'email'
      group_wait: 10s
      match_re:
        severity: critical|warning|none
      continue: true
```

Receiver

The receiver attribute[359] defines the integration with specific receivers (email, chat, webhooks, etc). Receivers usually require an authentication token, which, in the case of Gmail, is the app password.

In the receiver you must declare the sender's email (the one associated to the app password), and the recipient's email.

The receiver is defined as follows:

```
receivers:
  - name: 'email'
    email_configs:
      - to: 'all@my-company.com'
        from: 'user.name@gmail.com'
        smarthost: 'smtp.gmail.com:587'
        auth_username: 'user.name@gmail.com'
        auth_password: 'APP_TOKEN'
        auth_identity: 'user.name@gmail.com'
```

Apply the Alertmanager configuration

With the previously defined Alertmanager configuration in an `alert-manager.yaml`, you can apply the configuration to the Juju application.

The `alertmanager-k8s` declares a `config_file` configuration[360] that can be used with the following command:

```
juju config alertmanager config_file='@./alert-manager.yaml'
```

You can verify that the configuration got applied by going in the status tab of Alertmanager.

[359] https://prometheus.io/docs/alerting/latest/configuration/#receiver-integration-settings
[360] https://charmhub.io/alertmanager-k8s/configurations#config_file

Since the Alertmanager Juju application has a watchdog alert, you will receive your first alert right after.

Email alert template

Now that the configuration of the application is done, you can also configure the email notification template[361] used for sending notifications. The template lets you customize both the appearance and the data in your notification.

In the case of an email notification, the template is an HTML file. You can get the default template from GitHub[362] with the following command:

```
wget https://raw.githubusercontent.com/prometheus/alertmanager/refs/heads/main/template/email.html
```

Similarly to the `config_file` configuration, `alertmanager-k8s` has a `template-config`[363]

You can then apply the new template with the following command:

```
juju config alertmanager templates_file='@./email.html'
```

You must now update your Alertmanager configuration, so the receiver uses the new template:

```
receivers:
  - name: 'email'
    email_configs:
      - to: 'all@my-company.com'
        from: 'user.name@gmail.com'
        smarthost: 'smtp.gmail.com:587'
        auth_username: 'user.name@gmail.com'
        auth_password: 'APP_TOKEN'
        auth_identity: 'user.name@gmail.com'
        html: '{{ template "templates.tmpl" . }}'
```

Make sure to run the `juju config` command again with the updated `alert-manager.yaml`.

Now, when an alert will trigger you will receive a customized notification!

Observe COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

COS for robotics provides valuable insights about one's fleet of devices. Maybe more importantly it also alerts the fleet operators should anything dysfunction. As such, COS for robotics

[361] https://prometheus.io/docs/alerting/latest/notifications/
[362] https://github.com/prometheus/alertmanager/blob/main/template/email.html
[363] https://charmhub.io/alertmanager-k8s/configurations#templates_file

can be seen as a critical piece of infrastructure. And while it is resilient, it can be subject to failure and disrupts the monitoring of the fleet.

For this reason, COS for robotics is itself observable as well, using most of the tools already used for observing the robots fleets. While COS for robotics could perfectly observe itself, this wouldn't make much sense in case of a large outage. Instead, we recommend deploying a separate COS Lite[364] stack in production which responsibility is to monitor COS for robotics.

We assume hereafter that both COS for robotics and COS Lite are deployed and set up. COS Lite deployment is very similar to that of COS for robotics and a tutorial can be found on COS Lite documentation website[365]. We also assume that COS Lite deployment includes support for distributed tracing support with Tempo[366] as well as the Blackbox Exporter[367] for probing of endpoints.

Note:

There are multiple strategies for the topology of this double deployment. For the sake of this how-to, we assume that COS Lite and COS for robotics live in their respective models and that both are controlled by the same Juju controller. Make sure to read Juju's documentation about coss-model relations[368] as we are using this feature and since it may impact the very boostrapping of the controllers depending on the topology.

[368] https://documentation.ubuntu.com/juju/3.6/reference/relation/#cross-model-relation

Prelude

For good measure, let us start by making sure that everything is fine. We first check the COS for robotics stack:

```
$ juju status --model robcos-model
Model         Controller         Cloud/Region        Version  SLA          Timestamp
robcos-model  robcos-controller  microk8s/localhost  3.6.4    unsupported  14:39:49Z

App                     Version  Status  Scale  Charm                        Channel
  Rev  Address         Exposed  Message
alertmanager              0.27.0   active      1  alertmanager-k8s             latest/
stable  156  10.152.183.200  no
catalogue                        active      1  catalogue-k8s                latest/
stable   81  10.152.183.111  no
cos-registration-server          active      1  cos-registration-server-k8s  latest/edge
    8  10.152.183.235  no
foxglove-studio                  active      1  foxglove-studio-k8s          latest/edge
    1  10.152.183.96   no
grafana                   9.5.3    active      1  grafana-k8s                  latest/
stable  139  10.152.183.182  no
loki                      2.9.6    active      1  loki-k8s                     latest/
stable  187  10.152.183.140  no
prometheus                2.52.0   active      1  prometheus-k8s               latest/
stable  232  10.152.183.148  no
```

(continues on next page)

[364] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[365] https://charmhub.io/topics/canonical-observability-stack/tutorials
[366] https://charmhub.io/topics/canonical-observability-stack/how-to/add-distributed-tracing
[367] https://charmhub.io/blackbox-exporter-k8s/docs/using

(continued from previous page)

```
ros2bag-fileserver                active      1  ros2bag-fileserver-k8s       latest/edge
    3  10.152.183.83   no
traefik                  2.11.0   active      1  traefik-k8s                  latest/
stable  234  10.152.183.242  no       Serving at 100.83.155.248

Unit                        Workload  Agent  Address       Ports  Message
alertmanager/0*             active    idle   10.1.105.131
catalogue/0*                active    idle   10.1.105.135
cos-registration-server/0*  active    idle   10.1.105.167
foxglove-studio/0*          active    idle   10.1.105.173
grafana/0*                  active    idle   10.1.105.149
loki/0*                     active    idle   10.1.105.136
prometheus/0*               active    idle   10.1.105.155
ros2bag-fileserver/0*       active    idle   10.1.105.139
traefik/0*                  active    idle   10.1.105.133         Serving at 100.83.155.
248

Offer    Application  Charm        Rev  Connected  Endpoint       Interface      Role
traefik  traefik      traefik-k8s  234  1/1        traefik-route  traefik_route  provider
```

The COS for robotics stack is ready. What about the COS Lite stack:

```
$ juju status --model cos-model
Model      Controller         Cloud/Region       Version  SLA          Timestamp
cos-model  robcos-controller  cos-k8s/localhost  3.6.4    unsupported  09:55:08Z

App           Version                Status  Scale  Charm                  Channel
Rev  Address         Exposed  Message
alertmanager  0.27.0                 active      1  alertmanager-k8s       latest/stable
156  10.152.183.87   no
blackbox      0.24.0                 active      1  blackbox-exporter-k8s  latest/stable
25  10.152.183.83   no
catalogue                            active      1  catalogue-k8s          latest/stable
81  10.152.183.144  no
grafana       9.5.3                  active      1  grafana-k8s            latest/stable
139  10.152.183.247  no
loki          2.9.6                  active      1  loki-k8s               latest/stable
187  10.152.183.62   no
minio         res:oci-image@220b31a  active      1  minio                  ckf-1.9/edge
419  10.152.183.64   no
prometheus    2.52.0                 active      1  prometheus-k8s         latest/stable
232  10.152.183.50   no
s3                                   active      1  s3-integrator          latest/edge
145  10.152.183.29   no
tempo                                active      1  tempo-coordinator-k8s  latest/edge
73  10.152.183.237  no       metrics-generator disabled. Add a relation over send-remote-
write
tempo-worker  2.7.1                  active      1  tempo-worker-k8s       latest/edge
53  10.152.183.161  no       metrics-generator disabled. No prometheus remote-write
relation configured on the coordinator
traefik       2.11.0                 active      1  traefik-k8s            latest/stable
234  10.152.183.73   no       Serving at 100.83.164.181

Unit            Workload  Agent  Address       Ports          Message
alertmanager/0*  active    idle   10.1.128.139
blackbox/0*      active    idle   10.1.128.133
```

(continues on next page)

(continued from previous page)

```
catalogue/0*     active    idle   10.1.128.141
grafana/0*       active    idle   10.1.128.155
loki/0*          active    idle   10.1.128.137
minio/0*         active    idle   10.1.128.132  9000-9001/TCP
prometheus/0*    active    idle   10.1.128.138
s3/0*            active    idle   10.1.128.145
tempo-worker/0*  active    idle   10.1.128.134                metrics-generator disabled.
No prometheus remote-write relation configured on the coordinator
tempo/0*         active    idle   10.1.128.140                metrics-generator disabled.
Add a relation over send-remote-write
traefik/0*       active    idle   10.1.128.191                Serving at 100.83.164.181
```

Alright, we're all setup and we can get to relating the stacks.

Deploy the Grafana agent

With both COS for robotics and COS Lite deployed in their respective models, we must now 'relate' them through Juju relations[369].

Since the stacks live in separate models, we must establish a so called coss-model relations[370]. These are two folds, firstly, we need to expose some applications from one model to the other, secondly, we can relate applications as we normally would using Juju.

To ease the setup, we deploy the Grafana agent[371] in the COS for robotics model. This allows for connecting our applications in COS for robotics to the Grafana instance in COS Lite in a simpler manner as we will see later on. Not only is this simplifying the deployment but it also offer more flexibility when it comes to modifying the overall deployment topology. This setup is depicted in the following diagram:

Fig. 1: A bi-model deployment of COS Lite observing COS for robotics.

To deploy the agent, issue the command:

```
juju deploy grafana-agent-k8s
```

Relating to the agent

Now that the agent is deployed, we can connect all the observability endpoints to it. And there are quite a few of them:

```
# alermanager
juju relate alermanager:grafana-dashboard grafana-agent:grafana-dashboards-consumer
juju relate alermanager:self-metrics-endpoint grafana-agent:metrics-endpoint
juju relate alermanager:tracing grafana-agent:tracing-provider
# catalogue
juju relate catalogue:tracing grafana-agent:tracing-provider
# cos-registration-server
```

(continues on next page)

[369] https://documentation.ubuntu.com/juju/latest/reference/relation/
[370] https://documentation.ubuntu.com/juju/3.6/reference/relation/#cross-model-relation
[371] https://charmhub.io/grafana-agent-k8s

(continued from previous page)

```
juju relate cos-registration-server:logging grafana-agent:logging-provider
juju relate cos-registration-server:tracing grafana-agent:tracing-provider
juju relate cos-registration-server:grafana-dashboard grafana-agent:grafana-dashboards-
consumer
# foxglove-studio
juju relate foxglove-studio:logging grafana-agent:logging-provider
juju relate foxglove-studio:tracing grafana-agent:tracing-provider
juju relate foxglove-studio:grafana-dashboard grafana-agent:grafana-dashboards-consumer
# grafana
juju relate grafana:charm-tracing grafana-agent:tracing-provider
juju relate grafana:workload-tracing grafana-agent:tracing-provider
juju relate grafana:metrics-endpoint grafana-agent:metrics-endpoint
# loki
juju relate loki:metrics-endpoint grafana-agent:metrics-endpoint
juju relate loki:grafana-dashboard grafana-agent:grafana-dashboards-consumer
juju relate loki:charm-tracing grafana-agent:tracing-provider
juju relate loki:workload-tracing grafana-agent:tracing-provider
# prometheus
juju relate prometheus:self-metrics-endpoint grafana-agent:metrics-endpoint
juju relate prometheus:grafana-dashboard grafana-agent:grafana-dashboards-consumer
juju relate prometheus:charm-tracing grafana-agent:tracing-provider
juju relate prometheus:workload-tracing grafana-agent:tracing-provider
# traefik
juju relate traefik:metrics-endpoint grafana-agent:metrics-endpoint
juju relate traefik:grafana-dashboard grafana-agent:grafana-dashboards-consumer
juju relate traefik:charm-tracing grafana-agent:tracing-provider
juju relate traefik:workload-tracing grafana-agent:tracing-provider
```

With all the relations established to the agent, we can now move on to exposing the COS Lite endpoints to the COS for robotics model in order to connect them.

Making an offer

As we mentioned earlier, the first step is to issue 'offers'. To do so, issue the commands:

```
juju offer cos-model.grafana:grafana-dashboard cos-grafana
juju offer cos-model.loki:logging cos-loki
juju offer cos-model.prometheus:receive-remote-write cos-prometheus
juju offer cos-model.tempo:tracing cos-tempo
juju offer cos-model.blackbox:probes cos-blackbox
```

These create the offers from the `cos-model` model, where the COS Lite lives, to be consumed in another model.

They are then 'consumed' on the COS for robotics model with:

```
juju consume cos-model.cos-grafana
juju consume cos-model.cos-loki
juju consume cos-model.cos-prometheus
juju consume cos-model.cos-tempo
juju consume cos-model.cos-blackbox
```

Once consumed, they appears in the `juju status` output as SAAS entries:

```
$ juju status
Model         Controller         Cloud/Region        Version  SLA          Timestamp
robcos-model  robcos-controller  microk8s/localhost  3.6.4    unsupported  12:44:19Z

SAAS            Status  Store              URL
cos-blackbox    active  robcos-controller  admin/cos-model.cos-blackbox
cos-grafana     active  robcos-controller  admin/cos-model.cos-grafana
cos-loki        active  robcos-controller  admin/cos-model.cos-loki
cos-prometheus  active  robcos-controller  admin/cos-model.cos-prometheus
cos-tempo       active  robcos-controller  admin/cos-model.cos-tempo
...
```

Relating to COS Lite

With the COS Lite endpoints now available in the COS for robotics model, all we have to do is to relate them to the Grafana agent.

We proceed with:

```
juju relate grafana-agent:grafana-dashboards-provider cos-grafana:grafana-dashboard
juju relate grafana-agent:logging-consumer cos-loki:logging
juju relate grafana-agent:send-remote-write cos-prometheus:receive-remote-write
juju relate grafana-agent:tracing cos-tempo:tracing
```

We also relate our applications to Blackbox:

```
juju relate cos-registration-server:probes cos-blackbox:probes
juju relate foxglove-studio:probes cos-blackbox:probes
```

Voila.

The COS for robotics stack is now observable by COS Lite.

Set up Blackbox Exporter to monitor COS for robotics devices

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

Blackbox Exporter[372] allows active monitoring of devices and endpoints by probing them over protocols such as HTTP, TCP and ICMP.

This guide explains how to use the blackbox-exporter-k8s-operator charm[373] to monitor a fleet of COS for robotics devices and display their status in a Grafana dashboard.

Blackbox Exporter works by receiving a list of targets to probe and returning metrics about their availability, which can then be scraped by Prometheus and visualized in Grafana.

[372] https://github.com/prometheus/blackbox_exporter
[373] https://github.com/canonical/blackbox-exporter-k8s-operator

In this guide, we assume that COS for robotics has been deployed following the instructions available in *this tutorial* (page 79).

Deploy Blackbox Exporter charm

Let's start by deploying the Blackbox Exporter charm as part of our COS for robotics with:

```
juju deploy blackbox-exporter-k8s
```

Verify the deployment status with:

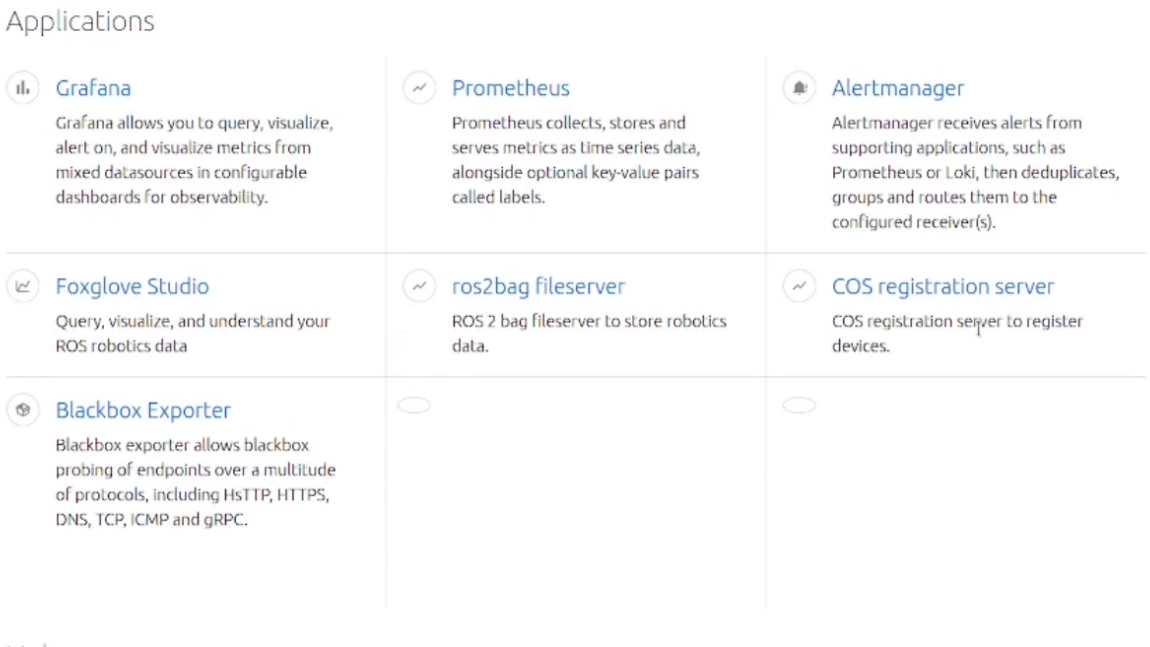
```
juju status
```

Relating Blackbox Exporter to COS for robotics charms

Once the `blackbox-exporter-k8s` app is in `active` state, relate it to the following services:

```
juju relate blackbox-exporter-k8s:self-metrics-endpoint prometheus:metrics-endpoint
juju relate blackbox-exporter-k8s:ingress traefik:ingress
juju relate blackbox-exporter-k8s:catalogue catalogue:catalogue
```

You should now see the `blackbox-exporter-k8s` app listed in the Catalogue UI as follows:



Probing devices

The COS registration server holds the list of devices to be probed by Blackbox. Any device registered is then automatically targeted by Blackbox for ICMP probing.

To enable this, relate Blackbox Exporter to the registration server as follows:

```
juju relate blackbox-exporter-k8s:probes cos-registration-server:probes-devices
```

To confirm everything is working, open the Blackbox panel in the Catalogue UI to check the list of probed devices as shown below:

Blackbox Exporter

Probe prometheus.io for http_2xx

Debug probe prometheus.io for http_2xx

Metrics

Configuration

Recent Probes

Module	Target	Result	Debug
icmp	10.166.108.1	Success	Logs

Next, let's configure a custom Grafana dashboard to visualize the probed devices with labels and status indicators.

Deploy COS configuration charm

By default, the Blackbox Exporter charm includes a standard Grafana dashboard template[374]. However, this dashboard does not include the visualization of devices UUID as labels.

To visualize the status of devices along with their UUID, we are going to use the cos-configuration-k8s operator[375] charm, which enables syncing and applying custom dashboards from a Git repository.

A custom Grafana dashboard template is available at https://github.com/canonical/robotics-cos-k8s-config/blob/feat/blackbox-grafana-dashboard/dashboards/grafana/blackbox/blackbox.json.tmpl.

Let's now deploy the configuration charm with the appropriate flags to pull in this dashboard:

[374] https://github.com/canonical/blackbox-exporter-k8s-operator/blob/main/src/grafana_dashboards/blackbox.json.tmpl

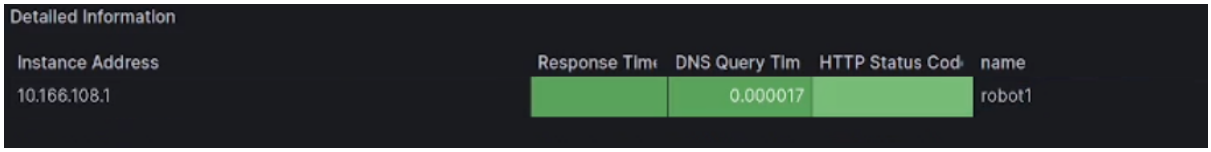
[375] https://github.com/canonical/cos-configuration-k8s-operator

```
juju deploy cos-configuration-k8s \
  --config git_repo=https://github.com/ubuntu-robotics/robotics-cos-k8s-config.git \
  --config git_branch=feat/blackbox-grafana-dashboard \
  --config git_depth=1 \
  --config grafana_dashboards_path=/var/lib/juju/agents/unit-cos-configuration-k8s-0/
dashboards/grafana/blackbox
```

Finally, relate the configuration charm to Grafana to visualize the dashboard:

```
juju relate cos-configuration-k8s grafana
```

Now, by navigating to Grafana via the Catalogue UI, select the Blackbox Exporter dashboard, and you will see a list of the probed devices with their UUID and the status color:



And that's it! You're now all set to easily monitor the health of your devices!

Enable TLS encryption in COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

COS for robotics offers flexible deployment options, allowing for either an unencrypted configuration or a more secure setup with TLS termination enabled. With TLS termination enabled, the Traefik charm acts as the TLS termination point by integrating to a self signed certificate charm or to an external certificate authority charm.

This guide details the deployment of COS for robotics with TLS termination. It outlines the necessary steps to set up a COS for robotics device, enabling it to successfully register and establish secure communication with the server. The first part will detail how to setup the Server side, while the second will focus on the device.

Server side

First, we will proceed by setting up TLS on the server side. This guide assumes that COS for robotics is up an running. You can follow the *main tutorial* (page 79) to do so. From there, we'll set up TLS for the deployment using the self-signed-certificates charm[376], which provides self-signed X.509 certificates to charms.

To keep things organized, We deploy the TLS components in a separate Juju model called `tls`. Let us start by creating this model:

[376] https://charmhub.io/self-signed-certificates

```
juju add-model tls
juju switch tls
```

Deploy the charm in the `tls` model with:

```
juju deploy self-signed-certificates --channel=1/stable
```

This charm will manage the certificate authority (CA) and issue self-signed certificates to Traefik and devices. In order to make the TLS charm endpoints available to other models, we need to setup cross-models relations[377]. This is achieved by offering the charm relations:

```
juju offer self-signed-certificates:send-ca-cert send-ca-cert
juju offer self-signed-certificates:certificates certificates
```

The output will look as follows:

```
Application "self-signed-certificates" endpoints [send-ca-cert] available at "admin/tls.
send-ca-cert"
Application "self-signed-certificates" endpoints [certificates] available at "admin/tls.
certificates"
```

In this way, those relations can be consumed by the charms in our COS for robotics model, such as Traefik.

Now, let's switch back to the `cos-robotics-model` and consume the relations:

```
juju switch cos-robotics-model
juju consume admin/tls.certificates
juju consume admin/tls.send-ca-cert
```

Finally, integrate the relations with Traefik and Grafana to enable TLS:

```
juju integrate traefik certificates
juju integrate traefik send-ca-cert
juju integrate grafana send-ca-cert
```

The `send-ca-cert` relation provides the public root CA certificate that is used to sign all certificates issued by the `self-signed-certificates` charm. Prometheus and Loki are accessed by Grafana through Traefik, which means Grafana must trust the CA that issued Traefik's certificates. Without the `send-ca-cert` relation, Grafana would see Traefik's certificates as not trusted, preventing secure communication from being established.

That's it for the server side, you can verify that Traefik is now serving endpoints on `https` with:

```
juju run traefik/leader show-proxied-endpoints
```

The output should look like this:

```
Running operation 12 with 1 task
  - task 13 on unit-traefik-0

Waiting for task 13...
proxied-endpoints: '{"traefik": {"url": "https://10.239.43.60"}, "ros2bag-fileserver/0":
```

(continues on next page)

[377] https://documentation.ubuntu.com/juju/3.6/reference/relation/

(continued from previous page)

```
  {"url": "10.239.43.60:2222"}, "prometheus/0": {"url": "https://10.239.43.60/cos-
robotics-model-prometheus-0"},
  "loki/0": {"url": "https://10.239.43.60/cos-robotics-model-loki-0"}, "alertmanager":
  {"url": "https://10.239.43.60/cos-robotics-model-alertmanager"}, "catalogue": {"url":
  "https://10.239.43.60/cos-robotics-model-catalogue"}, "foxglove-studio": {"url":
  "https://10.239.43.60/cos-robotics-model-foxglove-studio"}, "cos-registration-server":
  {"url": "https://10.239.43.60/cos-robotics-model-cos-registration-server"}, "ros2bag-
fileserver":
  {"url": "https://10.239.43.60/cos-robotics-model-ros2bag-fileserver"}}'
```

We can see in this output that the proxied-endpoints URLs are using the https protocol.

Device side

When TLS termination is enabled, each device must trust the certificate provided by Traefik for the registration and communication to work. Below, the steps to install the self-signed certificate generated on the server onto the device are outlined.

Set the certificate on the device

First, retrieve the CA certificate from the `self-signed-certificates` charm on the server:

```
juju run self-signed-certificates/0 get-ca-certificate
```

The output will show the public CA as follows:

```
Running operation 11 with 1 task
  - task 12 on unit-self-signed-certificates-0

Waiting for task 12...
ca-certificate: |
  -----BEGIN CERTIFICATE-----
  MIIDZzCCA....
  -----END CERTIFICATE-----
```

Copy the certificate content and save it into a file ending with `.crt`, for example:

```
nano traefik-ca.crt
```

Warning:

Note: When copying the certificate from the output above, Make sure you *do not* include any leading spaces before the certificate lines. The file should start exactly with `-----BEGIN CERTIFICATE-----` at the beginning of the line. If the certificate lines are indented, the certificate will be invalid.

Move this file into the system's trusted CA directory:

```
sudo mv traefik-ca.crt /usr/local/share/ca-certificates/
```

Finally activate the certificate by updating the certificates trust store with:

```
sudo update-ca-certificates
```

In this way, the certificate will be available system wide, and the agents running on the robot can trust the certificate.

Verify connectivity

Now that the certificate is installed, verify that you can successfully connect over HTTPS:

```
curl -vvv https://<cos-robotics-server-ip>/cos-robotics-model-catalogue
```

If the setup is correct, you should see a valid TLS handshake and the expected response from the server.

Warning:

Note: this process of making the certificates available system-wide, works only on Ubuntu Desktop and Server. For Ubuntu Core a system wide solution is not available yet.

Install the agents

The *main tutorial* (page 85) shows how to register a device with COS for robotics without TLS, using the basic[378] configuration snap. This configuration provides a basic setup to quickly start monitoring and collecting data from a device.

In this guide, we use the advanced[379] configuration snap instead. This extended setup enables TLS and configures all agents to use the certificates installed on the device.

The advanced[380] branch of the demo configuration snap[381], sets the `generate-device-tls-certificate` flag in the device configuration YAML. This flag triggers the generation of a private key and a certificate signing request, which is sent to the registration server upon registration[382]. The server generates a leaf certificate to be stored in the device's `rob-cos-data-sharing` snap.

The Foxglove bridge running on the device uses WebSockets to exchange data with Foxglove Studio served by the browser. To establish a secure WebSocket connection (**wss://**), the Foxglove bridge configuration[383] then uses this certificate by referencing the relevant paths.

Before proceeding, let's make sure that the Foxglove bridge snap can read the certificates from the `rob-cos-data-sharing` snap. Let's connect the bridge to it by executing the following command:

```
sudo snap connect foxglove-bridge:rob-cos-common-read rob-cos-data-sharing:rob-cos-common-read
```

[378] https://github.com/canonical/rob-cos-demo-configuration
[379] https://github.com/canonical/rob-cos-demo-configuration/tree/advanced
[380] https://github.com/canonical/rob-cos-demo-configuration/tree/advanced
[381] https://snapcraft.io/rob-cos-demo-configuration
[382] https://github.com/canonical/cos-registration-agent?tab=readme-ov-file#setup
[383] https://github.com/canonical/rob-cos-demo-configuration/blob/advanced/snap/local/configuration/foxglove-bridge.yaml

Now, disconnect the device by removing the cos-registration-agent snap:

```
sudo snap remove cos-registration-agent
```

Next, refresh the configuration snap to switch to the advanced configuration channel:

```
sudo snap refresh rob-cos-demo-configuration --channel=advanced/beta
```

Reset the configuration to ensure the advanced settings are applied:

```
sudo rob-cos-demo-configuration.reset-config
```

Configure the base URL for the model, making sure to use the https schema:

```
snap set rob-cos-demo-configuration rob-cos-base-url=https://<rob-cos-server-ip>/cos-robotics-model
```

Finally, reinstall the cos-registration-agent snap to register the device with TLS:

```
sudo snap install cos-registration-agent --edge
```

Warning:

Note: The generation of the leaf certificate for the device is asynchronous. As soon as the registration is started a private key and a CSR are generated, the CSR is sent to the registration server. Then the agent will keep polling for a signed certificate to be available on the database. This might take up to 5 minutes.

Your device should now be successfully registered with COS for robotics, with TLS enabled.

Laptop Side

Now that our server and device are setup with the correct certificates, we need to make sure our laptop and browser trust them. The only certificate that has to be trusted is the CA certificate issued by the self-signed-certificates charm, since the device certificate are leaf certificates.

First, let's create a `traefik-ca.crt` file on our laptop. The content is the same of the CA we installed on the device earlier, obtained by running the following command on the server:

```
juju run self-signed-certificates/0 get-ca-certificate
```

Warning:

Note: When copying the certificate from the output above, Make sure you *do not* include any leading spaces before the certificate lines. The file should start exactly with `-----BEGIN CERTIFICATE-----` at the beginning of the line. If the certificate lines are indented, the certificate will be invalid.

Finally, import the file in Google Chrome as follows:

- Open a new tab and navigate to `chrome://certificate-manager/localcerts/usercerts`.

- Click on **Import** and select the certificate files from your laptop.
- Once imported, the certificate should appear under the **Installed by You** section.

After these steps, your browser will trust the certificate, allowing for full TLS.

Security considerations

Before and after deploying COS for robotics with TLS, review the *Security considerations for COS for robotics* (page 256) page for important guidance on CA distribution, certificate renewal, device key protection, and known limitations.

Migrate a `cos-registration-server` database from sqlite3 to postgresql

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

The first version of `cos-registration-server` used SQLite3[384] as its database backend.

Starting with version `1/stable`, `cos-registration-server` now uses PostgreSQL[385] as its database backend to enhance the performance, reliability, and scalability.

This guide explains how to migrate an existing `cos-registration-server` database from SQLite3 to PostgreSQL.

At the end of this process, the `cos-registration-server` instance will be using PostgreSQL as its database backend, with all data migrated from the previous SQLite3 database.

Am I concerned?

In order to determine if you need to follow this migration guide, check if your existing `cos-registration-server` instance is using SQLite3 as its database backend.

```
juju ssh --container cos-registration-server cos-registration-server/0 \
ls /server_data/db.sqlite3 \
&& echo "You should migrate" \
|| echo "No need to migrate"
```

This command checks if the SQLite3 database file exists, and lets you know whether you need to migrate or not.

[384] https://sqlite.org/
[385] https://www.postgresql.org/

Prerequisites

- A running instance of `cos-registration-server-k8s` using SQLite3.
- A running instance of `postgresql-k8s` or a Juju deployment where you can deploy it.

Export the SQLite3 database

First, we retrieve all the data from the SQLite3 database file from the existing `cos-registration-server` instance.

To do so, we run `django` commands inside the container running our `cos-registration-server`:

```
juju ssh --container cos-registration-server cos-registration-server/0 \
DATABASE_BASE_DIR_DJANGO=/server_data/ \
/usr/bin/python3 \
/usr/lib/python3.10/site-packages/cos_registration_server/manage.py \
dumpdata   \
--natural-foreign \
--natural-primary \
--exclude contenttypes \
--exclude sessions \
--exclude admin.logentry
--indent 2 \
--output /tmp/data_export.json
```

Note:

Everything happening on this `cos-registration-server` instance after the dumpdata won't be included in the exported data, and will be lost. Make sure to stop any activity on the `cos-registration-server` while performing this migration.

We've now exported all the data from the SQLite3 database file. This file `data_export.json` contains all the data we need to migrate to PostgreSQL.

We now retrieve this file from the `cos-registration-server` container to our local machine:

```
juju scp --container cos-registration-server \
cos-registration-server/0:/tmp/data_export.json data_export.json
```

We now have all the data from our previous deployment.

To make sure we won't reuse this old SQLite3 database by mistake, we can move the SQLite3 database file from the `cos-registration-server` instance:

```
juju ssh --container cos-registration-server cos-registration-server/0 \
mv /server_data/db.sqlite3 /server_data/db.sqlite3.migrated
```

Set up PostgreSQL

Tip:

You can skip this step if you already have a Juju/Charm PostgreSQL instance running.

If no PostgreSQL instance is already available in the Juju model, we can deploy the `postgresql-k8s` charm:

```
juju deploy postgresql-k8s postgresql --channel 14/stable --trust
```

Import the data into the new PostgreSQL deployment

We refresh the instance of `cos-registration-server-k8s`, pulling the latest release that includes PostgreSQL support:

```
juju refresh cos-registration-server \
--channel 1/stable
```

Then, connect the `cos-registration-server` instance to the `postgresql` instance:

```
juju integrate postgresql cos-registration-server
```

Import the database file inside the new container

We now copy the `data_export.json` file available locally to the `cos-registration-server` instance:

```
juju scp --container cos-registration-server data_export.json cos-registration-server/0:/
tmp/data_export.json
```

Warning:

Note that we are importing the database file inside the `cos-registration-server` instance, and not the postgresql instance itself.
This is because we will use Django to load the data into PostgreSQL.

Load the data into PostgreSQL

First, we must retrieve the `DATABASE_URL` value used by the `cos-registration-server` instance:

```
juju ssh cos-registration-server/0 \
PEBBLE_SOCKET=/charm/containers/cos-registration-server/pebble.socket \
/charm/bin/pebble plan \
| grep DATABASE_URL
```

This will output a line similar to:

```
    "DATABASE_URL": "postgres://username:password@hostname:port/databasename",
```

Copy the value of `DATABASE_URL` (the part between the quotes) and replace `XXXXXXXXXX` in the command below. Now we can use this value while loading the data with Django.

```
juju ssh --container cos-registration-server \
cos-registration-server/0 \
DATABASE_URL=XXXXXXXXXX SECRET_KEY_DJANGO=\$\(cat /server_data/secret_key\) \
/usr/bin/python3 \
/usr/lib/python3.10/site-packages/cos_registration_server/manage.py \
loaddata /tmp/data_export.json
```

All the data from the previous SQLite3 database is now imported into PostgreSQL.

Make sure to run proper tests with the new `cos-registration-server` to ensure everything is working as expected.

2.3. Maintenance

This section includes all guides related to a robot's long term maintenance.

2.3.1. Maintenance

This section includes all guides related to a robot's long term maintenance.

Enable ROS ESM

ROS ESM builds on two Canonical ESM products: `ESM-infra` and `ESM-apps`. As a result, there are three steps to enabling ROS ESM:

1. Obtaining your Ubuntu Pro token
2. Enabling ESM-infra and ESM-apps
3. Enabling ROS ESM.

Note that `ESM-infra` and `ESM-apps` are required dependencies of ROS ESM.

Obtain your authentication token

Access to ESM is controlled by a token associated with your Ubuntu Single Sign-on (`SSO`) account. To obtain a token go to this page https://ubuntu.com/pro/subscribe. You can register for free up to 5 tokens to try out Ubuntu Pro and ROS ESM. If you already purchased ROS ESM or Ubuntu Pro, then you will have the token and you can review it at:

https://ubuntu.com/pro

Get in touch with us[386] if you need a personalised offer.

[386] https://ubuntu.com/robotics/ros-esm#get-in-touch

Enable ESM-infra and ESM-apps

In order to enable these services, you will need:

- An Ubuntu LTS machine with a version similar to or above 16.04 LTS
- Sudo access
- An email address, or an existing Ubuntu One account
- Ubuntu Pro client version `27.11.2` or newer

Once you have your Ubuntu Pro token, make sure your Pro client is up to date:

Ubuntu 20.04 and later

Ubuntu 18.04 and below

```
sudo apt update && sudo apt upgrade
sudo apt install -y ubuntu-pro-client
```

This is because the `ubuntu-advantage-tools` package has been deprecated in favour of the `ubuntu-pro-client` package.

> See More: For more information, please visit Ubuntu Pro Client Guide[387].
>
> [387] https://ubuntu.com/pro/tutorial

```
sudo apt update && sudo apt upgrade
sudo apt install -y ubuntu-advantage-tools
```

Confirm your Ubuntu Pro client version

Regardless of your Ubuntu distribution, make sure you are running the latest version of the Ubuntu Pro client. To check it, run:

```
pro version
```

You should have a version greater than or equal to `27.11.2`.

Attach the Pro client

Use the client to attach this machine to your contract using your token:

```
sudo pro attach YOUR_TOKEN
```

In order to see which Ubuntu Pro services are enabled you can run:

```
$ pro status
SERVICE          ENTITLED  STATUS       DESCRIPTION
anbox-cloud      yes       disabled     Scalable Android in the cloud
esm-apps         yes       enabled      Expanded Security Maintenance for Applications
esm-infra        yes       enabled      Expanded Security Maintenance for Infrastructure
```

(continues on next page)

(continued from previous page)

```
fips             yes       disabled     NIST-certified FIPS crypto packages
fips-updates     yes       disabled     FIPS compliant crypto packages with stable
security updates
livepatch        yes       enabled      Canonical Livepatch service
ros              yes       disabled     Security Updates for the Robot Operating System
usg              yes       disabled     Security compliance and audit tools
```

You should see some of the Ubuntu Pro services, such as Expanded Security Maintenance for Infrastructure (`esm-infra`) automatically enabled, while others will remain disabled until you switch them on.

If it's not, enter the following:

```
sudo pro enable esm-infra
```

Then, enable ESM Apps with:

```
sudo pro enable esm-apps
```

At any time, you can check how many deb packages are installed on your machine and from which source using:

```
pro security-status
```

Congratulations, you now have ESM-infra and ESM-apps enabled! Run an upgrade to install available security updates, if any:

```
sudo apt update
sudo apt upgrade
```

Enable ROS ESM

ROS ESM is exposed in the Pro client similar to `ESM-infra` and `ESM-apps` and is controlled by that same token. However, ROS ESM is disabled by default and not listed in the common service list. First, let's make sure that the Pro client is up-to-date:

```
pro version
```

Should return version `27.11.2` or greater.

Then, let's make sure that your token is entitled to enabling ROS ESM with:

```
pro status --all
```

You should now see the following ROS ESM services: `ros` and `ros-updates`. Make sure that the `entitled` column displays a `yes` in front of these services. If not, please reach out to customer service.

Now you have everything needed to enable ROS ESM. There are two suites available:

- **ros**: only security-related updates for ROS-related software.
- **ros-updates**: security and non-security-related updates for ROS-related software. These are security updates and bug fixes.

To enable the ROS security updates, run the following command:

```
sudo pro enable ros
```

To enable non-security updates, run the following command:

```
sudo pro enable ros-updates
```

Note that if you enter directly:

```
sudo pro enable ros-updates
```

You will be prompted to enable the `ros` service first, as `ros-updates` depends on `ros`.

Rosdep set up

ROS ESM provides its own distribution and `rosdep` files. Let's make sure you install `rosdep` from ESM and re-initialise it as follows:

Noetic/Foxy (Python3)

Kinetic/Melodic (Python2)

```
sudo apt install python3-rosdep
sudo rm /etc/ros/rosdep/sources.list.d/20-default.list
sudo rosdep init
rosdep update
```

```
sudo apt install python-rosdep
sudo rm /etc/ros/rosdep/sources.list.d/20-default.list
sudo rosdep init
rosdep update
```

Set up a ROS ESM environment

This guide will walk you through setting up your environment once you've *enabled ROS ESM* (page 167).

You have a couple of different choices: you can either install the complete ROS distro variant offered by ROS ESM (`ros_base`), or you can use rosdep to install the specific dependencies required by your ROS project. Let's quickly explore both options.

Installing ROS ESM base variant

ROS ESM offers the upstream `metapackage` variant called `ros_base`, which facilitates the installation of all ROS packages included in this variant. For example, if you are working with `Xenial` and its corresponding version ROS Kinetic, run the command:

ROS Foxy

ROS Noetic

ROS Melodic

ROS Kinetic

```
sudo apt install ros-foxy-ros-base
```

```
sudo apt install ros-noetic-ros-base
```

```
sudo apt install ros-melodic-ros-base
```

```
sudo apt install ros-kinetic-ros-base
```

Note:

Remember that the Ubuntu version and ROS version are co-dependent, so you have to choose a pair. For example, Ubuntu 16.04 LTS and ROS Kinetic, Ubuntu 18.04 LTS and ROS Melodic, Ubuntu 20.04 LTS and ROS Noetic/ROS 2 Foxy. Here you can find more information for ROS distributions[388] and ROS 2 distributions[389].

[388] http://wiki.ros.org/Distributions
[389] https://docs.ros.org/en/foxy/Releases.html

Note on rosdep set up

Note that ROS ESM is its own ROS distribution, and thus provides its own distribution and `rosdep` files. If you already have upstream ROS installed and initialised (e.g. you previously ran `sudo rosdep init`), you'll need to make sure you install `rosdep` from ESM and re-initialise it as follows:

```
sudo apt install python-rosdep
sudo rm /etc/ros/rosdep/sources.list.d/20-default.list
sudo rosdep init
rosdep update
```

Installing ROS ESM project-specific dependencies

Typically, when using ROS ESM, your ROS workspace would already be configured with the relevant source code. In such cases, it is highly recommended to accurately define the dependencies of your packages in the *package.xml* file and proceed by installing all the required ROS ESM dependencies by executing the following command:

```
cd ros-ws
rosdep install --ignore-src --from-paths src
```

By doing so, the packages required for your project will be fetched and installed from the `ROS ESM ppa`, ensuring smooth operation.

ESM and non-ESM components

A given ROS distribution includes a huge number of packages with wildly varying levels of quality. ROS ESM does not attempt to support them all, and instead focuses on core functionality.

We of course realise that everyone's needs are different, and are very open to receiving feedback[390] about anything that should be added to ROS ESM. While such additions will need to pass some scrutiny, we fully expect the number of ROS packages included in ESM to grow over time.

If you want to learn more about how to combine ROS ESM and upstream ROS components, check out *this guide* (page 172).

To see which packages are currently being supported for each distro, see *the current list of ROS packages included in ROS ESM* (page 197).

Combine ESM and upstream ROS components

We don't support enabling both ROS ESM as well as the upstream ROS Debian repository. This means that every ROS component you use must either be from ESM, or built from source against ESM.

There is tooling that makes this fairly straightforward, called `rosinstall_generator`, that will generate a rosinstall file containing the desired package(s) and all dependencies not already satisfied.

In a sourced ROS ESM environment, execute the following:

Noetic/Melodic

Foxy

Kinetic

```
sudo apt install python-rosinstall-generator
export ROSDISTRO_INDEX_URL="https://raw.githubusercontent.com/ros/rosdistro/master/index-v4.yaml"
rosinstall_generator <package>  --rosdistro <ros-distro>  --deps-up-to RPP > ~/extra-stuff.rosinstall
```

For example:

```
rosinstall_generator desktop_full --rosdistro noetic --deps-up-to RPP > ~/extra-stuff.rosinstall
```

Once that file is obtained, there are a few steps left to have the software usable.

First, if there isn't a workspace already, this needs to be created:

```
mkdir -p ~/ros_ws/src
```

If not already installed, install `vcs-tool` with the following command:

[390] https://ubuntu.com/robotics/ros-esm#get-in-touch

```
curl -s https://packagecloud.io/install/repositories/dirk-thomas/vcstool/script.deb.sh |
sudo bash
sudo apt-get update
sudo apt-get install python3-vcstool
```

Then the repos in the rosinstall file need to be fetched into the workspace with the following command:

```
cd ~/ros_ws
vcs import --shallow < ~/extra-stuff.rosinstall
```

Now dependencies of the workspace need to be installed:

```
cd ~/ros_ws
rosdep install --ignore-src --from-paths src --default-yes
```

Finally, the workspace needs to be built:

```
cd ~/ros_ws
catkin_make_isolated
```

```
sudo apt install python3-rosinstall-generator
export ROSDISTRO_INDEX_URL="https://raw.githubusercontent.com/ros/rosdistro/master/index-
v4.yaml"
rosinstall_generator <package>  --rosdistro <ros-distro>  --deps-up-to RPP > ~/extra-
stuff.rosinstall
```

For example:

```
rosinstall_generator desktop_full --rosdistro foxy --deps-up-to RPP > ~/extra-stuff.
rosinstall
```

Once that file is obtained, there are a few steps left to have the software usable.

First, if there isn't a workspace already, this needs to be created:

```
mkdir -p ~/ros_ws/src
```

If not already installed, install `vcs-tool` with the following command:

```
curl -s https://packagecloud.io/install/repositories/dirk-thomas/vcstool/script.deb.sh |
sudo bash
sudo apt-get update
sudo apt-get install python3-vcstool
```

Then the repos in the rosinstall file need to be fetched into the workspace with the following command:

```
cd ~/ros_ws
vcs import --shallow < ~/extra-stuff.rosinstall
```

Now dependencies of the workspace need to be installed:

```
cd ~/ros_ws
rosdep install --ignore-src --from-paths src --default-yes
```

Finally, the workspace needs to be built:

```
cd ~/ros_ws
colcon build --cmake-args -DCMAKE_BUILD_TYPE=Release
```

```
sudo apt install python-rosinstall-generator
export ROSDISTRO_INDEX_URL="https://raw.githubusercontent.com/ros/rosdistro/master/index-v4.yaml"
rosinstall_generator <package>  --rosdistro <ros-distro>  --deps-up-to RPP > ~/extra-stuff.rosinstall
```

For example:

```
rosinstall_generator desktop_full --rosdistro kinetic --deps-up-to RPP > ~/extra-stuff.rosinstall
```

Once that file is obtained, there are a few steps left to have the software usable.

First, if there isn't a workspace already, this needs to be created:

```
mkdir -p ~/ros_ws/src
```

If not already installed, install `wstool` with the following command:

```
sudo apt-get install python-wstool
```

Then the repos in the rosinstall file need to be fetched into the workspace with the following command:

```
cd ~/ros_ws
wstool init src ~/extra-stuff.rosinstall
```

Now dependencies of the workspace need to be installed:

```
cd ~/ros_ws
rosdep install --ignore-src --from-paths src --default-yes
```

Finally, the workspace needs to be built:

```
cd ~/ros_ws
catkin_make_isolated
```

That builds the required software against the ESM ROS release, where ABI will not break. Once the process is complete, the required software is available in the workspace.

Important:

Since ROS Groovy, not all packages belonging to the `desktop_full` `metapackage` have been `catkinized`. As a result, when using `rosinstall_generator`, it is necessary to compile the workspace using `catkin_make_isolated`.

Check if a CVE is fixed in your environment

If you're running ROS in production, it's important to know whether a specific CVE has been patched in your environment.

You can find detailed step-by-step instructions to check if your system is affected by a CVE[391], and to resolve a specific CVE[392] in the Ubuntu Pro Client documentation.

If you still **need to enable Ubuntu Pro and ROS ESM**, check out our *step-by-step guide* (page 167).

1. Get more details on the CVE

Go to the Ubuntu CVE Tracker website[393] and search for the CVE ID, for example: `CVE-2025-3753`. You'll find details about the vulnerability, including:

- Affected packages
- Impacted Ubuntu releases
- Fix status (e.g., Released, Needed, Not affected)
- Links to the associated public CVE entries in the NVD database

2. Find the fixed version

Look for the version number where the fix was released. Make a note of the package name and the patched version for your ROS ESM release. For example, you will find CVE-2025-3753[394], affecting the `ros-comm` package has been fixed for ROS ESM Noetic from version `1.17.4+2`:

ros-kinetic-ros-comm	16.04 LTS xenial	Fixed 1.12.17+9
ros-melodic-ros-comm	18.04 LTS bionic	Fixed 1.14.13+5
ros-noetic-ros-comm	20.04 LTS focal	Fixed 1.17.4+2

[391] https://documentation.ubuntu.com/pro-client/en/latest/howtoguides/fix_how_to_know_if_system_affected_by_cve/
[392] https://documentation.ubuntu.com/pro-client/en/latest/howtoguides/fix_how_to_resolve_given_cve/
[393] https://ubuntu.com/security/cves
[394] https://ubuntu.com/security/CVE-2025-3753

3. Check fix status in your system

If you're using **Ubuntu Pro with ROS ESM**, first make sure security updates are enabled:

```
pro status
```

You can use the Ubuntu Pro Client tool to check if your system is affected by running:

```
pro fix --dry-run CVE-2025-3753
```

The output of the dry run will also indicate whether if a fix is available, without actually applying it.

4. Update if needed

Finally, use the `pro fix` command to apply the needed fix to your system:

```
pro fix CVE-2020-25686
```

This command will:

- describe the CVE/USN;
- display the affected packages;
- fix the affected packages; and
- show if the CVE/USN is fully fixed in the machine.

This quick check helps you confirm whether potentially critical vulnerabilities have been addressed in your ROS-based systems.

2.4. Security

This page provides a **guide to strengthening a robot's security** by focusing on its underlying operating system (Ubuntu 22.04 LTS or Ubuntu Core 22), and emphasizing a Defense in Depth (DiD) approach.

It outlines six core steps for enhanced security, including securing connections (firewall, SSH, disabling Bluetooth), limiting network and physical access, customizing user permissions, keeping the system up-to-date with patches, and hardening kernel configurations.

- *Hardening your robot* (page 176)

2.4.1. Hardening your robot

Modern robots are typically designed to be open, robust, and easy to operate and repair. However, many of these systems are not adequately secured against threats – particularly given that robots are often accessible via the internet for remote operation, creating a uniquely large attack surface.

There is no silver bullet when it comes to robotics security. Instead, the best approach is `defense in depth (DiD)`, combining multiple layers of protection.

This section will address an **essential security layer**, whose key role is easily overlooked: **your robot's underlying operating system (OS)**. We'll discuss the easy steps you can take

to secure your robot by building on top of Ubuntu, and how Ubuntu Core provides you with enterprise-grade security for your robot out of the box.

How to secure your robot's base Ubuntu OS

For this step-by-step hardening exercise, we will secure your robot for deployment to a production environment. The focus will be on securing the underlying operating system beneath your ROS or ROS 2 application running on top of Ubuntu 22.04 LTS or Ubuntu Core 22. Therefore, we assume that you have already developed your application and are ready to deploy.

Most of the suggestions in this white paper are agnostic to CPU architecture. If there are nuances related to a particular architecture, those are named explicitly in the material that follows.

There are 6 core steps you can take to significantly enhance the security of your robot – and if you are using Ubuntu Core, many of these measures are implemented by default. The recommendations below are grouped into logical categories based on generally recognised good security practices, as they apply to your robot's OS. These are:

1. *Secure connections to your robot* (page 177)
2. *Limit network access* (page 180)
3. *Limit physical connectivity* (page 181)
4. *Customise user access* (page 184)
5. *Keep up to date with security patches* (page 186)
6. *Harden your kernel configurations* (page 187)

1. Secure connections to your robot

Configure firewall

The ideal robotics network would be an isolated Virtual Local Area Network (VLAN) with Access Control Lists (ACL) limiting inbound and outbound traffic. With the truly distributed nature of robots, this is not often the case. Robots must coexist with other WiFi guests. In this case, it is necessary to apply a rule set, allowing in only SSH.

If you're using Core, first install the `ufw` snap

```
sudo snap install ufw
```

Then run:

```
$ sudo ufw limit OpenSSH; sudo ufw enable
Rules updated
Rules updated (v6)
```

Note that `ufw allow ssh` would also work, but using `ufw limit`, we get an extra benefit. This way, the firewall will stall brute-force password attacks because it will start throttling new connections if it receives too many.

SSH hardening

SSH is the de facto method for connecting to a Linux server. SSH, as most know, allows for encrypted communication between client and server. However, since it does allow remote access, it also is a target for attackers. Attackers will try to perform brute force password attacks aimed at SSH connections. This is prevalent in attacks like those of the `Mirai Botnet`[395] that tried a static list of 60 usernames and passwords to compromise hosts.

To prevent this type of attack, you can configure two types of mitigation: requiring SSH keys, and a tool for detecting and blocking these attacks.

First, you need to generate an SSH key on your workstation and install the public SSH key on the robot. On the workstation, run the `ssh-keygen` command:

```
ssh-keygen -t rsa -b 4096
```

Make sure to use a key passphrase when prompted. This will encrypt the key on disk with that passphrase, which means if someone stole your private key, they would still need to know your passphrase to use it.

Next, you need to distribute the new SSH key to your robot, and you do that via a command called `ssh-copy-id`

```
ssh-copy-id -i ~/.ssh/id_rsa.pub yournewuser@robot
```

Now that you have a key on the robot, you can proceed with configuring SSH daemon options in `/etc/ssh/sshd_config`. The daemon allows for extensive configuration. To get a better understanding of the available options, run `man sshd_config`. To secure your SSH sessions, set the following options:

```
PermitRootLogin no
X11Forwarding no
PasswordAuthentication no
```

If you have users with no shell access, you can additionally disallow Transmission Control Protocol (TCP) and agent forwarding. Keep in mind that users with shell access can install their own forwarders.

```
AllowTcpForwarding no
AllowAgentForwarding no
```

Restart `sshd`:

```
sudo service ssh restart
```

If you have made a mistake you may no longer be able to SSH into the robot. In the event this happens, you can attach a keyboard and monitor to the Raspberry Pi and revert the changes made in `sshd`.

Next, you will want to install `sshguard` to prevent users from performing SSH brute force attacks. Why is this necessary after requiring keys? `A Denial of Service (DoS)` can be achieved by continual login attempts from a password attack. A tool like `sshguard` will block the requests to login at the firewall after several failed logins over a short period.

[395] https://www.cisecurity.org/insights/blog/the-mirai-botnet-threats-and-mitigations

```
sudo apt install sshguard
```

Secured remote connections are the default in Ubuntu Core

By default, Ubuntu Core runs an OpenSSH server which is configured for security to accept `PubkeyAuthentication` only. See how this setup works[396] and integrates with your UbuntuOne account.

Disable Bluetooth

Bluetooth is a convenient way to connect devices wirelessly, but it is not without vulnerabilities and, naturally, exploits. For example, `BlueBorne`[397] doesn't even need to pair with a Bluetooth device or even need the device to be discoverable. There are also known exploits for the zero-click kernel-level vulnerability `BleedingTooth`[398], which allows unauthenticated attackers to execute arbitrary code with kernel privileges on vulnerable devices. If your robot is not using Bluetooth for any function, you should disable it.

Disable Bluetooth on Ubuntu Desktop

You can manage services such as Bluetooth via the systemd daemon. Use the `systemctl` utility to disable Bluetooth and configure it to not start when system boots:

```
systemctl disable bluetooth.service
```

No Bluetooth connectivity by default on Ubuntu Core

If you are using Ubuntu Core, Bluetooth will be disabled by default. In order to enable Bluetooth on Core[399], you will need the `BlueZ` protocol stack snap installed, the Bluetooth daemons running, and the corresponding plugs and slots connected (thanks to Snaps' interfaces mechanism[400]). The lower-level part of it comes with the kernel snap, but the user-space portion has to be installed as a separate snap. This is one less open service to worry about.

[396] https://ubuntu.com/core/docs/connect-with-ssh
[397] https://en.wikipedia.org/wiki/BlueBorne_(security_vulnerability)
[398] https://portswigger.net/daily-swig/bleedingtooth-google-drops-full-details-of-zero-click-linux-bluetooth-bug-chain-leading
[399] https://documentation.ubuntu.com/core/explanation/system-snaps/bluetooth/#bluez
[400] https://snapcraft.io/docs/interfaces

2. Limit network access

Disable WiFi

If your production robot is stationary and uses an ethernet connection instead of wireless, you should disable the wireless chip on all versions of Ubuntu. You can do that by using the `rfkill` command line tool.

```
rfkill block wlan
```

Disable ethernet

If your production robot uses a WiFi connection instead of wired, then disable the wired connection. You can use the Network Manager utility to detect and disconnect any ethernet devices from your system. Install the Network Manager snap, then check which devices are connected and disconnect each one explicitly:

```
sudo snap install network-manager
nmcli dev
nmcli dev disconnect <dev_name>
```

If you want to permanently disallow the use of ethernet, you could also blocklist the ethernet driver of your device. For example, for a driver named `lan78xx`, you need to add the following lines in `/etc/modprobe.d/blacklist.conf` and reboot.

```
blacklist lan78xx
```

Disable IPv6

By default, all interfaces come up with an IPv6 address. If you are not using IPv6, you should disable it. Not because there is anything wrong with IPv6, but because you want to reduce the number of pathways through which your robot could be attacked.

```
$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
1000
     link/ether bc:a8:a6:fd:43:be brd ff:ff:ff:ff:ff:ff
     inet 192.168.128.46/24 brd 192.168.128.255 scope global dynamic noprefixroute eth0
        valid_lft 48058sec preferred_lft 48058sec
     inet6 fe80::ae66:f8fd:c277:efba/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

To disable IPv6 add the following lines into `/etc/sysctl.conf` in Desktop, or `/etc/sysctl.d/10-ipv6-privacy.conf` in Core:

```
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
```

And then run:

```
$ sudo sysctl -p
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
```

Verify that IPv6 is gone:

```
$ ip addr show eth0
2: eth0: &lt;BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default
qlen 1000
     link/ether bc:a8:a6:fd:43:be brd ff:ff:ff:ff:ff:ff
     inet 192.168.128.46/24 brd 192.168.128.255 scope global dynamic noprefixroute eth0
        valid_lft 48058sec preferred_lft 48058sec
```

Network access is optional by design in Ubuntu Core

In Ubuntu Core, where every application is a snap, you have an extra security layer in terms of network access. By design, you have control over whether each snap will get network access as defined in their interfaces[401] or not. Learn more about snaps' confinement types[402] in the Snapcraft documentation.

3. Limit physical connectivity

Disable USB

Universal Serial Bus (USB) has been around for a long time as a convenient and quick way to expand system peripherals and transfer data among disconnected systems. This easy expansion, unfortunately, made way for USB devices that can be used for malicious intent. Examples of USB attack platforms are `PoisonTap`[403], `Responder`[404] or `P4wnP1`[405], which can run on a Raspberry Pi Zero, Hack5 Turtle, or `USB Armory`. Some dangerous USB-based attacks have included `BadUSB`[406] and `RubberDucky`[407], which when plugged in will have the computer recognise the device as a keyboard, leaving an open door for executing commands remotely.

Disable USB on Ubuntu Desktop

If you are using Ubuntu Desktop, one way to prevent USB abuses on your robot is to disable various USB types of devices if you are not using them. To check if you can disable individual USB types of devices, run:

```
$ egrep -e “USB_NET_DRIVERS=” -e “USB_STORAGE=” -e “USB_HID=” -e “USB_SERIAL=” /boot/
config-`uname -r`
CONFIG_USB_NET_DRIVERS=y
CONFIG_USB_HID=y
```

(continues on next page)

[401] https://snapcraft.io/docs/interfaces
[402] https://snapcraft.io/docs/network-interfaces
[403] https://samy.pl/poisontap/
[404] https://room362.com/post/2016/snagging-creds-from-locked-machines/
[405] https://github.com/RoganDawes/P4wnP1
[406] https://www.vesiluoma.com/exploiting-with-badusb-meterpreter-digispark/
[407] https://www.theverge.com/23308394/usb-rubber-ducky-review-hack5-defcon-duckyscript

(continued from previous page)

```
CONFIG_USB_STORAGE=y
CONFIG_USB_SERIAL=m
```

Anything marked with a "=m" can be disabled, and anything marked with a "=y" can't because "y" means that the driver is compiled into the kernel. The example above is from a TurtleBot3[408] with a Raspberry Pi 3+, and unfortunately, those modules need to be compiled into the kernel. The only things we could disable are the serial devices, but the TurtleBot3 uses that module for the LiDAR communication.

We'll return to the Raspberry Pi based TurtleBot3, but if your robot is based on a different architecture, the kernel options could be different. For example, the options on the x86_64 architecture are below.

```
user@x86_64:~$ egrep -e "USB_NET_DRIVERS=" -e "USB_STORAGE=" -e "USB_ HID=" -e "USB_
SERIAL=" /boot/config-`uname -r`
CONFIG_USB_NET_DRIVERS=m
CONFIG_USB_HID=m
CONFIG_USB_STORAGE=m
CONFIG_USB_SERIAL=m
```

USB drivers are compiled as modules, therefore, we can disable individual types of devices. Starting with Human Interface Devices (HID), if you want to prevent someone from plugging in a keyboard or other HID devices, then block the module from loading.

```
user@x86_64:~$ sudo rmmod usbhid
user@x86_64:~$ echo "blacklist usbhid"| sudo tee -a /etc/modprobe.d/ blacklist.conf
```

To prevent someone from plugging in a storage device such as a USB disk or a flash drive, block off the modules from loading. Copy this script into your console and run it:

```
for i in usb-storage usb_storage; \
   do sudo rmmod $i ; echo "blacklist $i" | sudo tee -a /etc/modprobe.d/blacklist.conf; \
done
```

Disallow USB serial devices by blocklisting the USB serial driver:

```
user@x86_64:~$ sudo rmmod usbserial
user@x86_64:~$ echo "blacklist usbserial"| sudo tee -a /etc/modprobe.d/blacklist.conf
```

Disallow USB networking devices by blocklisting USB networking modules:

```
find /lib/modules/`uname -r`//drivers/net/usb -type f -name *.ko | xargs basename -s .ko |
sed s'/^/blacklist /' | sudo tee -a /etc/modprobe.d/blacklist.conf
```

Disallow all USB devices, effectively disabling USB functionality:

```
for i in $(find /lib/modules/`uname -r` -name usb -type d); \
   do find $i -name *.ko | sed 's/.ko$//g' | awk -F/ '{print "blacklist",$(NF-0)}'; \
done | sudo tee -a /etc/modprobe.d/blacklist.conf
```

While you can perform the above steps to disable the loading of all USB device modules, there are some limitations that you should take into consideration. If you recall the kernel configuration, some crucial modules are compiled into the kernel.

[408] https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/#overview

```
CONFIG_USB_NET_DRIVERS=y
CONFIG_USB_HID=y
CONFIG_USB_STORAGE=y
```

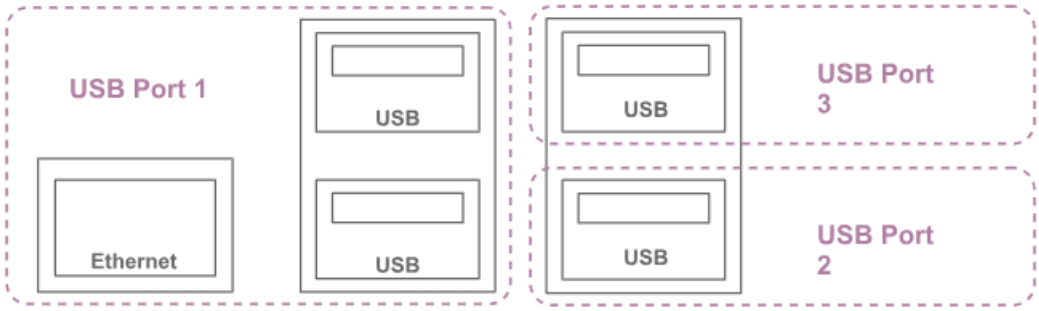
Therefore, we can't prevent someone from plugging in HID devices, and we can't preclude storage devices from being added. There is a bit more of a grey area for network devices. While the `USB_NET_DRIVERS` compiles the `usbnet` module into the kernel, we can't blocklist it. But `usbnet` is only a "base" driver, and a network device will require an additional chip-specific driver. And those you can blocklist, see above on the topic of "blocklisting USB networking modules". Keep in mind that the Raspberry Pi 3+ ethernet interface is connected to the internal USB bus. If you blocklist all USB network modules, your ethernet will also be disabled.

As we can't prevent people from plugging in storage and HID devices with blocklisting since modules are compiled in, what other options are there? You can turn off the USB port power on your robot. Get the `hub-ctrl`[409] utility. You will have to compile it from source. If you compile the software on the robot it is best practice to remove the compiler before deploying the robot. By leaving compilers on the robot, an attacker who gained system level access would have the ability to compile malicious software.

Now that you have the binary on your robot, to turn off the power on Hub 1 to USB Port 2 and 3, for example, run:

```
user@x86_64:~$ sudo ./hub-ctrl -h 1 -P 2 -p 0
user@x86_64:~$ sudo ./hub-ctrl -h 1 -P 3 -p 0
```

Try plugging in a keyboard or a flash drive; nothing should happen. Try plugging in a keyboard or a flash drive; nothing should happen. See the diagram below to determine the location of Port 3 on Raspberry Pi3 +.



Raspberry Pi3 USB ports

On to the remaining two USB plugs on the left. That's where you should connect the LiDAR and the `OpenCR` board. The ethernet, `OpenCR`, and LiDAR will all be on the USB Port 1, and you should physically prevent tampering with the two plugged cables. This is where another limitation of Raspberry Pi3 + comes into play. You cannot turn off the power with `hub-ctrl` to USB Port 1, because for that to work, no devices can actually be plugged into the port when you turn off the power. If a device is plugged in, the USB system will just re-initialise the device, so the port won't actually lose power. But it's not possible to unplug all devices on Port 1. The ethernet device is electrically wired internally to Port 1; thus, it prevents the power from being turned off to Port 1. And because port 1 has two USB plugs, those two plugs will always be powered. Therefore, they will require some physical anti-tampering method.

[409] https://github.com/codazoda/hub-ctrl.c

USB is less problematic on Ubuntu Core

If your robot is using Ubuntu Core, individual applications are sandboxed via a policy-based system that restricts access to the filesystem, network interfaces, serial devices, external hardware, system calls and other kernel features. This mechanism by default keeps in check the access that each system component can have. Learn more about how applications are confined in Ubuntu Core[410] and how Snap confinement works[411].

In addition, Core can defend against software corruption or running unauthorised applications via its integrated code authenticity validation, such that unauthorised or malicious code cannot be introduced.

4. Customise user access

Remove any default users

Part of gaining access to a system remotely is attacking default usernames and simple passwords. Default users are one way to make it easier for an attacker to access the system. Removing default usernames and switching to named user accounts also has the added benefit of accountability of user actions. When an account is shared it is difficult, if not impossible, to determine who exactly performed the actions that may have led to a disruption in service. It is also a good idea to install `libpam-passwdqc`, which will ensure that user passwords meet a minimum security requirement. Run the commands below to install `passwdqc`, create a new user, add the user to the sudo group (if required), logout as "ubuntu" user, login with the new user, and remove the default "ubuntu" user.

```
ubuntu@robot:~$ sudo apt install libpam-passwdqc
ubuntu@robot:~$ adduser yournewuser
ubuntu@robot:~$ sudo usermod -aG sudo yournewuser
ubuntu@robot:~$ su yournewuser
yournewuser@robot:~$ sudo deluser --remove-home ubuntu
```

No default users on Ubuntu Core

Because Ubuntu Core is designed as a user-less system, your device does not have a default username and password in the first place. This mechanism does away with the need to monitor for common defaults, making your life easier as you work to reduce your attack surface.

For development, Ubuntu Core's reference images provide the `console-conf system`[412] to allow automatic provisioning of a user account linked to an existing Ubuntu One account. The public SSH key of the Ubuntu One account is automatically copied to the device, allowing for secure remote connections to the device via SSH.

For production devices, Core has a secure system available (system-user assertions[413]) that can be used to securely trigger user account creation on devices in the field.

[410] https://ubuntu.com/core/docs/snaps-in-ubuntu-core
[411] https://snapcraft.io/docs/snap-confinement
[412] https://ubuntu.com/core/docs/system-user
[413] https://ubuntu.com/core/services/guide/create-a-system-user-assertion

User permissions

Access to any data stored in your robot by existing users should be carefully handled following the principle of least privilege. The default home directory permissions on Ubuntu allow users to share files in their home directories. To prevent users from accessing other users' files, you can make the following changes:

```
sudo chmod 0750 /home/*
sudo sed -i.orig -e 's/=0755/=0750/' /etc/adduser.conf
```

In addition, file creation and access race conditions are a way users could escalate their access beyond what they were granted. To help mitigate that, make sure to use `umask`. Users `umask` sets the file mode creation mask of processes; you can use it to restrict access to the content a given user generates. To prevent users from accessing each others files, run:

```
echo "umask 077" >> /etc/profile
```

Since not all shells interpret the `/etc/profile` file, you should also add the following line into `/etc/pam.d/login`:

```
session optional pam_umask.so umask=0077
```

While the options above set the `umask` for children of "bash" or PAM sessions, don't rely on your parent process `umask` in your ROS code – always set your process `umask` explicitly (type `man 2 umask` or `man pam_umask` in your console for more information on using `umask` in your code).

Permissions are restricted by default on Ubuntu Core

When using Ubuntu Core, you have the ability to control how each snap will interact with your home directory. All of the software in Core is delivered via strict snaps, and snap strict confinement leverages AppArmor to lock down the filesystem. Permissions for snaps and Core are handled via interfaces[414], and the home interface is not automatically connected when a snap is installed. Manual connections give you complete control over what kind of access you allow. Dedicated snap stores[415] also offer the ability for companies to declare snap connections for snaps hosted in their own private store.

Check which interfaces a given snap currently uses with:

```
snap connections <snap-name>
```

If needed, the 'snap connect' command will connect Ubuntu Core and the snap via the home interface, allowing it to save files to your home directory.

```
snap connect <snap-name>:home :home
```

You can even configure a user's home directory to be something other than /home with a simple system command line option[416].

[414] https://snapcraft.io/docs/interfaces
[415] https://ubuntu.com/core/docs/dedicated-snap-stores
[416] https://snapcraft.io/docs/system-options#heading--homedirs

Finally, to control access to specific files, snaps also offer the 'personal-files' interface. This interface provides read and/or write access to privileged files in a user's home directory. Once you've defined this interface for a directory with restricted access, you can enable it for a specific snap:

```
snap connect <snap-name>:restricted-dir
```

5. Keep up to date with security patches

Unattended upgrades

Part of overall security hygiene is to patch security vulnerabilities in a timely manner. As ROS is based on Ubuntu, you can keep up to date with security patches by enabling unattended upgrades. All you have to do is make sure that the options below are uncommented in `/etc/apt/apt.conf.d/20auto-upgrades`:

```
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

This will periodically refresh the package list and upgrade packages that have security patches available. Although auto-upgrade for most packages will require no reboot, there are going to be updates that require restarts. To see if any upgrades need a reboot, you can periodically check by running the command below:

```
ls -l /var/run | grep reboot-required
```

If a reboot is required, there will be a file such as `reboot-required.pkgs`. Run `cat /var/run/reboot-required.pkgs` to check which ones.

Easily enable kernel updates on Ubuntu Core

In Ubuntu Core, snaps update automatically, and the snapd daemon will by default check for updates 4 times a day. You can alternatively specify the time ranges for the updates to occur by setting a system-wide timer option:

```
snap set system refresh.timer=4:00-7:00,19:00-22:00
```

In addition, the kernel and base operating system are handled as snaps, and so will receive regular and consistent updates the same way as all installed applications. This also includes graceful error handling with automatic rollbacks on improperly updated kernels. This means an easy way to keep your entire system up to date.

It is also possible to build a device-agent which can control updates on a Core device, so you can have more control over the timing of updates.

6. Harden your kernel configurations

Set secure IPv4 configurations

`Sysctl` is an interface to modify kernel settings at runtime. Use `sysctl` to harden your device by applying secure configurations related to network and system settings.

To see the current values of the `sysctl` variables you can run:

```
sudo sysctl -a
```

The recommended way to change the variables is to edit `/etc/sysctl.conf` in Desktop, or `/etc/sysctl.d/10-network-security.conf` in Core, which will make the changes persist after reboot.

Modify the following variables in the corresponding `sysctl` file to log packets which have "impossible" addresses, spoofed and source-routed packets, and redirects which could be a sign of adversarial network activity:

```
net.ipv4.conf.all.log_martians = 1
net.ipv4.conf.all.accept_source_route = 0
net.ipv4.conf.default.accept_redirects = 0
net.ipv4.conf.default.secure_redirects = 0
```

Furthermore, these configurations can be used to protect your device against potential SYN flood attacks, a form of `Denial of Service (DoS)` attack where many SYN requests without completing the connection:

```
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_synack_retries = 5
```

Apply `address space layout randomisation` (ASLR)

Take advantage of address space layout randomisation (ASLR), a technique to prevent attackers from using knowledge of the memory address allocated to functions in a given vulnerable program to execute exploits. The ASLR configuration can have one of three values: 0 (disabled), 1 (conservative randomisation), or 2 (full randomisation). Because the effectiveness of the technique depends on the amount of randomised address space, it is recommended to apply full randomisation to reduce the likelihood of a successful attack. Full randomisation is already the default in Ubuntu[417]; you can ensure this configuration is permanent by adding the following lines to `/etc/sysctl.conf` in Desktop, or `/etc/sysctl.d/10-kernel-hardening.conf` in Core:

```
kernel.randomize_va_space = 2
```

Apply the changes with:

```
sudo sysctl -p
```

[417] https://wiki.ubuntu.com/Security/Features#Address_Space_Layout_Randomisation_.28ASLR.29

Disable core dump

Getting a core dump from an application when it experiences a crash is a great way to debug issues in applications. The size of the core dump can vary widely from application to application; it all depends on the memory footprint of the application. The assumption is that an application would seldom experience a problem, but when it does, it will provide memory contents for the developers to look through.

While that assumption is reasonable for development, once you move your robot to production you could face a `Denial of Service (DoS)` attack. When a process crashes, it's generally assumed that the process is restarted to restore the particular service. For example, the systemd will do that automatically for you. Suppose an attacker finds a way to trigger this application crash through your exported interface, or worse, if they find a way to script it. They could cause the particular service to crash and dump the core hundreds of times per second, which would fill up your storage. The entire robot then will be subject to all sorts of unpredictable consequences as all services start behaving in unexpected ways due to a lack of space. By disabling the core dump, the attacker can only affect the buggy service and not the entire robot system. To disable the core, add the following lines to `/etc/sysctl.conf` in Desktop, or `/etc/sysctl.d/10-kernel-hardening.conf` in Core:

```
kernel.core_uses_pid = 0
```

And run:

```
sudo sysctl -p
```

Remove unneeded kernel modules

The `modprobe` utility can be used to add or remove loadable kernel modules `(LKM)` to the kernel. These modules are a desirable location for an attacker to place a rootkit and gain kernel-level access. Rootkits are particularly dangerous, as they allow an attacker to compromise the core components of your OS while going unnoticed. Thus, it is a good practice to deny or disable unneeded modules from loading at boot time.

`Modprobe` can first help you check which `LKMs` are in your system, and disable any you don't need. You can use this command to list them (by convention, they use the `.ko` extension, which stands for `kernel object`):

```
find /lib/modules/`uname -r` -type f -name '*.ko'
```

Then, run `modinfo` on a module to investigate further, and see all its dependent modules. When you are sure you can disable a module, define it as `/bin/false` in its `.conf` file, so that the module will not be loaded. This is a good convention to visualise when something is not allowed.

```
echo "install <module> /bin/false" > /etc/modprobe.d/&lt;module>.conf
```

In addition, to increase your rootkit detection abilities, use the `rkhunter` (Rootkit Hunter) utility to scan for rootkits, backdoors, and possible local exploits in your robot. It achieves this by checking default files where rootkits are stored, comparing the SHA-1 hashes of critical files with known good ones, hidden files, and suspicious strings in kernel modules. It's advisable

to schedule a task to run this check automatically, including ensuring you have an updated database.

In `/etc/default/rkhunter`:

```
CRON_DAILY_RUN = “true”
CRON_DB_UPDATE = “true”
APT_AUTOGEN = “true”
```

Disallow dynamically loaded modules

Consider restricting dynamic loading of kernel modules more generally. Dynamically loaded modules, as much as they provide great flexibility, are especially dangerous, as they allow a rootkit to inject itself into the kernel without requiring a reboot. If you decide you can do without a dynamic kernel for your use case, just edit this parameter in `/etc/sysctl.conf` in Desktop, or `/etc/sysctl.d/10-kernel-hardening.conf` in Core:

```
kernel.modules_disabled = 0
```

Secure boot configurations will prevent kernels with unsigned modules, so there would be no way to dynamically load a rootkit. Ubuntu Core has secure boot enabled by default and thus mitigates these related attacks. It’s also possible to enable secure boot in Desktop[418].

You can flexibly configure the kernel on Ubuntu Core with gadget snaps

Canonical provides several reference kernel snaps[419] that you can also configure using a gadget snap with a number of different options[420]. Contact us[421] to get help in setting up your Ubuntu Core Image with the gadget snap.

Conclusion

As you prepare your robot for production, security shouldn’t be an afterthought. Putting an upfront effort can, in the long run, save you resources, and the headache that comes with a security breach. The more awareness you have of the security of your robot’s essential layers such as your operating system, the better prepared you will be from the start. And Ubuntu Core provides the level of control you want for a system that is seamlessly secure.

As our world becomes more and more connected, a paradigm shift from “if we become a target” to “when we become a target” warrants a proactive approach to security. And remember, security is not a single on/off switch, but many smaller actions that build strength in numbers. More breaches than we think are simply opportunistic. If you put up enough barriers, attackers are likely to move on to a weaker target – and with Ubuntu Core, those barriers are in place out of the box.

For more information about Ubuntu and robotics[422] please visit our website.

[418] https://wiki.ubuntu.com/UEFI/SecureBoot
[419] https://ubuntu.com/core/docs/kernel-building
[420] https://ubuntu.com/core/docs/modify-kernel-options
[421] https://ubuntu.com/core/features/secure-boot#get-in-touch
[422] https://ubuntu.com/robotics

You may also consider reading the following materials:

- Distributing ROS apps with snaps[423]
- Ubuntu Core documentation[424]
- Key considerations when choosing a robot's operating system[425]
- ROS Expanded Security Maintenance[426]

Need help getting to market? Contact us[427]

[423] https://ubuntu.com/robotics/docs/ros-deployment-with-snaps-part-1
[424] https://ubuntu.com/core/docs
[425] https://ubuntu.com/engage/robot-operating-system-choice
[426] https://ubuntu.com/robotics/ros-esm
[427] https://ubuntu.com/robotics#get-in-touch

3. Reference

Technical information such as specifications, architecture, API documentation, and troubleshooting tips.

3.1. Snapcraft

Snapcraft technical reference for snapping your ROS applications.

3.1.1. Snapcraft

Snapcraft technical reference for snapping your ROS applications.

Snapcraft Plugins

- **Colcon plugin**[428]
  The **colcon** plugin is useful when building ROS 2[429] parts.
- **Catkin plugin**[430]
  The **catkin** plugin is useful when building ROS[431] parts.
- **Catkin-tools plugin**[432]
  The **catkin_tools** build plugin is useful when building ROS[433] parts.

Snapcraft Extensions

- **ROS Noetic extension**[434]
  This extension helps you snap ROS[435] applications for the Noetic Ninjemys[436] distribution.
- **ROS 2 Foxy extension**[437]
  This extension helps you snap ROS 2[438] applications for the Foxy Fitzroy[439] distribution.
- **ROS 2 Humble extension**[440]
  This extension helps you snap ROS 2[441] applications for the Humble Hawksbill[442]

[428] https://snapcraft.io/docs/colcon-plugin
[429] https://docs.ros.org/
[430] https://snapcraft.io/docs/catkin-plugin
[431] https://www.ros.org/
[432] https://snapcraft.io/docs/catkin-tools-plugin
[433] https://www.ros.org/
[434] https://snapcraft.io/docs/ros-noetic
[435] https://www.ros.org/
[436] https://wiki.ros.org/noetic
[437] https://snapcraft.io/docs/ros2-foxy-extension
[438] https://docs.ros.org/
[439] https://docs.ros.org/en/foxy/Releases/Release-Foxy-Fitzroy.html
[440] https://snapcraft.io/docs/ros2-humble-extension
[441] https://docs.ros.org/
[442] https://docs.ros.org/en/foxy/Releases/Release-Humble-Hawksbill.html

distribution.

- **ROS 2 Jazzy extension**[443]
  This extension helps you snap ROS 2[444] applications for the Jazzy Jalisco[445] distribution.

FAQ & Troubleshooting

This page reference ROS and ROS 2 snap common questions and troubleshooting:

Frequently Asked Questions

If you cannot find an answer to your question here, feel free to ask it on the ubuntu forum[446].

I cannot snap my application. What should I check?

- Snapcraft uses the familiar ROS tools (`rosdep`/`catkin`/`colcon` etc). Which means that your application must follow the ROS directives for proper packaging, such as declaring all the necessary dependencies in the *`package.xml`* files or the install rules in your *`CMakeFile.txt`*.

Make sure that these are in good order before attempting to create a snap.

- Use colcon-in-container to validate your workspace before snapping
  If you're encountering persistent issues when snapping your ROS 2 application, especially related to missing dependencies or environment mismatches, we recommend trying **colcon-in-container**[447]. This tool builds and tests your ROS workspace inside an ephemeral, isolated container with a clean ROS environment.

Which base should I use (`core18`, `core20`, `core22` or `core24`)?

- You should use the base that corresponds to your ROS version. That is,
  - `core18` for ROS Melodic and ROS 2 Dashing.
  - `core20` for ROS Noetic and ROS 2 Foxy.
  - `core22` for ROS 2 Humble.
  - `core24` for ROS 2 Jazzy.

[443] https://snapcraft.io/docs/ros2-jazzy-extension
[444] https://docs.ros.org/
[445] https://docs.ros.org/en/jazzy/Releases/Release-Jazzy-Jalisco.html
[446] https://discourse.ubuntu.com/c/project/robotics/121
[447] https://github.com/canonical/colcon-in-container/blob/main/README.md

For ROS 1, do I have to expose `roscore` from my snap?

- Exposing a `roslaunch` command from your snap will automatically launch a `roscore` if needed. The only reason to expose explicitly the `roscore` would be if you plan to start the `roscore` explicitly from your snap.

Where should my *snapcraft.yaml* file live?

- Within the package:
  - In `core20` and above, the *snap/* directory should be located at the root of the package (next to your *package.xml* file)
  - In `core18` the snap/ directory should be located either one folder behind your package root or at the root of your workspace
- Outside the package:
  - Using a `rosinstall` file to download the sources.
  - Using a single git repository holding the sources.

Can my snap save data on the host?

- The snap defines some *environment variables for data and file storage* (page 229) pointing to different locations that a snap can write to depending on the use case of your data.
- You can save data that are common across revisions of a snap. These directories **won't be backed-up** and restored across revisions:
  - `$SNAP_COMMON`, typical value: `/var/snap/hello-world/common`. Owned by `root`
  - `$SNAP_USER_COMMON`, typical value: `/home/$USER/snap/hello-world/common`. Owned by `$USER`
- You can save data for a revision of a snap. This directory **is backed up** and restored across revisions:
  - `$SNAP_DATA`, typical value: `/var/snap/hello-world/27`. Owned by `root`
  - `$SNAP_USER_DATA`, typical value: `/home/$USER/snap/hello-world/27`. Owned by `$USER`
- Additionally, with the `home` interface[448], your snap could access the real `$HOME` of the user by accessing `$SNAP_REAL_HOME`.

[448] https://snapcraft.io/docs/home-interface

Troubleshooting

The command(s) `rosrun` and/or `roslaunch` are not available in my snap

- If this happens, it means that your ROS project does not define a runtime dependency on either `rosrun` nor `roslaunch` anywhere. You can fix this by declaring the dependency in the appropriate ROS *package.xml* file. Another option is to list either (or both) ROS packages as `stage-packages` in your *snapcraft.yaml*. The ROS packages for `rosrun` and `roslaunch` are respectively:
  - `ros-${ROS-DISTRO}-rosbash`
  - `ros-${ROS-DISTRO}-roslaunch`.

With `core18` Catkin plugin creates an external link that prevents the security checks to pass

- Please see: Catkin generating an external link[449].

Missing `lapack` and/or `blas`

- Paths to the libraries `lapack` and `blas` are not included in the library path by default. Thus, it must be extended manually in your app.

```
environment:
  "LD_LIBRARY_PATH": "$LD_LIBRARY_PATH:$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/blas:
$SNAP/usr/lib/$SNAPCRAFT_ARCH_TRIPLET/lapack"
```

Warning: *"This part is missing libraries that cannot be satisfied with any available stage-packages known to snapcraft"*

- Some libraries are build-time only dependencies, but are still reported as run-time dependencies by `snapcraft`. This warning is a false positive and will be fixed soon in snapcraft. For instance, when snapping `ros2-demo` you might encounter:

```
This part is missing libraries that cannot be satisfied with any available stage-
packages known to snapcraft:
# false-positive, none of the following are necessary at run-time
libnddsc.so
libnddscore.so
libnddscpp.so
librosidl_typesupport_connext_c.so
librosidl_typesupport_connext_cpp.so
librticonnextmsgcpp.so
```

[449] https://forum.snapcraft.io/t/store-unable-to-accept-contains-external-symlinks-to-sudo-service/23269

Strictly confined ROS 2 snaps shows an access error regarding shared memory

If you see something similar to:

```
[RTPS_TRANSPORT_SHM Error] Failed to create segment 86bb3c83d0835208: Permission denied ->
  Function compute_per_allocation_extra_size
[RTPS_MSG_OUT Error] Permission denied -> Function init
```

- ROS 2 communication library is trying to use the shared memory mechanism. Don't worry, even if you see this error, the messages are going to be transmitted (just not through shared memory). If you want to use the shared memory of ROS 2 within snap, visit: *ROS 2 shared memory in snap* (page 115)

At runtime, the snap shows an error similar to

```
[rospack] Unable to create temporary cache file /home/USER/.ros/.rospack_cache.VyyWPF:
Permission denied
```

- By default `rospack` and `roslog` write to the `$HOME/.ros`. When strictly confined a snap which doesn't have the `home` interface[450] cannot access the host `$HOME`. Also, even with the `home` plug[451] the snap cannot access to hidden directories (.directories) for security reasons (like .ssh).
  - To solve that, we can write ROS logs in the `$SNAP_USER_DATA` environment variable. We can do so by defining the ROS environment variable `ROS_HOME`. We can do so by adding to a snap app in the `snapcraft.yaml`:

    ```
    [...]
    apps:
      myapp:
        environment:
          ROS_HOME: $SNAP_USER_DATA/ros
        command: [...]
    ```

  - The data will also be available from the host in: `~/snap/YOUR_SNAP_NAME/current/ros`.

Calling `snapcraft` give the following error

```
Failed to install GPG key: unable to establish connection to key server 'keyserver.ubuntu.
com'

Recommended resolution:
Verify any configured GPG keys.

Detailed information:
GPG key ID: C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
GPG key server: keyserver.ubuntu.com
```

- If the problem is persistent, it's most probably a DNS issue.

[450] https://snapcraft.io/docs/home-interface
[451] https://snapcraft.io/docs/home-interface

- To verify if it's a DNS issue, if the following command succeeds it's most probably a DNS issue: `sudo -E apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654`
- We can also verify that the port `11371` is not blocked or occupied.

ROS snap with shared memory doesn't receive or send data on topic

- If you have properly followed the *ROS snap and shared memory how to guide* (page 115) but still have problems, make sure that the different processes publishing/subscribing ROS 2 data over shared memory are using the same `USER`. FastDDS shared memory can generate communication problems[452] if access from different users.
- Keep in mind that snap services are running with the `root` user while CLI applications might use a less privileged user (i.e: `ubuntu`) causing the FastDDS shared memory communication problems.
- **Plugins (page 191)**
  The Snapcraft plugins to build ROS parts.
- **Extensions (page 191)**
  The Snapcraft extensions to help you snap ROS applications.
- **FAQ & Troubleshooting (page 192)**
  ROS & ROS 2 snap common questions and troubleshooting.

See Also

- **Snap reference**[453]
  Reference section detailing which options can be used, what functions the API supports, which environment variables can be accessed, and the contents of gadget.yaml.
- **Snapcraft reference**[454]
  Reference section detailing which plugins we offer, which interfaces we can use, and what we can add to snapcraft.yaml.
- **Ubuntu Core reference**[455]
  Reference section for detailing which options can be used, what functions the API supports, which rescue modes are supported, and the contents of gadget.yaml.

[452] https://github.com/eProsima/Fast-DDS-docs/blob/master/docs/fastdds/transport/shared_memory/shared_memory.rst?plain=1#L71:L78
[453] https://snapcraft.io/docs/snap-reference
[454] https://documentation.ubuntu.com/snapcraft/stable/reference/
[455] https://ubuntu.com/core/docs/reference

3.2. ROS ESM

In this page you will find the list of the ROS packages that are supported in the Expanded Security Maintenance (ESM) service from Canonical under the Ubuntu Pro offering.

Generally, the packages that we support for each included ROS version are the ones required from the meta-package `ros-base` (see REP142[456] and REP2001[457]).

3.2.1. List of ROS ESM packages

In this page you will find the list of the ROS packages that are supported in the Expanded Security Maintenance (ESM) service from Canonical under the Ubuntu Pro offering.

Generally, the packages that we support for each included ROS version are the ones required from the meta-package `ros-base` (see REP142[458] and REP2001[459]).

Note:

Need additional packages or distributions covered under ESM? Canonical can add and support your specific ROS packages or distributions as part of a commercial engagement. If your organization has unique needs beyond the default set, contact us[460] to explore custom ESM support.

[460] https://ubuntu.com/robotics#get-in-touch

ROS 1 Noetic

- **ros-noetic-ros-base**
- **ros-noetic-ros-core**
- ros-noetic-actionlib
- ros-noetic-actionlib-msgs
- ros-noetic-actionlib-tools
- ros-noetic-bond
- ros-noetic-bond-core
- ros-noetic-bondcpp
- ros-noetic-bondpy
- ros-noetic-catkin
- ros-noetic-class-loader
- ros-noetic-cmake-modules
- ros-noetic-common-msgs
- ros-noetic-cpp-common

[456] https://www.ros.org/reps/rep-0142.html#ros-base
[457] https://www.ros.org/reps/rep-2001.html#ros-base
[458] https://www.ros.org/reps/rep-0142.html#ros-base
[459] https://www.ros.org/reps/rep-2001.html#ros-base

- ros-noetic-diagnostic-msgs
- ros-noetic-dynamic-reconfigure
- ros-noetic-gencpp
- ros-noetic-geneus
- ros-noetic-genlisp
- ros-noetic-genmsg
- ros-noetic-gennodejs
- ros-noetic-genpy
- ros-noetic-geometry-msgs
- ros-noetic-message-filters
- ros-noetic-message-generation
- ros-noetic-message-runtime
- ros-noetic-mk
- ros-noetic-nav-msgs
- ros-noetic-nodelet
- ros-noetic-nodelet-core
- ros-noetic-nodelet-topic-tools
- ros-noetic-pluginlib
- ros-noetic-ros
- ros-noetic-ros-base
- ros-noetic-ros-comm
- ros-noetic-ros-core
- ros-noetic-ros-environment
- ros-noetic-rosbag
- ros-noetic-rosbag-migration-rule
- ros-noetic-rosbag-storage
- ros-noetic-rosbash
- ros-noetic-rosboost-cfg
- ros-noetic-rosbuild
- ros-noetic-rosclean
- ros-noetic-rosconsole
- ros-noetic-rosconsole-bridge
- ros-noetic-roscpp
- ros-noetic-roscpp-core

- ros-noetic-roscpp-serialization
- ros-noetic-roscpp-traits
- ros-noetic-roscreate
- ros-noetic-rosgraph
- ros-noetic-rosgraph-msgs
- ros-noetic-roslang
- ros-noetic-roslaunch
- ros-noetic-roslib
- ros-noetic-roslisp
- ros-noetic-roslz4
- ros-noetic-rosmake
- ros-noetic-rosmaster
- ros-noetic-rosmsg
- ros-noetic-rosnode
- ros-noetic-rosout
- ros-noetic-rospack
- ros-noetic-rosparam
- ros-noetic-rospy
- ros-noetic-rosservice
- ros-noetic-rostest
- ros-noetic-rostime
- ros-noetic-rostopic
- ros-noetic-rosunit
- ros-noetic-roswtf
- ros-noetic-sensor-msgs
- ros-noetic-shape-msgs
- ros-noetic-smclib
- ros-noetic-std-msgs
- ros-noetic-std-srvs
- ros-noetic-stereo-msgs
- ros-noetic-topic-tools
- ros-noetic-trajectory-msgs
- ros-noetic-visualization-msgs
- ros-noetic-xmlrpcpp

ROS 1 Melodic

- **ros-melodic-ros-base**
- **ros-melodic-ros-core**
- ros-melodic-actionlib
- ros-melodic-actionlib-msgs
- ros-melodic-bond
- ros-melodic-bond-core
- ros-melodic-bondcpp
- ros-melodic-bondpy
- ros-melodic-catkin
- ros-melodic-class-loader
- ros-melodic-cmake-modules
- ros-melodic-common-msgs
- ros-melodic-cpp-common
- ros-melodic-diagnostic-msgs
- ros-melodic-dynamic-reconfigure
- ros-melodic-gencpp
- ros-melodic-geneus
- ros-melodic-genlisp
- ros-melodic-genmsg
- ros-melodic-gennodejs
- ros-melodic-genpy
- ros-melodic-geometry-msgs
- ros-melodic-message-filters
- ros-melodic-message-generation
- ros-melodic-message-runtime
- ros-melodic-mk
- ros-melodic-nav-msgs
- ros-melodic-nodelet
- ros-melodic-nodelet-core
- ros-melodic-nodelet-topic-tools
- ros-melodic-pluginlib
- ros-melodic-ros
- ros-melodic-ros-base

- ros-melodic-ros-comm
- ros-melodic-ros-core
- ros-melodic-ros-environment
- ros-melodic-rosbag
- ros-melodic-rosbag-migration-rule
- ros-melodic-rosbag-storage
- ros-melodic-rosbash
- ros-melodic-rosboost-cfg
- ros-melodic-rosbuild
- ros-melodic-rosclean
- ros-melodic-rosconsole
- ros-melodic-rosconsole-bridge
- ros-melodic-roscpp
- ros-melodic-roscpp-core
- ros-melodic-roscpp-serialization
- ros-melodic-roscpp-traits
- ros-melodic-roscreate
- ros-melodic-rosgraph
- ros-melodic-rosgraph-msgs
- ros-melodic-roslang
- ros-melodic-roslaunch
- ros-melodic-roslib
- ros-melodic-roslisp
- ros-melodic-roslz4
- ros-melodic-rosmake
- ros-melodic-rosmaster
- ros-melodic-rosmsg
- ros-melodic-rosnode
- ros-melodic-rosout
- ros-melodic-rospack
- ros-melodic-rosparam
- ros-melodic-rospy
- ros-melodic-rosservice
- ros-melodic-rostest

- ros-melodic-rostime
- ros-melodic-rostopic
- ros-melodic-rosunit
- ros-melodic-roswtf
- ros-melodic-sensor-msgs
- ros-melodic-shape-msgs
- ros-melodic-smclib
- ros-melodic-std-msgs
- ros-melodic-std-srvs
- ros-melodic-stereo-msgs
- ros-melodic-topic-tools
- ros-melodic-trajectory-msgs
- ros-melodic-visualization-msgs
- ros-melodic-xmlrpcpp

ROS 2 Foxy

- **ros-foxy-ros-base**
- **ros-foxy-ros-core**
- ros-foxy-action-msgs
- ros-foxy-actionlib-msgs
- ros-foxy-ament-clang-format
- ros-foxy-ament-clang-tidy
- ros-foxy-ament-cmake
- ros-foxy-ament-cmake-auto
- ros-foxy-ament-cmake-clang-format
- ros-foxy-ament-cmake-clang-tidy
- ros-foxy-ament-cmake-copyright
- ros-foxy-ament-cmake-core
- ros-foxy-ament-cmake-cppcheck
- ros-foxy-ament-cmake-cpplint
- ros-foxy-ament-cmake-export-definitions
- ros-foxy-ament-cmake-export-dependencies
- ros-foxy-ament-cmake-export-include-directories
- ros-foxy-ament-cmake-export-interfaces

- ros-foxy-ament-cmake-export-libraries
- ros-foxy-ament-cmake-export-link-flags
- ros-foxy-ament-cmake-export-targets
- ros-foxy-ament-cmake-flake8
- ros-foxy-ament-cmake-gmock
- ros-foxy-ament-cmake-google-benchmark
- ros-foxy-ament-cmake-gtest
- ros-foxy-ament-cmake-include-directories
- ros-foxy-ament-cmake-libraries
- ros-foxy-ament-cmake-lint-cmake
- ros-foxy-ament-cmake-mypy
- ros-foxy-ament-cmake-nose
- ros-foxy-ament-cmake-pclint
- ros-foxy-ament-cmake-pep257
- ros-foxy-ament-cmake-pycodestyle
- ros-foxy-ament-cmake-pyflakes
- ros-foxy-ament-cmake-pytest
- ros-foxy-ament-cmake-python
- ros-foxy-ament-cmake-ros
- ros-foxy-ament-cmake-target-dependencies
- ros-foxy-ament-cmake-test
- ros-foxy-ament-cmake-uncrustify
- ros-foxy-ament-cmake-version
- ros-foxy-ament-cmake-xmllint
- ros-foxy-ament-copyright
- ros-foxy-ament-cppcheck
- ros-foxy-ament-cpplint
- ros-foxy-ament-flake8
- ros-foxy-ament-index-cpp
- ros-foxy-ament-index-python
- ros-foxy-ament-lint
- ros-foxy-ament-lint-auto
- ros-foxy-ament-lint-cmake
- ros-foxy-ament-lint-common

- ros-foxy-ament-mypy
- ros-foxy-ament-package
- ros-foxy-ament-pclint
- ros-foxy-ament-pep257
- ros-foxy-ament-pycodestyle
- ros-foxy-ament-pyflakes
- ros-foxy-ament-uncrustify
- ros-foxy-ament-xmllint
- ros-foxy-bag-recorder-nodes
- ros-foxy-builtin-interfaces
- ros-foxy-class-loader
- ros-foxy-common-interfaces
- ros-foxy-composition-interfaces
- ros-foxy-console-bridge-vendor
- ros-foxy-diagnostic-msgs
- ros-foxy-domain-coordinator
- ros-foxy-eigen3-cmake-module
- ros-foxy-example-interfaces
- ros-foxy-examples-tf2-py
- ros-foxy-fastrtps
- ros-foxy-fastrtps-cmake-module
- ros-foxy-foonathan-memory-vendor
- ros-foxy-geometry2
- ros-foxy-geometry-msgs
- ros-foxy-gmock-vendor
- ros-foxy-google-benchmark-vendor
- ros-foxy-gtest-vendor
- ros-foxy-kdl-parser
- ros-foxy-launch
- ros-foxy-launch-ros
- ros-foxy-launch-testing
- ros-foxy-launch-testing-ament-cmake
- ros-foxy-launch-testing-ros
- ros-foxy-launch-xml

- ros-foxy-launch-yaml
- ros-foxy-libstatistics-collector
- ros-foxy-libyaml-vendor
- ros-foxy-lifecycle-msgs
- ros-foxy-message-filters
- ros-foxy-mimick-vendor
- ros-foxy-nav-msgs
- ros-foxy-orocos-kdl
- ros-foxy-osrf-pycommon
- ros-foxy-osrf-testing-tools-cpp
- ros-foxy-performance-test-fixture
- ros-foxy-pluginlib
- ros-foxy-python-cmake-module
- ros-foxy-rcl
- ros-foxy-rcl-action
- ros-foxy-rcl-interfaces
- ros-foxy-rcl-lifecycle
- ros-foxy-rcl-logging-log4cxx
- ros-foxy-rcl-logging-noop
- ros-foxy-rcl-logging-spdlog
- ros-foxy-rcl-yaml-param-parser
- ros-foxy-rclcpp
- ros-foxy-rclcpp-action
- ros-foxy-rclcpp-components
- ros-foxy-rclcpp-lifecycle
- ros-foxy-rclpy
- ros-foxy-rcpputils
- ros-foxy-rcutils
- ros-foxy-rmw
- ros-foxy-rmw-dds-common
- ros-foxy-rmw-fastrtps-cpp
- ros-foxy-rmw-fastrtps-dynamic-cpp
- ros-foxy-rmw-fastrtps-shared-cpp
- ros-foxy-rmw-implementation

- ros-foxy-rmw-implementation-cmake
- ros-foxy-robot-state-publisher
- ros-foxy-ros2action
- ros-foxy-ros2bag
- ros-foxy-ros2cli
- ros-foxy-ros2component
- ros-foxy-ros2doctor
- ros-foxy-ros2interface
- ros-foxy-ros2launch
- ros-foxy-ros2lifecycle
- ros-foxy-ros2lifecycle-test-fixtures
- ros-foxy-ros2multicast
- ros-foxy-ros2node
- ros-foxy-ros2param
- ros-foxy-ros2pkg
- ros-foxy-ros2run
- ros-foxy-ros2service
- ros-foxy-ros2test
- ros-foxy-ros2topic
- ros-foxy-ros2trace
- ros-foxy-ros-base
- ros-foxy-ros-core
- ros-foxy-ros-environment
- ros-foxy-ros-testing
- ros-foxy-ros-workspace
- ros-foxy-rosbag2
- ros-foxy-rosbag2-compression
- ros-foxy-rosbag2-converter-default-plugins
- ros-foxy-rosbag2-cpp
- ros-foxy-rosbag2-storage
- ros-foxy-rosbag2-storage-default-plugins
- ros-foxy-rosbag2-test-common
- ros-foxy-rosbag2-tests
- ros-foxy-rosbag2-transport

- ros-foxy-rosgraph-msgs
- ros-foxy-rosidl-adapter
- ros-foxy-rosidl-cmake
- ros-foxy-rosidl-default-generators
- ros-foxy-rosidl-default-runtime
- ros-foxy-rosidl-generator-c
- ros-foxy-rosidl-generator-cpp
- ros-foxy-rosidl-generator-dds-idl
- ros-foxy-rosidl-generator-py
- ros-foxy-rosidl-parser
- ros-foxy-rosidl-runtime-c
- ros-foxy-rosidl-runtime-cpp
- ros-foxy-rosidl-runtime-py
- ros-foxy-rosidl-typesupport-c
- ros-foxy-rosidl-typesupport-cpp
- ros-foxy-rosidl-typesupport-fastrtps-c
- ros-foxy-rosidl-typesupport-fastrtps-cpp
- ros-foxy-rosidl-typesupport-interface
- ros-foxy-rosidl-typesupport-introspection-c
- ros-foxy-rosidl-typesupport-introspection-cpp
- ros-foxy-rpyutils
- ros-foxy-sensor-msgs
- ros-foxy-sensor-msgs-py
- ros-foxy-shape-msgs
- ros-foxy-shared-queues-vendor
- ros-foxy-spdlog-vendor
- ros-foxy-sqlite3-vendor
- ros-foxy-sros2
- ros-foxy-sros2-cmake
- ros-foxy-statistics-msgs
- ros-foxy-std-msgs
- ros-foxy-std-srvs
- ros-foxy-stereo-msgs
- ros-foxy-test-msgs

- ros-foxy-tf2
- ros-foxy-tf2-bullet
- ros-foxy-tf2-eigen
- ros-foxy-tf2-eigen-kdl
- ros-foxy-tf2-geometry-msgs
- ros-foxy-tf2-kdl
- ros-foxy-tf2-msgs
- ros-foxy-tf2-py
- ros-foxy-tf2-ros
- ros-foxy-tf2-sensor-msgs
- ros-foxy-tf2-tools
- ros-foxy-tinyxml2-vendor
- ros-foxy-tinyxml-vendor
- ros-foxy-tracetools
- ros-foxy-tracetools-launch
- ros-foxy-tracetools-read
- ros-foxy-tracetools-test
- ros-foxy-tracetools-trace
- ros-foxy-trajectory-msgs
- ros-foxy-uncrustify-vendor
- ros-foxy-unique-identifier-msgs
- ros-foxy-urdf
- ros-foxy-urdfdom
- ros-foxy-visualization-msgs
- ros-foxy-yaml-cpp-vendor
- ros-foxy-zstd-vendor

3.3. Observability

This section provides COS for robotics technical details such as the COS registration server API, FAQ, and troubleshooting tips.

3.3.1. Observability

Warning:

Beta Notice: COS for robotics is currently in *beta*. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

COS for robotics stands for **Canonical Observability Stack for robotics**, and is a superset of COS Lite[461].

This section provides technical details about COS for robotics, such as the COS registration server API, FAQ, and troubleshooting tips.

COS registration server API

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

The `cos-registration-server` exposes a public API. The API potential usages are:

- Devices can register, upload dashboards etc.
- Operators can manipulate the database to modify a device, dashboard etc.
- The `cos-registration-server-k8s` uses the API to retrieve data later shared with integrations[462].

The use of the API is only recommended from outside of Juju. Within Juju, `cos-registration-server-k8s` integrations[463] are recommended.

API endpoints

FAQ & Troubleshooting

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

[461] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[462] https://documentation.ubuntu.com/juju/latest/reference/relation/index.html
[463] https://charmhub.io/cos-registration-server-k8s/integrations

Can I use COS for robotics without Juju and charms?

While this is not a supported use case, you can redeploy the server side without Juju and charms by deploying and integrating all the server side applications manually.

Can I use COS for robotics without snaps?

While this is not a supported use case, you can redeploy the device side without snaps by repackaging and managing yourself the installation, configuration and orchestration of the different agents on the device.

Is Canonical maintaining Foxglove Studio 1?

No, we are not. The development of COS for robotics started before Foxglove Studio 1 got discontinued. We are only providing a packaged version of the latest open source Foxglove studio 1 release, along with a patch to support passing layout by URL. We are not guaranteeing the support of our packaged version of Foxglove studio.

Is COS for robotics compatible with Foxglove Studio 2?

No, Foxglove Studio 2 is not currently supported in COS for robotics. Since it's closed source it's up to the Foxglove company to integrate it if they want.

Is Canonical providing a managed instance of COS for robotics?

Yes, Canonical provides a service of managed instances of COS for robotics for companies.

Is COS for robotics going to be deployable completely open source?

Yes, all the charms and snaps are open source.

Can I integrate a custom/private application in COS for robotics?

Yes, in the case of a server side application, the application must be charmed[464]. Depending on the desired visibility of your charm, you might upload it to charmhub.io[465] or by deploying your own charmstore[466]. Additionally, you could keep your charm local.

In the case of a device application, the application must be snapped[467]. It can then be deployed publicly to the Snap Store[468] or privately on the dedicated Snap Store[469].

[464] https://juju.is/docs/sdk/from-zero-to-hero-write-your-first-kubernetes-charm
[465] http://charmhub.io
[466] https://github.com/juju/charmstore
[467] https://documentation.ubuntu.com/snapcraft/stable/tutorials/craft-a-snap/
[468] https://snapcraft.io/store
[469] https://ubuntu.com/core/docs/dedicated-snap-stores

How can I suggest features to the COS for robotics?

You can reach ubuntu-robotics-community-group@canonical.com. Once publicly released, suggestions can be made on https://discourse.ubuntu.com/ as well as with tickets in the different repositories.

Who is maintaining the charms and snaps?

The robotics team at Canonical. All contributions and suggestions are welcome in all the repositories.

For how long, COS for robotics is going to be maintained?

We currently commit to our rolling releases.

When is COS for robotics going to be publicly released?

The first public release of COS for robotics will happen in the 6 months after the closing of the private beta testing.

Are old revisions of charms and snaps going to receive security updates?

No, all the charms and snaps are only going to be updated on their latest version. Thanks to Juju and snapd, all the updates will be seamless and automatic.

Can I use another VPN than NetBird?

Yes you can, multiple VPNs are available on the Snap Store. The role of the VPN is to secure the connections and provide direct connectivity between the devices and the server.

Does my device need to use Ubuntu?

No, although COS for robotics is recommended for Ubuntu Core[470]. COS for robotics is supported on Ubuntu Server, Desktop and all the Linux distributions supporting the snapd daemon[471].

[470] https://ubuntu.com/core
[471] https://snapcraft.io/docs/installing-snapd

Does my server need to use Ubuntu?

No, although COS for robotics is recommended for Ubuntu server. COS for robotics is supported on all the Linux distributions supporting the snapd daemon[472].

3.4. Reference architecture

This page outlines the reference architecture of the Canonical robotics stack, detailing the essential components and their roles across development, deployment, and observability phases of a robot development.

3.4.1. Robotics reference architecture

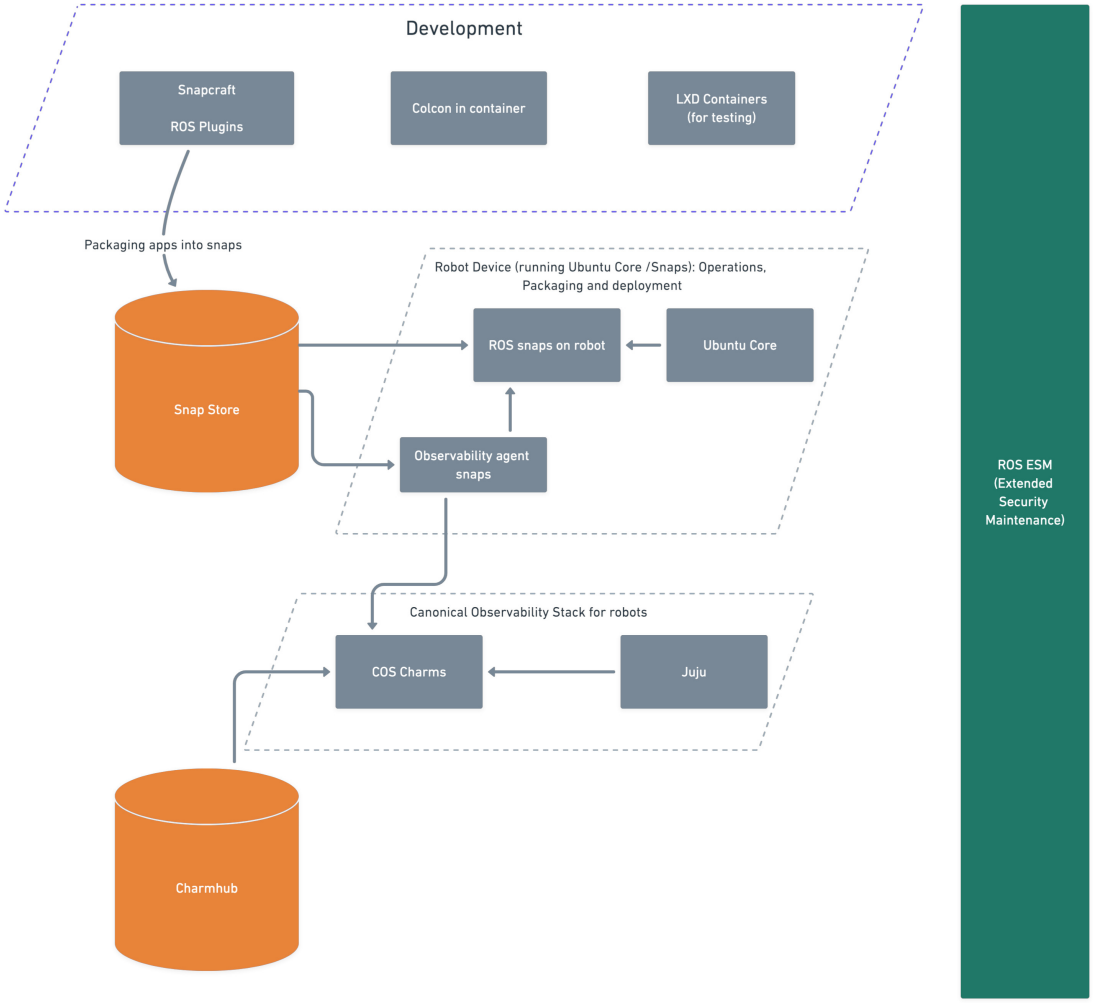
This page outlines the reference architecture of the Canonical robotics stack, detailing the essential components and their roles across development, deployment, and observability phases of a robot development.

Note:

This document is intended for developers and engineers working with the Canonical robotics stack, including those involved in the development, packaging, deployment and observability of ROS snap applications. It provides a **high-level overview** of the architecture and its components, serving as a guide for understanding how they interact within the ecosystem.

[472] https://snapcraft.io/docs/installing-snapd

Canonical Robotics Stack Overview



The Canonical robotics stack streamlines the lifecycle of robotics applications through the following key phases:

1. **Development and Packaging**: Developers use Snapcraft to create and package robotics applications into secure, portable snaps, tested in isolated LXD containers.
2. **Distribution**: Packaged snaps are uploaded to the Snap Store, serving as the central repository for robot devices and observability systems.
3. **Deployment and Operations**: Applications and observability agents run on Ubuntu Core-based robot devices, ensuring a reliable and secure runtime environment.
4. **Observability and Monitoring**: The Canonical Observability Stack (COS), managed by Juju, collects and visualizes performance data from robots, enabling effective monitoring and troubleshooting.

Development Phase

- **Snapcraft**: The tool used to package applications into snaps. Developers use it to create confined, secure, and portable packages.

  Get started with *Snapcraft tutorials for packaging and distributing ROS snap applications* (page 2)

- **LXD Containers**[473]: Lightweight containers used to test applications in isolated environments before deployment.
- **Colcon in-container**[474]: `colcon` extension to build, test and release inside a fresh and isolated ROS environment and transfer the results back to the host.

Packaging and Publishing

- **Snap Store**[475]: The centralized repository for distributing snaps. Once your ROS apps are *packaged with Snapcraft* (page 2), they are uploaded here. This store serves both the robot devices and the observability stack.

Robot Device (Running Ubuntu Core)

- **Ubuntu Core**[476]: A minimal, immutable version of Ubuntu tailored for embedded and IoT devices. It serves as the base OS on the robot.
- **ROS Snaps**: Applications and libraries using the Robot Operating System (ROS), delivered as snaps and running directly on the robot.
- **Observability Agent Snaps** **(page 78)**: Monitoring agents also packaged as snaps, deployed on the robot to collect metrics and logs for observability.

Canonical Observability Stack (COS)

Important:

We have implemented an observability stack (COS for robotics) purposefully for ROS snap applications.

- Tutorials can be found in the *observability section* (page 78).
- How-to guides for customization can be found in the *COS for robotics section* (page 143).

[473] https://documentation.ubuntu.com/lxd/en/stable-5.21/
[474] https://github.com/canonical/colcon-in-container
[475] https://snapcraft.io/docs
[476] https://ubuntu.com/core/docs

- **COS Lite**[477]: A set of Juju-managed applications (charms) that provide metrics, logging, and tracing capabilities. It collects and displays data from observability agents running on robots.
- **Juju**[478]: The orchestration engine used to deploy and manage the COS services.
- **COS Charms**: Deployed from Charmhub[479], these include components like Prometheus[480], Loki[481] and Grafana[482]. They are deployed on a Kubernetes (k8s) cluster to provide observability infrastructure.

Summary of the entire workflow

1. Developers write and package applications using Snapcraft.
2. The snaps are uploaded to the Snap Store.
3. The robot devices fetch ROS snaps and observability agents from the Snap Store.
4. Observability data is sent to the COS Server, which is deployed via charms from Charmhub.
5. Juju orchestrates and manages both the robot-side observability agents and the COS backend services.

Conclusion

This stack ensures a fully integrated workflow for development, deployment, and monitoring of robotics applications, leveraging Canonical's ecosystem of snaps, Juju, and Ubuntu Core.

[477] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[478] https://documentation.ubuntu.com/juju/3.6/
[479] https://charmhub.io/cos-lite
[480] https://charmhub.io/prometheus-k8s
[481] https://charmhub.io/loki-k8s
[482] https://charmhub.io/grafana-k8s

4. Explanation

Explanations and clarifications of the core concepts and key topics that underpin Canonical's robotics solutions.

These guides help understand how all the pieces fit together and get the most out of this ecosystem.

4.1. Snap & Ubuntu Core

All things Snap and Ubuntu Core.

4.1.1. Snaps

Snaps are containers that bundle an application and all its dependencies. As such, snaps offer a solution to build and distribute containerized robotics applications or any software.

Snaps are ideal for robotics developers, as they bundle all your dependencies and assets in one package, making applications installable on dozens of Linux distributions and across distributions versions. You won't even have to install anything else on your robots' operating system, no dependencies, not even ROS if you are using it.

The creation of snaps can be integrated into your CI pipeline, making the updates effortless. Snaps can update automatically and transactionally, making sure the device is never broken.

Snapcraft[483], the tooling for building snaps, comes with native integrations through plugins and extensions dedicated to both ROS[484] and ROS 2[485]; developed and maintained by Canonical.

ROS architectures with snaps

This documentation details different snap architectures that developers can adopt for their ROS applications.

There are two main approaches that can be taken: the monolithic approach and the multi-snap approach.

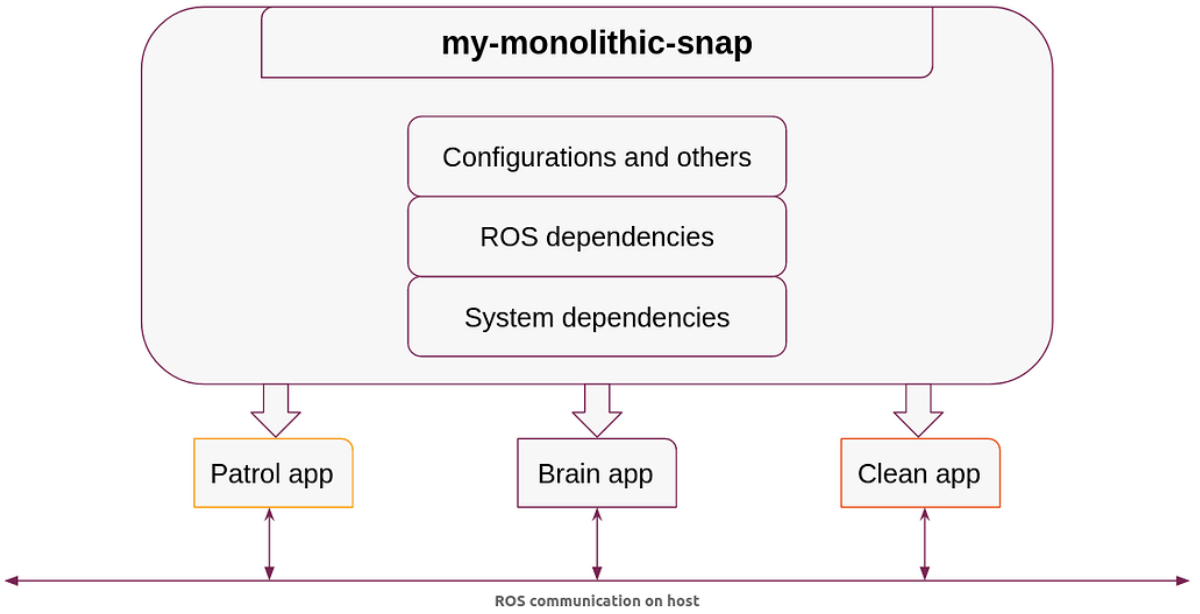
To represent and image the different architectures we will use an example of a robot meant to clean and patrol. This robot consists of three application. The main one being the Brain app. The Brain app is responsible for all the basic features of a robot, controlling the robot, navigating, etc. The Patrol app is simply responsible for patrolling and sending the rights commands to the Brain app. Similarly, the Clean app is meant to send the right cleaning command and logic to the Brain app. The same logic could be used for another robot with the Brain app being a reusable and generic application.

[483] https://snapcraft.io/docs/snapcraft-overview
[484] https://snapcraft.io/docs/ros-applications
[485] https://snapcraft.io/docs/ros2-applications

Monolithic architecture

The monolithic snap for a ROS application is a common approach that involves shipping the complete software stack and applications in one snap. **This approach is recommended for first-time snaps developers.**

The monolithic snap architecture includes all binaries, libraries, configurations, and dependencies in one snap. This means that only one snapcraft.yaml is required to snap all applications. The `snapcraft.yaml` can still contain multiple parts[486] in case some dependencies of the ROS workspace must be built.



As you can see in the picture, the monolithic snap called “my-monolithic-snap” contains all the system and ROS dependencies as well as the configurations and everything needed at run-time. Additionally, this single monolithic snap is exposing three applications. The three applications are communicating using ROS (Could be the network, shared memory, etc.).

Complexity

A monolithic architecture is a common approach since snaps can bundle all dependencies and applications in a single snap.

This approach require writing only one `snapcraft.yaml` to snap all our applications and stack.

The `snapcraft.yaml` can still contain multiple parts in case some dependencies of the ROS workspace must be built.

[486] https://snapcraft.io/docs/snapcraft-yaml-schema

Stability

All the binaries, libraries, configurations, etc live inside the same snap. Since snaps are immutable, having everything in one snap makes an application robust. Additionally, when the snap is being tested, users or devices will always get the same behaviour.

Size

Having everything in one snap avoids duplicating files or binaries across snaps. This is beneficial for devices with limited storage space.

Deployment

In case there is a change in the snap, rebuilding the whole snap is necessary (which can take time for complex robotics applications). Due to changes in libraries (Ubuntu libraries or even ROS ones), the final snap will probably differ a lot from the original one. Even with delta updates, updating the snap will most probably be in the order of magnitude of the snap size.

Reusability

Having two robotics applications using the same "Brain" (navigation stack, hardware controllers, etc) will require building and distributing two completely different snaps. This means that there is very little chance that such a monolithic snap will be deployed on another robot.

You can find a complete implementation of the TurtleBot3 as a monolithic snap on GitHub[487].

Pros of monolithic snaps

- Easy to set up and maintain
- Self-sufficient snaps
- Robust
- Space efficient

Cons of monolithic snaps

- Updates can be heavy to deploy on devices
- Reduced reusability
- Require the release of a completely new snap if the application is slightly changed

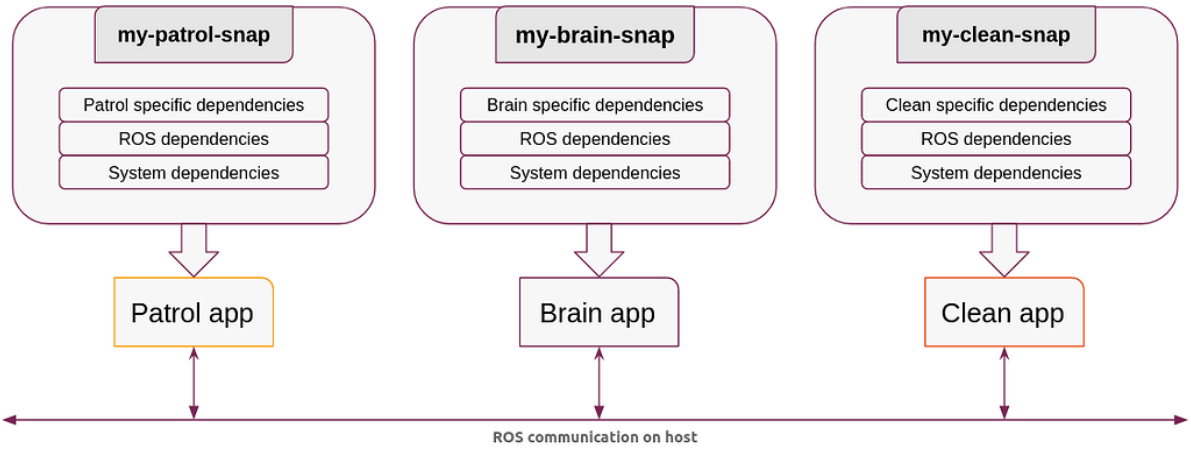
[487] https://github.com/canonical/turtlebot3c-snap

Multi snaps architecture

ROS is a modular system that allows for the exchange of data between applications via the network. With the right network interfaces, two applications can exchange data via ROS without being in the same snap, enabling a complete robot software stack to be deployed over multiple snaps.

The composite of these snaps will then be the complete software stack of the robot.

In the case of our mobile robot application, the snap architecture will look like:



As you can see in the picture, the “Brain” snap called “my-brain-snap” contains everything for the Brain app. Meaning that “my-brain-snap” is carrying its own ROS installation as well as dependencies. Additionally, the “my-patrol-snap” and “my-clean-snap” also carry their ROS installation and specific dependencies. The three apps are exposing their own applications working together communicating with each other with their ROS communication (topics, services, etc) through the host. Each snap is an application, you shouldn’t snap just a library or a ROS package. The content of the snap should make sense as an application.

Complexity

Deploying multiple snaps means multiple snapcraft.yaml files to define, build, and maintain, making the multi-snap architecture more complex.

ROS 2 DDS default implementation `FastDDS` can use shared memory to exchange faster when two `DomainParticipants`[488] are on the same host. You can enable shared memory across multi snaps with the *addition of an extra interface* (page 115).

[488] https://fast-dds.docs.eprosima.com/en/latest/fastdds/dds_layer/domain/domainParticipant/domainParticipant.html

Stability

Multi-snap architecture will require additional testing to ensure that the multiple snaps are working well together, especially when updating the snaps.

In case some snaps will no longer be compatible with each other, channels[489] could be used to clarify the compatibility between snaps.

Moreover, Ubuntu Core's validation set[490] prevents incompatible software installation on a device.

Size

Deploying robot software via multiple snaps is going to take more space on the disk. Since snaps bundle all their dependencies, splitting a robot software stack in multiple snaps will most probably mean shipping different snaps containing some common libraries (e.g. ROS base libraries).

Deployment

When updating a multi snap architecture you don't have to redeploy and update all the other snaps that didn't change. This reduces bandwidth constrains.

Finally, if a set of snaps is needed to run an application, you can pair multiple snaps together for deployment via a private Snap Store[491] or by creating a custom Ubuntu Core image[492].

Reusability

Developers could keep the benefits of ROS modularity and be able to reuse one Brain snap for all the robots while deploying an "application" snap to enable a certain function on the robot. Through this multiple applications could be developed, and they could all work along the Brain snap bringing the basic functionality of a robot.

The developed applications relying on the Brain of a certain robot could be reused on another robot as long as the interface is standardised (same topic names, units, etc).

Pros of multi snaps

- Brings reusability for the snaps
- Allows modularity in applications
- Reduces update bandwidth cost in case of an update

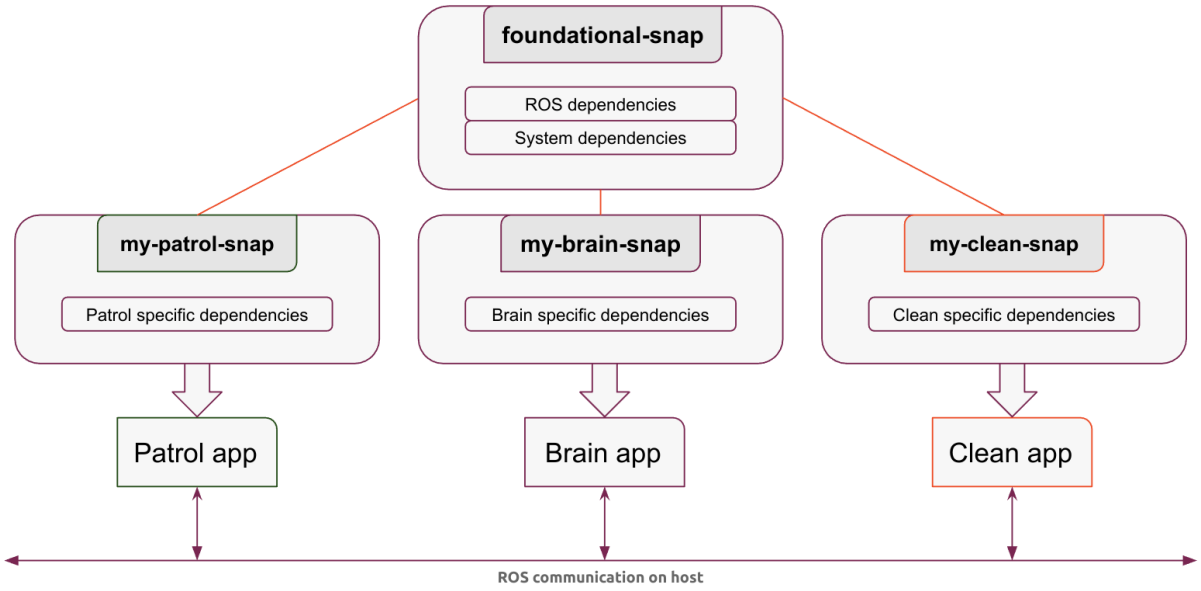
[489] https://snapcraft.io/docs/channels
[490] https://ubuntu.com/core/docs/reference/assertions/validation-set
[491] https://ubuntu.com/core/docs/dedicated-snap-stores
[492] https://ubuntu.com/core/docs/build-an-image

Cons of multi snaps

- Less space efficient
- Potential snaps incompatibility
- Harder to maintain
- Might require additional interfaces[493]
- Need coordination between the releases of the snaps

Multi snaps using content sharing

In the previous section we showed that the robot software stack can be broken down into different snaps, one that contains all the core components of the robot (controllers, drivers, start/stop sequences etc) and a myriad of other application specific snaps (in our example, Clean and Patrol). From an architectural perspective this design is sound, however when looking closer at it (e.g. last diagram) we realise that the same system dependencies and the same ROS packages are duplicated in each and every snap. Indeed, since snaps are self-contained and ship all the dependencies an application may require, it only makes sense to see them duplicated. However this can quickly become a waste of resources (disk space, bandwidth etc). Using the content-sharing feature, we can extract and centralise most of those dependencies in a snap which all the other snap depends upon.



[493] https://snapcraft.io/docs/supported-interfaces

Complexity

The complexity is similar to that of the regular multi snap design presented above. Indeed using the provided extensions (see the extension list[494]), the foundational snap containing the ROS libraries and executables is already provided and maintained on the store. The only difference to the packager is thus the declared ROS extension used.

Stability

As for the multi snap, care has to be taken to make sure that the overall stack is working well across updates. However, using content sharing also requires making sure that the application snaps are API/ABI with the foundational snap. It is thus recommended to rebuild and redeploy the application snaps when an update is available for the foundational snap.

Size

The size of the overall deployment is one of the main advantages of using content-sharing for multi-snap. Indeed, with all the ROS libraries and executable being shared across multiple snaps instead of being replicated in each and every snap, the gain is substantial.

Deployment

When updating a multi snap architecture using content-sharing, you only need to update the snap(s) that has changed. However when updating the foundation snap, you may have to rebuild and redeploy all the snap relying on it. The foundational snap provided by the extensions packages upstream ROS and as such do not make any further guarantees than those provided upstream, especially concerning API and ABI stability.

Take a look at the private Snap Store[495] or at creating a custom Ubuntu Core image[496] as solutions to pin the foundational snap at a given version for your application snap.

Reusability

Similarly to the multi-snap architecture, this allows to decouple the robot specific bits such as controllers, drivers etc from the various applications. It is then much easier to re-use applications across different platforms.

[494] https://snapcraft.io/docs/supported-extensions
[495] https://ubuntu.com/core/docs/dedicated-snap-stores
[496] https://ubuntu.com/core/docs/build-an-image

Pros of multi snap content sharing

- Brings reusability for the snaps
- Allows modularity in applications
- Reduces bandwidth consumption
- Saves some disk space

Cons of multi snap content sharing

- Potential snaps incompatibility
- Harder to maintain
- Requires a tighter release schedule

Conclusion

While the monolithic approach is relatively easy to set up and maintain, it does come with some downsides. For one thing, it requires heavy updates, which can be time-consuming and potentially disruptive. Additionally, it can be less reusable, as developers may need to duplicate code across different applications.

On the other hand, the multi-snap approach offers greater reusability and modularity, which can be a significant advantage in certain contexts. However, it does come with some trade-offs as well. For instance, it may be less space-efficient and harder to maintain than the monolithic approach. Furthermore, it may require additional interfaces and coordination between snap releases in order to function properly. A multi-snap approach is a better solution when scaling up the number of deployments, robots and applications, once the basic setup for content-sharing and release synchronisation is in place.

Identify functionalities and applications of a robotics snap

Since snap is meant to deploy applications, we must define our robot applications. Our robot applications are meant to fulfil the functionalities of a robotics product.

Robot functionalities

The robot functionalities are defined by the product. They refer to how a person/customer can use a product or what they can do with it. These functionalities are defined by the person in charge of the product definition. The functionalities usually come from the customer's needs.

Snaps are going to be a support to distribute those software functionalities. They must be identified before starting the snapping of the software stack.

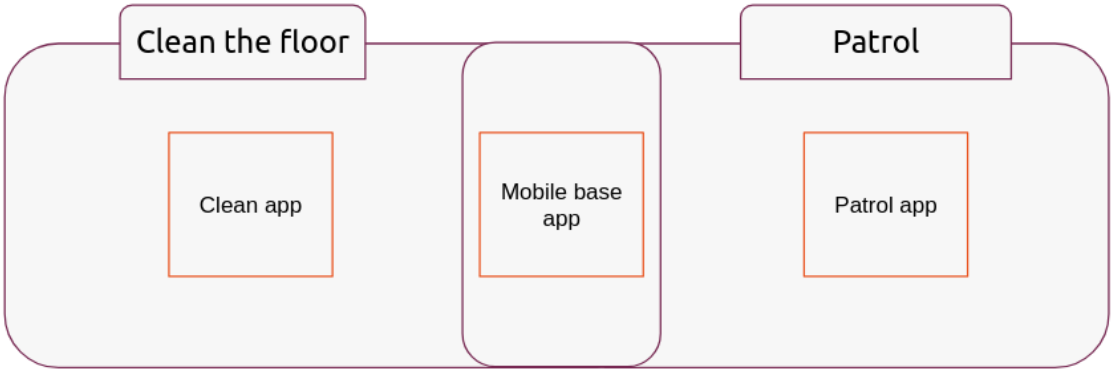
As an example, a supermarket needs a robot to clean the floor and possibly patrol at night. The functionalities associated are:

- The robot must be able to clean the floor.

- The robot must be able to patrol.

For the robot to have such functionalities, developers must create applications. One functionality can be performed by one or multiple applications. Some applications can even be used for multiple functionalities.

In the following drawing, we represented the two functionalities: “Clean the floor” and “Patrol”. Both functionalities need their own application. Additionally, they share an application to control the mobile base.



Snap applications

What are snap applications?

Applications can be programs to call from a terminal or daemons if we need them to start automatically at boot or as background services. One snap can contain one or multiple applications.

An example would be the “bring up”. This application will be responsible for running the robot model and motor controllers. This application would be a background service available for other applications.

Define the role and the scope of the applications

Identify the applications and their roles

Designing our snap is very much like designing a library. We must first define what the API will be. Based on the desired functionalities, we must define what are the key components but also, identify the scope of each component. We want the applications of a snap to be at the same time independent enough to make sense as an application, but also modular enough to be potentially composed with each other.

Think about the final user. When distributing our software, we want to keep it as simple as possible for the user to use it. Seeing our robotics application from the user perspective will guide us toward our snap designing decisions.

Reusing our supermarket robot cleaner example, all our functionalities require the same base (the same bring-up) to control the robot, etc. We will define one background service to start

all the basic controls etc and two different applications for our robot use cases.

Define the applications

Defining the applications will be the task of expanding our ROS workspace into an application-oriented workspace.

For instance, a single snap application can replace all the launch files and commands called.

In the case of the Patrol application, one might need to run one `launchfile` and call a service at the start. All this can be done within one application simplifying the end-user experience.

We must also identify what applications must be started automatically at boot and which ones are commands to be called from the terminal.

The majority of the applications implementing functionalities are going to be started as daemons at boot, but we might want some calibration routines to be only started manually from the terminal.

Snap architecture

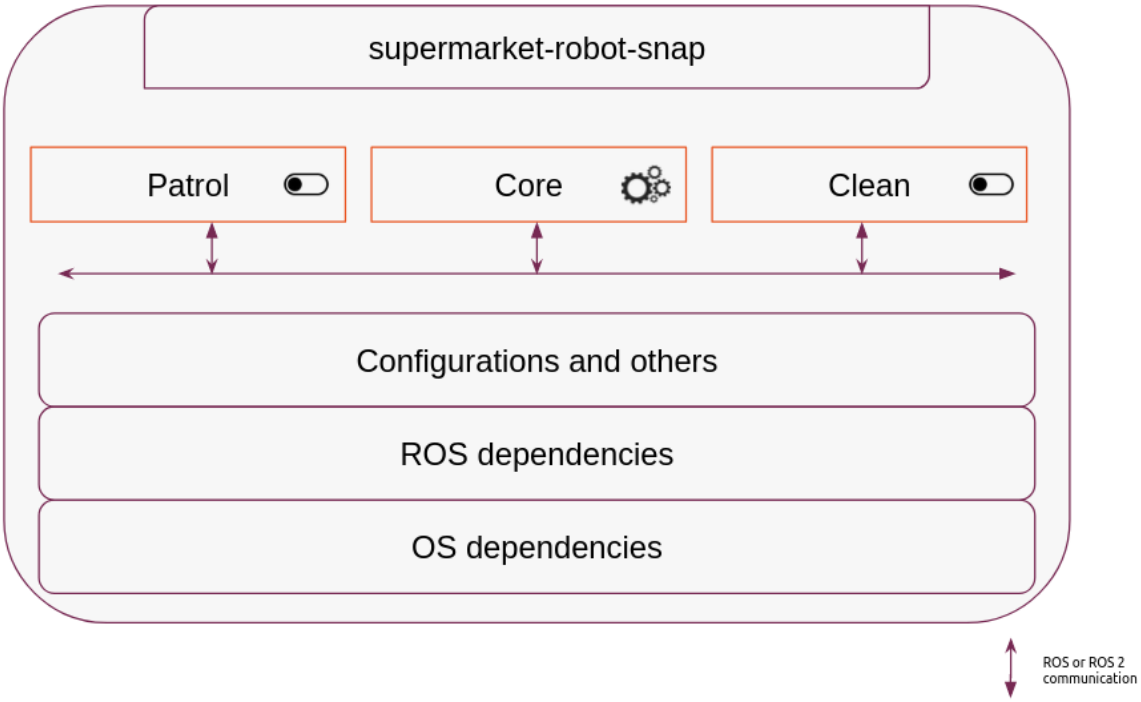
Finally, while the simplest snap architecture is one snap containing all our applications, we could also have a multi-snap architecture in order to distribute our application across multiple snaps. All the possible architectures for a ROS application are described *in the documentation* (page 216) for further investigation.

Example

To illustrate a potential structure of snap applications within a snap, we represented an example of such a structure.

In the following picture, we see “supermarket-robot-snap” containing all our applications. The snap contains all the OS/system and ROS dependencies. These dependencies can be Debian or manually built and installed ROS packages. Similarly, our snap contains any file necessary for our applications. The first application “core”, is a background service meant to be always running. The application is responsible for running all the basic and common nodes (robot model, motors controllers). It is shared and used by the two other applications. The second application “Patrol”, is also a background service, but we can enable/disable it. This application will be responsible for patrolling behaviour meaning sending navigation goals and reporting suspicious activity. The Patrol application is fulfilling our desired “Patrol” functionality. The last application “Clean”, is also an activatable background service. This application is responsible for cleaning of the floor, meaning navigating and activating the correct actuators. This application is fulfilling the “Clean” functionality of our product.

As we can see everything is packaged inside a single snap but yet let the developer define multiple applications covering all the software functionalities of a robot.



Snap configurations and hooks

Snaps have the capabilities to trigger actions depending on `snapd` hooks. These hooks are also the entry point to manage your snap parameters. Snap support being configured by the means of parameters.

As an example for a robotics snap this could be used to select the right LIDAR device.

What are hooks?

A hook is an executable file that runs within a snap's confined environment when a certain action occurs.

Common examples of actions requiring hooks include:

- **Notifying a snap that something has happened**
  Example: If a snap has been upgraded, the snap may need to trigger a scripted migration process to port an old data format to the new one.
- **Notifying a snap that a configuration was done**
  Example: When the user runs snap set|unset to change a configuration option

A hook is defined as an executable within a snap's `hooks/` directory. Hooks are usually POSIX shell scripts. The filename of the executable is based on the name of the hook. All the hooks stored in `snap/hooks/` are automatically going to be imported in our snap. `snapd` will execute the file when required by that hook's action.

The following hooks are currently implemented:

- configure hook[497]

[497] https://snapcraft.io/docs/supported-snap-hooks#heading--the-configure-hook

- full-disk-encryption hook[498]
- gate-auto-refresh[499]
- install hook[500]
- install-device hook[501]
- interface hooks[502]
- prepare-device hook[503]
- pre-refresh hook[504]
- post-refresh hook[505]
- remove hook[506]

Hooks are called with no parameters. When a hook needs to request or modify information within `snapd`, they can do so via the `snapctl` tool, which is always available within a snap's environment. The snapctl tool[507] can be used to access parameters, interface connections or even control our snap daemons.

Since our hooks are scripts we must make sure to make them executable. We can do so with:

```
chmod +x snap/hooks/*
```

Snap configurations

Snaps can have configurations. With these configurations our snaps can expose different behaviour, or we can set parameters options for our background services.

Snap configurations work hand in hand with hooks. Every time we set a parameter the configure hook is called. This might remind the mechanism of ROS dynamic reconfigure[508]. In the hook script, we can decide if we want to take actions based on the new parameter. We can also use the `install` hook to define the parameters.

The command snap set|get allows us to access configurations. An example would be:

```
$ sudo snap set mysnap myconfig=2

$ sudo snap get mysnap myconfig

2
```

The entire set of configuration options can be dumped as JSON by using the `-d` option:

[498] https://snapcraft.io/docs/supported-snap-hooks#heading--fde
[499] https://snapcraft.io/docs/supported-snap-hooks#heading--gate-auto-refresh
[500] https://snapcraft.io/docs/supported-snap-hooks#heading--install
[501] https://snapcraft.io/docs/supported-snap-hooks#heading--install-device
[502] https://snapcraft.io/docs/supported-snap-hooks#heading--interface
[503] https://snapcraft.io/docs/supported-snap-hooks#heading--prepare-device
[504] https://snapcraft.io/docs/supported-snap-hooks#heading--pre-refresh
[505] https://snapcraft.io/docs/supported-snap-hooks#heading--post-refresh
[506] https://snapcraft.io/docs/supported-snap-hooks#heading--remove
[507] https://snapcraft.io/docs/using-snapctl
[508] http://wiki.ros.org/dynamic_reconfigure

```
$ sudo snap get -d snapcraft
{
    "provider": "lxd"
}
```

These configurations can be accessed from within the snap.

As presented, the `snapctl` tool can be used inside the snap to access a parameter with the command:

```
snapctl get myconfig
```

To manage our configuration we will need to define our hooks.

The `install` hook file `snap/hooks/install` :

```
#!/bin/sh -e
# set default configuration value
snapctl set myconfig=false
```

And the configure hook file `snap/hooks/configure`:

```
#!/bin/sh -e
MYCONFIG="$(snapctl get myconfig)"

case "$MYCONFIG" in
        "true") ;;
        "false") ;;
        *)
            >&2 echo "'$MYCONFIG is not a supported value for myconfig." \
        "Possible values are true, false"
            return 1
            ;;
esac

snapctl stop "$SNAP_INSTANCE_NAME"
snapctl start "$SNAP_INSTANCE_NAME"
```

With these two simple hooks, we define a parameter in the `install` hook. When it's set we make sure in the configure hook that the parameter was acceptable. Finally, we decide to restart our application. All that with the help of the `snapctl` command.

Later in our application we can also use the `snapctl` command to get the value of our parameter and use it.

The parameters are kept over updates, but we can of course define a `post-refresh` hook if we want a custom behaviour for our updates and parameters.

Snap data and file storage

Environment variables are widely used across Linux to provide convenient access to system and application properties. Both `snapcraft` and `snapd` consume, set, and pass-through specific environment variables to support building and running snaps.

Snap makes available certain environment variables to identify data and file storage location at run-time.

Snaps runs in a custom environment specifically made for them. Additionally, our snap is strictly confined and immutable. As a result, they have dedicated locations where they can write data. Data environment variables provide different locations for snaps to write data depending on the purpose and lifespan of those data.

Environment variable	Data backed up over updates	Data kept over updates	Accessible from host	Accessible from other snaps
`SNAP_USER_DATA`	Yes	No	Yes	No
`SNAP_USER_COMMON`	No	Yes	Yes	No
`SNAP_DATA`	Yes	No	Read - Yes/Write - Root only	No
`SNAP_COMMON`	No	Yes	Read - Yes/Write - Root only	No
`SNAP_REAL_HOME` (*Requires additional* HOME interface[509])	No	Yes	Yes	Yes

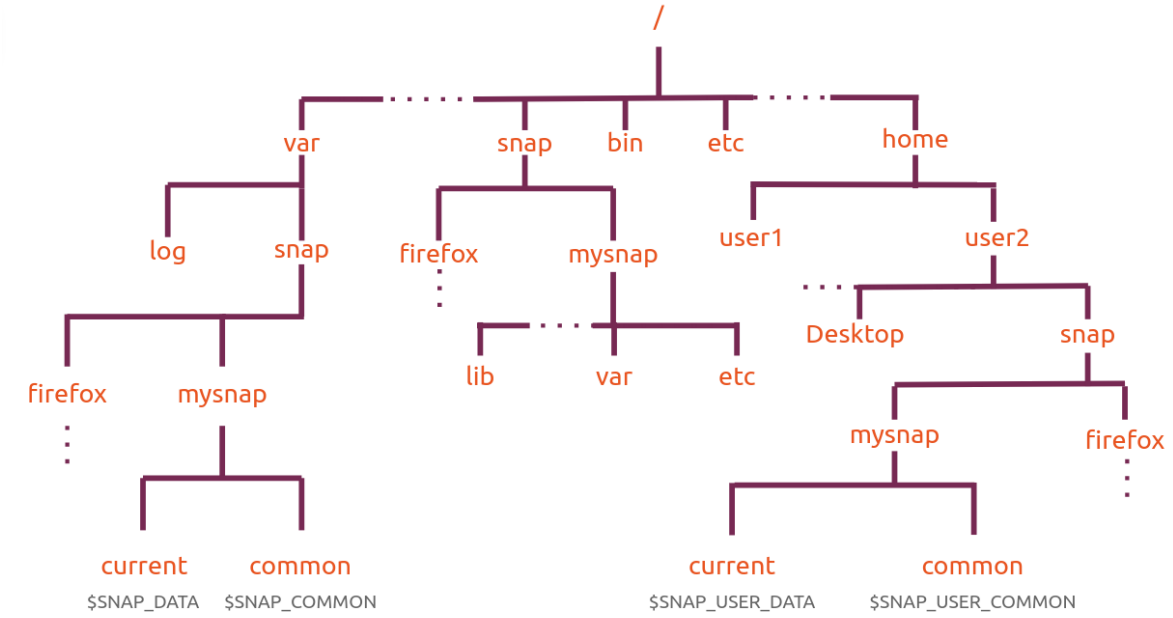
All the following examples are assuming the snap hello-world revision 27.

Note that `SNAP_USER_DATA`, `SNAP_USER_COMMON` depend on the user. When the user is `ROOT` (for daemons and command launched with sudo), `$HOME` correspond to `/root`.

The following diagram shows the relationships and location of a filesystem.

There is a dedicated location for each installed snap for storing data and files.

[509] https://snapcraft.io/docs/home-interface



SNAP_USER_DATA

The `SNAP_USER_DATA` variables store the path to the directory for the user data of a snap. This directory is owned and writable by the current user.

All the content of this directory is backed-up and restored across updates and reverts. This also means that the data stored here won't be available for the next update.

This variable is typically pointing to the host location:

```
$HOME/snap/hello-world/27
```

Also corresponding to:

```
$HOME/snap/hello-world/current
```

This directory can be accessed with read/write permissions by the user even outside the snap.

The typical use case of this variable would be:

- Log files
- Revision specific configuration files
- Temporary runtime created files

SNAP_USER_COMMON

The SNAP_USER_COMMON variable stores the path to the directory for the user data that are common across revisions of a snap. This directory is owned and writable by the user.

Unlike SNAP_USER_DATA this directory is not backed up and restored across snap refresh and revert operations.

This variable is typically pointing to the host location:

```
$HOME/snap/hello-world/common
```

The typical use case of this variable would be:

- Revision specific configuration files
- Any file that we want to keep over the updates

Similarly, there is SNAP_DATA and SNAP_COMMON for system specific data and not user specific data. They are respectively pointing to /var/snap/hello-world/27 and /var/snap/hello-world/common.

The snap documentation for environment variables[510] describe many more features that could be useful in other projects.

Snap environment variables

Environment variables are widely used across Linux to provide convenient access to system and application properties. Both snapcraft and snapd consume, set, and pass-through specific environment variables to support building and running snaps.

All the following examples are assuming the snap hello-world revision 27.

Snap makes available certain environment variables to identify the snap at run-time.

SNAP

The most important environmental variable is *$SNAP*. It contains the path to the directory where the snap is mounted. This is where all the installed files in our snap are visible in the filesystem. Remember, an installed snap is read-only and cannot be changed.

This variable is typically pointing to the host location:

```
/snap/hello-world/27
```

Which is the snap revision specific installation path. It also corresponds to:

```
/snap/hello-world/current
```

The typical use cases of this variable are:

- Locate a file relatively to the snap (an executable, a library, configuration file)
- Extend the $LD_LIBRARY_PATH to a new location inside our snap.

[510] https://snapcraft.io/docs/environment-variables

This variable is available with `snapd` and `snapcraft`.

SNAP_INSTANCE_NAME

The `$SNAP_INSTANCE_NAME` variable is less common.

This variable is going to contain the exact name of the snap. In the case of snap parallel installs[511] a snap could be installed under a different name. We could even have multiple times the same snap installed. By leveraging the `$SNAP_INSTANCE_NAME` we can make sure that we refer to our snap instance and not another one.

The typical use cases of this variable are:

- Restart a daemon application from a hook: `snapctl restart $SNAP_INSTANCE_NAME.MY_APP`
- Make sure there are no confusion between parallel installs

The snap documentation for environment variables[512] describe many more features that could be useful in other projects.

Applications orchestration

Snap robotics applications can be called from the terminal but also run in the background as daemons. Snap and snapcraft offer orchestration features that can become handy for robotics applications.

Command-chain

Valid for CLI: Yes

Valid for daemon: Yes

The `command-chain` keyword allows us to list commands to be executed before our main command. The ros1-noetic extension[513] is actually using this mechanism. Thanks to it, we don't have to worry about sourcing the ROS environment in the snap.

The main difference between `command-chain` and simply adding commands to the script launcher is that `snapd` is aware of it. Hence, if we are trying to debug something (with `snap run --shell myapp`) the command chain is still going to be called.

Note that an `exec $@` is necessary at the end of our `command-chain` scripts since our actual command is given as an argument of the `command-chain`.

This means that with the following snapcraft example:

```
apps:
my_app:
    command-chain: [command_chain_script1, command_chain_script2]
    command: main_command
```

[511] https://snapcraft.io/docs/parallel-installs
[512] https://snapcraft.io/docs/environment-variables
[513] https://snapcraft.io/docs/ros-noetic

The generated call will be:

```
./command_chain_script1 command_chain_script2 main_command
```

Potential use cases of command-chain are:

- Set up a shell environment (like setting up a ROS environment)
- Wait for another service to be started

Stop-command

Valid for CLI: No

Valid for daemon: Yes

`stop-command` allows one to specify a script, or a command, to be called right before the stop signal is sent to a program. This is only available for daemons since this is triggered by the `snap stop` command.

Potential use cases of stop-command are:

- Make sure everything is synchronised
- Wait for a job to finish
- Save before exiting

Post-stop-command

Valid for CLI: No

Valid for daemon: Yes

Similarly to the `stop-command` entry, the `post-stop-command` is also calling a command, but this time, only after the service is stopped. This means that in the sequence we are calling `stop-command`, then stopping the command with a signal and once it's done we call the `post-stop-command`. Also, only available for daemons.

Potential use cases of `post-stop-command` are:

- Clean-up after program exited (temporary files etc)
- Notify a system that the command has stopped
- Move generated files

The `snapcraft` documentation for daemons[514] describes many more features that could be useful in other projects.

[514] https://snapcraft.io/docs/services-and-daemons

Vcstool and rosinstall file

In ROS, it's common to have the list of repositories listed in a rosinstall file[515]. These files are read by Vcstool[516] to import the specified repositories.

In the `snapcraft.yaml`, writing multiple parts in order to cover multiple git repositories might not be necessary. Instead, a `rosinstall` file could be used.

Snapcraft support various source-type[517] but `rosinstall` is not part of the default implementation. Fortunately, we can still leverage the features of `Vcstool` within snapcraft by the means of the overrides part steps[518] feature.

Overrides part steps

Snapcraft provides plugins that ease the build steps of our parts. It automatises everything based on the most common way to use a tool.

Obviously, sometimes we might need to do things slightly differently. And that is why snapcraft has an overrides part steps[519] feature. This feature allows overriding and customising steps of a part's life-cycle[520] (pull, build, stage, and prime). Also, we can still call the default step action within our script. As an example, we display a message at the end of our build with:

```
parts:
foo:
    plugin: colcon
    # ...
    override-build: |
    craftctl build
    echo "Everything built!"
```

That is exactly what we need for our `rosinstall` case.

Using Vcstool

With a `rosinstall` file call `my_robot.rosinstall` placed at the root of our repository, we could simply call `Vcstool` manually. Also, we should make sure that `python3-vcstool` is listed in our `build-packages` as we will need it at build-time.

```
parts:
workspace:
    plugin: colcon # or catkin
    source: . # import our rosinstall file
    build-packages: [python3-vcstool]
    override-pull: |
    craftctl default # or snapcraftctl pull
```

(continues on next page)

[515] https://docs.ros.org/en/independent/api/rosinstall/html/rosinstall_file_format.html
[516] https://github.com/dirk-thomas/vcstool
[517] https://snapcraft.io/docs/snapcraft-yaml-schema
[518] https://documentation.ubuntu.com/snapcraft/stable/how-to/crafting/override-the-default-build/
[519] https://documentation.ubuntu.com/snapcraft/stable/how-to/crafting/override-the-default-build/
[520] https://snapcraft.io/docs/parts-lifecycle

(continued from previous page)

```
    # Here we are going to use the local .rosinstall file
    vcs import --input my_robot.rosinstall
```

As we can see apart from the `rosinstall` specificity, building a whole robot stack inside a robot is actually as simple as building a basic *talker-listener* example.

Debug the build of a snap

Before the snap is built, things can already go wrong. The parts could fail to build, or the application declaration might even fail, etc.

When snapcraft is building our snap, it's first starting a VM or a container. This way everything is built in an isolated environment. We can step into this environment in order to find out why our snap is not building.

Snapcraft environment

When the command snapcraft is called, by default snapcraft is launching an instance with an isolated environment (can be a container, a VM or even the host if enforced). The instance default directory is `/root` which is containing multiple folders:

```
├── parts/
├── prime/
├── project/
├── snap/
└── stage/
```

Most of these directories correspond to different lifecycle steps[521] of our snap building.

Project

This is the simplest directory. It's simply the mount point of the directory from where we called the snapcraft command. The debugging value of this directory is rather low.

Parts

`Parts` will contain different directories. One for each part. Inside each directory we will find a copy of the sources as well as the different build artefacts. We will retrieve our ROS workspace as well as the `build/`, `install/` and `log/` folders. They might be useful to find out why our part is not building. We could for example check the content of log, cache files etc.

This corresponds to the different `pull` and `build` steps of the snap.

[521] https://snapcraft.io/docs/parts-lifecycle#heading--steps

Stage

This corresponds to the `stage` step of the snap. The directory is going to be populated if the build step went well. It's in this directory that we can make sure that everything we compiled and wanted to be installed in our snap is really present. This directory is populated by the `stage-packages` entry but also by the `depend/exec-depend` from your package.xml.

Prime

This corresponds to the `prime` step. This directory is going to be populated after the stage step. Only what is necessary at run time will be copied to this directory. The `prime` directory should contain absolutely everything our snap needs. It can be a cleaner version of the stage (e.g. without header files) or exactly the same content.

Snap

This directory corresponds to the installed snap data directory. Indeed, some snaps are installed inside our container. An example would be all the `build-snaps` of our parts or even our base snap.

The instance environment itself is used to build our parts. This means that if one needed APT packages to build a part, they should be installed inside the instance itself. Hence, regular commands like `apt`, `dpkg`, etc can be used to inspect the packages available at build time.

Snapcraft debug

At the first error, the *snapcraft* command is going to fail and exit. This can be changed in order to step into the building environment after the error with the option `--debug`. The command is then:

```
snapcraft --debug
```

At the first error, we are going to step into our container in the `/root/project` folder.

We can then verify why our build is failing. Keep in mind that when entering the instance, the environment of the shell won't be specific to any part. One might need to redefine environment variables or sources scripts.

The `--debug` option is the perfect solution in case the `snapcraft` command fails. When developing a snap it's recommended to always enable this flag.

Snapcraft shell-after

Sometimes there is no error but yet the built snap is not what we expected it to be. We can still check the state of the building environment with the flag `--shell-after`. With this flag, once our snap is completely built we will still step into the build environment. The complete command is:

```
snapcraft --shell-after
```

We will enter the instance environment similarly to the *–debug* flag.

Snapcraft try

The snapcraft try command[522] can be used in combination with `snap try` to quickly test a snap and fix issues.

`snapcraft try` runs through the build process to the completion of the prime stage. It then exposes the resultant prime directory to the snapcraft directory.

Once you run:

```
snapcraft try
```

To use the built snap you can then run:

```
snap try prime
```

This way you can modify the content of the `prime/` directory without having to rebuild the snap. As an example we could modify a `launchfile`, a python script, etc.

Debug a snap application

Once a snap is built and installed one might face unexpected problems like missing a configuration file or a library or simply not the expected behaviour.

Even if snaps are immutable there are still means to introspect and analyse their state as well as running tests inside the snap environment.

Keep in mind that we want to debug the packaging. Debugging your own code etc should be done prior and outside the packaging.

Snap logs

When we call a snap app command, the logs are usually printed in the terminal. On the contrary when we are running a daemon, the logs are not instantly visible. Snap daemon are `systemd` background services.

To visualise the last 100 lines of a snap daemon:

[522] https://snapcraft.io/docs/snap-try

```
sudo snap logs SNAP_NAME.DAEMON_NAME -n 100
```

In case we want to wait for new lines and print them as they come in. We can use the -f option.

Additionally, if our snap contains multiple applications daemons, we can log them all by simply omitting the application name:

```
sudo snap logs SNAP_NAME -f
```

Similarly, we can use the pure systemd command:

```
journalctl -u snap.SNAP_NAME.DAEMON_NAME.service
```

As we can see, snap daemon are simply `systemd` services with the `snap.` prefix and the `.service` suffix. For the sake of simplicity, the `snap logs` command is preferred,

If the robot is not behaving as expected this should be the first action.

This way we can quickly check the output of our `launchfiles`.

Snap environment

When our snap is built and installed but doesn't work as expected there are solutions. Our snap is running in a confined and containerised environment making the debugging sometimes more difficult.

Snap file structure

If we are curious about the folder/file structure of our system, we can simply check it from our host. As an example, we can list the files at the root of our snap system with:

```
$ ls /snap/SNAP_NAME/current

etc/ lib/ meta/ opt/ usr/ var/
```

Everything under this directory `/snap/SNAP_NAME/current` is only for our snap. When strictly confined, our snap cannot access anything outside this (apart from the different data directories). Checking the files in this directory can sometimes be enough to figure out our issue.

Snap run

When it's not enough, and we actually need to be in the snap environment to debug, we can run the snap run command along with the `--shell` option.

```
snap run --shell SNAP_NAME.APP_NAME
```

This command is actually going to start a shell of the snap app environment (`command-chain` included) instead of starting our app. This way, **we have the exact same environment** to run any kind of command. We can even call the application ourselves if we want to reproduce the issue. We can also potentially launch an entirely different command. Furthermore, we can then check the environment variables, files location, permissions etc.

Note that when calling `snap run --shell` the started shell will be in the same directory where the command was called. Shelling into the snap environment will keep the original snap permissions.

This method has a very high potential for debugging our snaps.

Debugging a missing library

Robotics applications sometimes rely on hundreds of dynamic libraries. A very common error is that when an application starts, a library is missing. When this is happening a good way to verify that is to use `ldd`. It will print the shared object libraries and their paths as they are found. So after calling the `snap run --shell SNAP_NAME.APP_NAME` it's the perfect moment to call `ldd` on a library.

```
$ ldd $SNAP/opt/ros/foxy/lib/librmw.so

linux-vdso.so.1 (0x00007fff841c5000) librcutils.so => /snap/MYSNAP/REVISION/opt/ros/foxy/
lib/librcutils.so (0x00007f4c2276a000)

libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4c2251b000)

libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f4c22515000) /lib64/ld-linux-x86-
64.so.2 (0x00007f4c2278d000)
```

In case one library is marked as not found, the definition of `$LD_LIBRARY_PATH` is usually at fault.

If a library is missing in the library path but is installed, we can find its location with the help of the find command after entering the snap shell:

```
find $SNAP -type f -name "librmw.so"
```

Snap connections

Snaps are strictly confined, but they can access our host by the means of interfaces. These interfaces can sometimes be the source of our problems. Let's see the different ways to troubleshoot them.

The very first thing that can help is the following command:

```
$ snap connections lxd

Interface        Plug                Slot             Notes

lxd              multipass:lxd       lxd:lxd          -
Lxd-support      lxd:lxd-support     :lxd-support     -
network          lxd:network         :network         -
network-bind     lxd:network-bind    :network-bind    -
system-observe   lxd:system-observe  :system-observe  -
```

Here we listed all the connections of our snap `lxd`. As we can see all the interfaces have a `plug` and a `slot`. This means that everything is connected.

Some interfaces are auto-connect while some others are not. This means that we must connect them manually.

To do so we must use the command snap connect. An example of the usage would be:

```
sudo snap connect SNAP_NAME:camera :camera
```

The command above presupposes that our snap application had the camera plug declared[523]. Similarly, we can use the `snap disconnect` command to undo the `connect` action.

When a snap cannot access a host resource that it was declared to access, checking the connection is usually a good starting point.

One can request "auto-connect" on the forum[524] of an interface that doesn't auto-connect for a snap.

Snappy-debug

The snap, being strictly confined, sometimes tries to access resources that were not declared. It generates an App Armor[525] policy violation that might be hard to diagnose.

The easiest way to find and fix policy violations is to use the snappy-debug tool[526]. It's a tool provided by Canonical and allows us to:

- watches syslog for policy violations
- shows them in a human-readable format
- get recommendations for how to solve them

We can install the `snappy-debug` tool with the command:

```
sudo snap install snappy-debug
```

We can then call the `snappy-debug` command and in another terminal, call our snap app. The *snappy-debug* tool could then produce an output similar to the following one:

```
mars 02 17:27:39 user-computer audit[721546]: AVC apparmor="DENIED" operation="open"
profile="snap.SNAP_NAME" name="/dev/video0" pid=721546 comm="APP" requested_mask="c"
denied_mask="c" fsuid=1000 ouid=1000
```

In this log we can see that the access to `/dev/video0` was attempted and denied. This gives us the information that either our snap misses the `camera plug`, or that we simply forgot to connect it.

[523] https://snapcraft.io/docs/camera-interface
[524] https://snapcraft.io/docs/process-for-aliases-auto-connections-and-tracks
[525] https://apparmor.net/
[526] https://snapcraft.io/snappy-debug

4.1.2. Ubuntu Core

Ubuntu Core[527] is a version of the Ubuntu operating system designed and engineered for IoT and embedded systems.

Ubuntu Core updates itself and its applications automatically. Snap packages are used exclusively to create a confined and transaction-based system. Security and robustness are its key features, alongside being easy to install, easy to maintain, and easy to upgrade.

Ubuntu Core is great for robotics developers since it unlocks a complete infrastructure to reliably and securely deploy your robot, including automatic rollbacks on updates, delta updates, full disk encryption, and more. Ubuntu Core is ideal for embedded devices because it manages itself.

Learn more about Ubuntu Core[528]

Download Ubuntu Core certified images for different boards[529]

Learn more about the security features of Ubuntu Core and Snaps[530]

Review guidelines for hardening Ubuntu Core on your robot[531]

4.1.3. Dedicated Snap Store

Dedicated Snap Store is a custom application store tailored to software distribution across fleets of devices. This store allows you to create, publish and distribute software on one centralised platform, with reliable over-the-air updates to your devices in a secure and validated way. The Dedicated Snap Store is adopted by a large number of enterprises around the world, with a variety of IoT use cases.

The Dedicated Snap Store is a commercial service. Canonical also offers a global store[532], open to all,all with a wealth of software available to you and your devices.

The Dedicated Snap Store supports robotics deployment and management by providing companies with a trusted and reliable infrastructure. It is suited for managing software on huge numbers of distributed devices. It is private to you, the users you assign to the store and the devices you own.

No network? No problem. The Store comes with a proxy to allow devices with restricted network access to connect to the global store, even through a firewall. We also provide the air-gapped mode of the Store Proxy, allowing devices to be completely disconnected from the internet to receive updates.

Note:

Learn more about the Dedicated Snap Store[533].

[533] https://ubuntu.com/internet-of-things/appstore

[527] https://ubuntu.com/core/docs
[528] https://ubuntu.com/core/docs
[529] https://ubuntu.com/certified
[530] https://ubuntu.com/core/docs/security-and-sandboxing
[531] https://canonical-robotics.readthedocs-hosted.com/en/latest/explanations/security/securing-ros-robotic-platforms
[532] https://snapcraft.io/

4.2. Security

This page provides an overview of and references to various topics related to security for robotics on Ubuntu.

4.2.1. Security

This page provides an overview of and references to various topics related to security for robotics on Ubuntu.

Harden your robot

Check out the following how-to guide with steps to secure your robot with Ubuntu:

- *Hardening your robot* (page 176)

Security considerations for COS for Robotics

Security considerations for operating COS for robotics, including TLS encryption, certificate management, and device key protection:

- *Security considerations for COS for Robotics* (page 256)

Ubuntu Pro and ROS ESM

Ubuntu Pro enhances the security and compliance of your Ubuntu systems.

ROS ESM[534] is part of Ubuntu Pro[535] for Applications subscription.

ROS ESM guides

To learn more about ROS ESM, check out the explanation pages:

What is ROS ESM

As part of Ubuntu Pro[536] for Applications subscription, ROS ESM[537] gives you a hardened and long-term supported ROS system for robots and its applications. Even if your ROS distribution hasn't reached its end-of-life (EOL), you can count on backports for critical security updates and CVEs fixes for your environment. In addition, all upstream changes are evaluated by hand to minimise breaking changes. By enabling our repositories, you will get trusted and stable binaries for your environment. If you are a standard or advanced customer, you also get ROS support. This provides you with a single point of contact to log ROS bugs.

[534] https://ubuntu.com/robotics/ros-esm
[535] https://ubuntu.com/pro/beta
[536] https://ubuntu.com/pro/beta
[537] https://ubuntu.com/robotics/ros-esm

Benefits

ROS ESM provides four key benefits:

- 10 year LTS release lifetime for ROS bringing the highest level of security and compliance
- Security patching for over 25,000 packages in ROS, Ubuntu Universe and Ubuntu main
- Better security KPIs as critical CVEs patches are applied on average in less than 24h
- Single point of contact to log bugs and propose fixes to guarantee timely and quality fixes

For more information, you can visit:

- Ubuntu Pro Service Description, Extended Security Maintenance (ESM) page,[538]
- ROS ESM Specialist service description[539]

Or, if you have specific questions, contact us for more information[540]!

Enabling and using ROS ESM

For step-by-step guidance, take a look at our how-to guides on enabling and day-to-day usage of ROS ESM:

- *Enable ROS ESM* (page 167)
- *Set up a ROS ESM environment* (page 170)
- *Combine ESM and Upstream ROS Components* (page 172)

Interaction between ROS ESM and ROS upstream

When enabling ROS ESM using `pro enable ros` as described in *this guide* (page 167), some changes are made to `apt` configuration and it's important to be aware of those details.

Changes to PPAs

If you followed the official installation instructions for ROS 1[541] or ROS 2[542], you now have some additional files in your `/etc/apt/sources.list.d` folder.

These files are telling your apt software which server is able to provide specific packages. For the ROS ecosystem the files are usually called:

- `/etc/apt/sources.list.d/ros-latest.list` for ROS 1
- `/etc/apt/sources.list.d/ros2.list` for ROS 2

[538] https://ubuntu.com/security/esm
[539] https://ubuntu.com/legal/ubuntu-pro-description/ros-esm-service-description
[540] https://ubuntu.com/robotics/ros-esm#get-in-touch
[541] https://wiki.ros.org/noetic/Installation/Ubuntu
[542] https://docs.ros.org/en/rolling/Installation/Ubuntu-Install-Debs.html

For example, when you request `apt` to install `ros-noetic-std-msgs`, it will fetch it from the server written inside the `ros-latest.list` file. When you enable ROS ESM, you will notice that a new configuration file is added inside your `sources.list.d` folder. The file is called `ubuntu-ros.list` and tells `apt` to fetch packages from esm.ubuntu.com[543]. Our ESM packages can be distinguished because their version follows the pattern `X.Y.Z+<ubuntu-version>-<counter>` where:

- `X.Y.Z` is the usual ROS versioning system
- `<ubuntu-version>` is an LTS name such as `20.04.1`
- `<counter>` is a single integer.

After enabling ROS ESM you can inspect a package with `apt-cache`. For example:

```
apt-cache policy ros-foxy-std-msgs
```

```
ros-foxy-std-msgs:
  Installed: 2.0.5-1focal.20230527.044919
  Candidate: 2.0.5+20.04.1-0
  Version table:
     2.0.5+20.04.1-0 500
        500 https://esm.ubuntu.com/ros/ubuntu focal-security/main amd64 Packages
 *** 2.0.5-1focal.20230527.044919 500
        500 http://packages.ros.org/ros2/ubuntu focal/main amd64 Packages
        500 http://repo.ros2.org/ubuntu/main focal/main amd64 Packages
        100 /var/lib/dpkg/status
```

This indicates that `ros-foxy-std-msgs` is available on ros.org[544] at version `2.0.5-1focal`, while the ROS ESM repository provides version `2.0.5+20.04.1-0`.

Apt automatically decides to upgrade from upstream `ros.org` to ROS ESM if both are available, you can confirm this by running:

```
$ apt list --upgradable
ros-foxy-std-msgs/focal-security 2.0.5+20.04.1-0 amd64 [upgradable from: 2.0.5-1focal.
20230527.044919]
```

If you want to be sure you no longer consume any End-of-Life upstream packages, you should remove the `ros-latest.list` and `ros2.list` files and update your apt cache.

```
sudo rm /etc/apt/ros-latest.list /etc/apt/ros2.list
sudo apt update
```

[543] https://esm.ubuntu.com
[544] http://ros.org

Changes to rosdep

ROS ESM is its own ROS distribution, and thus provides its own distribution and `rosdep` files. If you already have upstream ROS installed and initialised (e.g. you previously ran `sudo rosdep init`), you'll need to make sure you install `rosdep` from ESM and re-initialise it as follows:

Noetic/Foxy (Python3)

Kinetic/Melodic (Python2)

```
sudo apt install python3-rosdep
sudo rm /etc/ros/rosdep/sources.list.d/20-default.list
sudo rosdep init
rosdep update
```

```
sudo apt install python-rosdep
sudo rm /etc/ros/rosdep/sources.list.d/20-default.list
sudo rosdep init
rosdep update
```

Now, the output of running `rosdep update` will look like the following:

```
$ rosdep update
reading in sources list data from /etc/ros/rosdep/sources.list.d
Hit https://ros.robotics.ubuntu.com/rosdep/osx-homebrew.yaml
Hit https://ros.robotics.ubuntu.com/rosdep/base.yaml
Hit https://ros.robotics.ubuntu.com/rosdep/python.yaml
Hit https://ros.robotics.ubuntu.com/rosdep/ruby.yaml
Hit https://ros.robotics.ubuntu.com/rosdep/fuerte.yaml
Query rosdistro index https://staging.ros.robotics.ubuntu.com/rosdistros/index-v4.yaml
Add distro "foxy"
Add distro "kinetic"
Add distro "melodic"
Add distro "noetic"
updated cache in /home/user/.ros/rosdep/sources.cache
```

Security vulnerability audits in ROS ESM

ROS ESM is maintained by the Ubuntu Robotics Team, in a close partnership with the Ubuntu security team. Its security processes apply the same rigor that secures millions of Ubuntu systems. This collaboration ensures timely triage and patching of vulnerabilities, as quickly as 24 hours for reported critical issues. ROS packages benefit from Canonical's expertise in vulnerability disclosure and non-disruptive updates, all aligned with Ubuntu LTS standards. Delivered through Ubuntu Pro, ROS ESM offers unified, long-term security and compliance for robotics deployments.

In addition to addressing reported security vulnerabilities, the Ubuntu Robotics team proactively runs a dedicated security analysis pipeline for ROS packages.

A close look at ROS ESM's ongoing security audits

The team leverages advanced static analysis using tools like Semgrep[545], Bandit[546], and Coverity[547] to detect memory safety vulnerabilities, insecure APIs, logic errors, and other high-risk code patterns in the ROS distributions it supports. Identified issues undergo rigorous triage by engineers, with critical findings validated through dynamic analysis and vulnerability proof-of-concept testing. Fixes are then delivered through a **controlled, quality-focused release process** that ensures both reliability and traceability. Security fixes are tested and staged before being deployed to users, and where appropriate, Canonical contributes them to upstream ROS repositories, following a **responsible, coordinated disclosure**, and ROS's own Vulnerability Disclosure Policy[548].

This is backed by a **robust, purpose-built CI infrastructure** that spans multiple stages of quality assurance. The pipeline runs a sequence of automated checks, including unit tests, ABI stability tests, reverse dependency testing, integration tests with external packages, and full Debian packaging via Bloom[549]. The integration tests confirm that **ROS ESM packages remain installable, compatible, and functional** when used alongside other ROS packages. This helps prevent regressions and ensures compatibility across supported platforms and architectures.

Security issues detected and fixed

As a direct result of this proactive security work, several vulnerabilities in the ROS ecosystem have been **identified, responsibly disclosed, and patched on ROS ESM**. This includes the publication and remediation of multiple High severity CVEs, such as:

- CVE-2025-3753 - Code execution vulnerability in rosbag[550]
- CVE-2024-39289 - Code execution vulnerability in rosparam[551]
- CVE-2024-39780 - YAML deserialization vulnerability in dynparam[552]
- CVE-2024-39835 - Code injection vulnerability in roslaunch[553]
- CVE-2024-41148 - Code injection vulnerability in rostopic[554]
- CVE-2024-41921 - Code injection vulnerability in rostopic[555]

These fixes demonstrate the value of continuous auditing, as well as Canonical's commitment to raising the security baseline for ROS-based systems in production.

If you're running ROS in production, it's important to know whether a specific CVE has been patched in your environment. To do this, check out *How to check if a CVE is fixed in your environment* (page 175).

[545] https://semgrep.dev/
[546] https://github.com/PyCQA/bandit
[547] https://scan.coverity.com/
[548] https://ros.org/reps/rep-2006.html
[549] https://github.com/ros-infrastructure/bloom
[550] https://nvd.nist.gov/vuln/detail/CVE-2025-3753
[551] https://nvd.nist.gov/vuln/detail/CVE-2024-39289
[552] https://nvd.nist.gov/vuln/detail/CVE-2024-39780
[553] https://nvd.nist.gov/vuln/detail/CVE-2024-39835
[554] https://nvd.nist.gov/vuln/detail/CVE-2024-41148
[555] https://nvd.nist.gov/vuln/detail/CVE-2024-41921

These focused how-to guides will show you how to enable and use the service:

- *How to enable ROS ESM* (page 167)
- *How to set up your ROS ESM environment* (page 170)
- *How to check if a CVE is fixed in your environment* (page 175)

4.3. Observability

This section is all about observability. The main focus of our documents is on the Canonical Observability Stack (COS) for robotics.

4.3.1. Observability

Warning:

Beta Notice: COS for robotics is currently in *beta*. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

COS for robotics stands for **Canonical Observability Stack for robotics**, and is a superset of COS Lite[556].

This section explains how Canonical's observability framework extends to robotic systems—connecting devices, applications, and cloud components into one monitoring and data-sharing pipeline.

What is COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

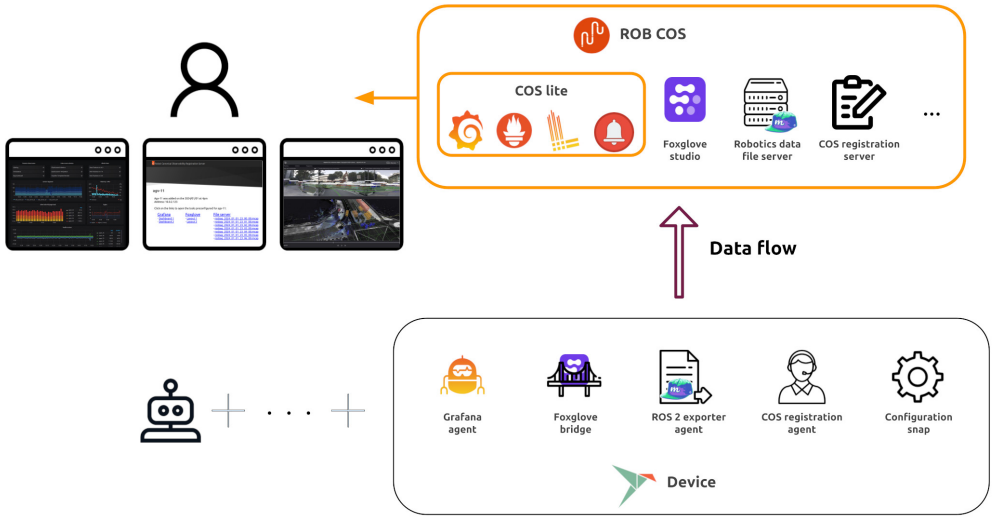
COS for robotics stands for Canonical Observability Stack for robotics and is a superset of COS Lite[557]. COS for robotics brings observability[558] to your robots and devices.

When deploying robots, the need to collect data arises sooner than expected. One might need to visualize live or previously stored data. The necessity for data can have multiple causes: debugging, statistics analysis, monitoring, data collection for machine learning, etc.

Moreover, as new devices are deployed, the observability capacity has to scale effortlessly. For all these reasons COS for robotics has been developed, offering an observability infrastructure and solution for devices.

On the following drawing, we can see a fleet of devices using snaps to push data to the COS for robotics server, which is then available to the user for visualization.

[556] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[557] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[558] https://ubuntu.com/observability/what-is-observability



How COS for robotics works

The COS for robotics consists of two main components:

- the server side: which hosts applications for monitoring, analysis and visualization extending COS lite.
- the device side: a set of snaps that allow the robot to interface and communicate with the server.

The COS for robotics is already including a set of applications on the server and device side. By the modular nature of COS[559], you can easily select a subset of applications or even extend it with open source or even proprietary applications.

The COS for robotics is extending COS Lite in the sense that it can handle robotics data and that the clients can be deployed on devices via snaps.

The server side

The server side (which can run on the cloud, a laptop, or any capable machine) relies on Juju[560], an open source orchestration engine, to easily deploy applications at any scale and Microk8s[561], a lightweight Kubernetes cluster bringing stability, security and scalability.

Every application running in Juju is a charmed operator (charm)[562]. This means the server side can also benefit from charmhub.io[563] to get seamless updates over time.

[559] https://charmhub.io/topics/canonical-observability-stack
[560] https://juju.is/docs/juju/tutorial
[561] https://microk8s.io/docs/getting-started
[562] https://canonical-juju.readthedocs-hosted.com/en/latest/user/reference/charm/
[563] https://charmhub.io/

The COS for robotics consists of a Juju bundle[564] ready to be deployed on any Juju k8s machine[565].

This bundle can easily be extended by the mean of an overlay[566].

Charms bundled in the COS for robotics are responsible for data visualization and data storage. The applications expected on the servers can be:

- Data processing
- Data analytics and visualization
- Monitoring system and data models
- Alert manager
- Logs aggregator
- Anomaly detector
- VPN server

The device side

On devices, the COS for robotics consists of a set of snap[567] packages. Snaps packages are particularly suited for robotics[568] and their limited resources reducing the need for on device operations. Installed snaps will benefit from seamless updates and rollback from the Snap Store[569]. Additionally, thanks to snaps, the device side can run completely from the Ubuntu Core[570] Operating system engineered for IoT and embedded.

Snaps running on the device are responsible for collecting data and syncing them to the server side. By the mean of configuration, device's snaps could collect and synchronize data according to the bandwidth and storage available.

The applications expected on the devices can be:

- Telemetry collectors
- Data collectors (i.e: ROS 2 data)
- Logs collectors
- VPN client
- Device manager client

[564] https://canonical-juju.readthedocs-hosted.com/en/latest/user/reference/bundle/
[565] https://canonical-juju.readthedocs-hosted.com/en/latest/user/explanation/kubernetes-in-juju/
[566] https://canonical-juju.readthedocs-hosted.com/en/latest/user/reference/bundle/
[567] https://snapcraft.io/docs
[568] https://ubuntu.com/robotics/docs
[569] https://snapcraft.io/store
[570] https://ubuntu.com/core/docs

Who is COS for robotics for

The COS for robotics stack is fully open source, so anyone can use it. Since the COS for robotics is meant to observe devices, the typical use case is to observe a fleet of devices. Thanks to the P2P VPN, devices can be on the same site or not.

COS for robotics is deployed with Juju, meaning anyone can deploy COS for robotics as a secure, scalable and resilient server.

The typical use case of COS for robotics is for a company to deploy a complete observability stack for a fleet of devices.

Whether deployed on self-hosted infrastructure or in the cloud, COS for robotics can meet all the observability needs of an organization.

Additionally, Canonical offers a managed version of COS for robotics so you can focus on your business. We will run the best-in-class open source monitoring tools you need for the observability of your applications.

You can learn more about open source observability on ubuntu.com/observability[571].

The generic COS documentation can be found on charmhub.io[572].

Alert rules configuration from device

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

Alerts rule file can be defined in Prometheus and Loki to trigger notifications on the Alert-manager.

These alert rule files can be provided by the devices directly and then hosted on the `COS-registration-server`[573].

This allows devices to deploy the alert configurations they need specifically or configurations that could be used by any other device.

Rules uploaded on the COS-registration-server

The `COS-registration-server` can host alert rule files. The supported rule files applications are Prometheus and Loki.

The server currently supports two types of rules:

- standard alert rule files: directly passed to the corresponding applications
- templated alert rule files: Jinja2 templated rule file to render against specific devices

[571] http://ubuntu.com/observability
[572] https://charmhub.io/topics/canonical-observability-stack
[573] https://charmhub.io/cos-registration-server-k8s

The templated rule files are designed to allow the creation of an alert that will only affect a defined list of devices. The templated rule will be rendered for the devices that explicitly declared it in the `device-loki-alert-rule-files` or `device-prometheus-alert-rule-files` while registering on the `COS-registration-server` with the `COS-registration-agent`[574].

Templated alert rule files format

The templated rule files are `Jinja2`[575] templates. In the context of alert rules with COS, the Jinja2 start and stop variable are: `%%`.

The current supported variable is: `%%juju_device_uuid%%`.

Below is an example of a templated alert rule file:

```
groups:
  - name: low-memory/%%juju_device_uuid%%
    rules:
    - alert: 5GBLowMemory%%juju_device_uuid%%
      annotations:
        description: "Low memory alert specific to {{ $labels.device_instance }}"
        summary: "Robot {{ $labels.device_instance }} has less than 5 GB of memory free."
      expr: (node_memory_MemFree_bytes{device_instance="%%juju_device_uuid%%"})/1e9 < 5
      for: 5m
      labels:
        severity: critical
```

Note:

The name of the **group** as well as the name of the alert must be templated to ensure its uniqueness.

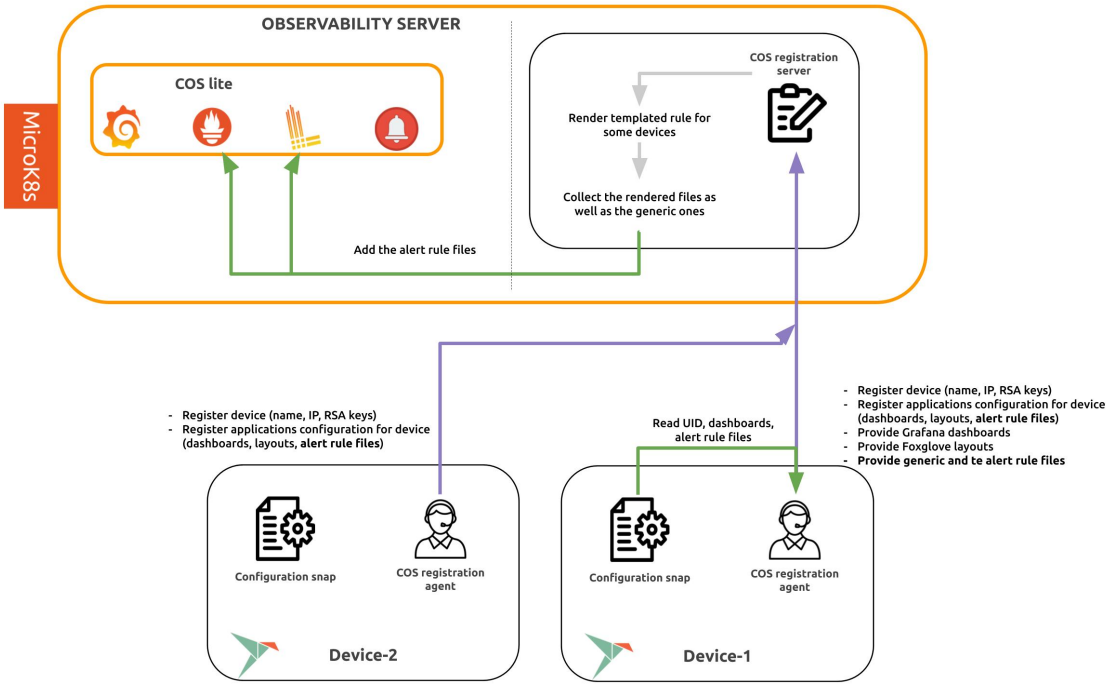
Alert rule files' flow from device to COS-registration-server and the applications

In the following diagram, we can see that the alert rule files distributed with the "Device-1" are getting uploaded to the `COS-registration-server` by the `COS-registration-agent`.

After that, the "Device-2" is also registering to the server and explicitly referring to the templated rule without having to upload it.

The `COS-registration-server` renders the templated files for the devices that specified them, depending on the type of alert rule file. It then sends the rendered files as well as the non-template ones to the various applications.

[574] https://snapcraft.io/cos-registration-agent
[575] https://jinja.palletsprojects.com/en/stable/



COS for robotics components

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

The COS for robotics is meant to monitor your devices. This means that we have a server hosting all our applications and devices sending data to our server.

The server side, based on COS Lite[576] is relying on Juju to orchestrate and MicroK8s to scale your applications. For the devices, all the components are snaps and can run on an Ubuntu machine or in the OS for robotics Ubuntu Core[577].

[576] https://charmhub.io/topics/canonical-observability-stack/editions/lite
[577] https://ubuntu.com/core/docs

Server components

The charmed operators that make up COS for robotics are available as the pre-configured COS for robotics bundle. COS for robotics is made up of the following Juju charmed operators:

- `cos-registration-server-k8s`[578]
- `foxglove-studio-k8s`[579]
- `ros2bag-fileserver-k8s`[580]

Additionally, you can package your own cloud application as a charm[581] and deploy it along COS for robotics.

`cos-registation-server-k8s`

The `cos-registration-server-k8s` operator is the first entry point for devices. With this server, devices are going to register themselves and upload their configurations. The `cos-registration-server-k8s` is then synching all the configurations to the appropriate applications (Grafana, Prometheus, Loki, etc.). Additionally, the operator offers a UI for the user to retrieve devices and the corresponding visualization.

The `cos-registration-server-k8s` is a charm for the `cos-registration-server`[582] working in complement with the `cos-registration-agent`[583].

`foxglove-studio-k8s`

The `foxglove-studio-k8s` operator is the charm of the former open-source version of Foxglove Studio[584]. The charm is meant to work with the `foxglove-bridge` snap[585]. The operator can be used to access live ROS data with the `foxglove-bridge` or to load bag files from the `ros2bag-fileserver-k8s`.

`ros2bag-fileserver-k8s`

The `ros2bag-fileserver-k8s` operator is used to store robotics data from devices. Robots are pushing data over SSH. The robotics data (ROS 2 bags), can latter be accessed with the file-server (Caddy[586]) exposed by the operator. Additionally, the file-server has a UI so you can access the files and their links to provide them to other applications (i.e. Foxglove Studio file entry). The charm is meant to work with the `ros2-exporter-agent`[587].

[578] https://charmhub.io/cos-registration-server-k8s
[579] https://charmhub.io/foxglove-studio-k8s
[580] https://charmhub.io/ros2bag-fileserver-k8s
[581] https://canonical-charmcraft.readthedocs-hosted.com/en/stable/tutorial/
[582] https://github.com/canonical/cos-registration-server
[583] https://snapcraft.io/cos-registration-agent
[584] https://foxglove.dev/
[585] https://snapcraft.io/foxglove-bridge
[586] https://caddyserver.com/docs/caddyfile/directives/file_server
[587] https://snapcraft.io/ros2-exporter-agent

Note:

If the space on your server is limited, make sure to clear periodically the data stored by the `ros2bag-fileserver-k8s`.

COS Lite components

COS for robotics is extending COS Lite and thus include its applications:

- `prometheus-k8s`[588]
- `alertmanager-k8s`[589]
- `loki-k8s`[590]
- `grafana-k8s`[591]
- `traefik-k8s`[592]
- `catalogue-k8s`[593]

Note:

More information about the COS Lite components can be found in the COS Lite documentation[594].

[594] https://charmhub.io/topics/canonical-observability-stack/editions/lite

Devices components

The snaps that make up COS for robotics are available on the Snap Store. COS for robotics is made up of the following snaps:

- `cos-registration-agent`[595]
- `foxglove-bridge`[596]
- `ros2-exporter-agent`[597]
- `rob-cos-data-sharing`[598]
- `rob-cos-grafana-agent`[599]
- `rob-cos-demo-configuration`[600]

[588] https://charmhub.io/prometheus-k8s
[589] https://charmhub.io/alertmanager-k8s
[590] https://charmhub.io/loki-k8s
[591] https://charmhub.io/grafana-k8s
[592] https://charmhub.io/traefik-k8s
[593] https://charmhub.io/catalogue-k8s
[595] https://snapcraft.io/cos-registration-agent
[596] https://snapcraft.io/foxglove-bridge
[597] https://snapcraft.io/ros2-exporter-agent
[598] https://snapcraft.io/rob-cos-data-sharing
[599] https://snapcraft.io/rob-cos-grafana-agent
[600] https://snapcraft.io/rob-cos-demo-configuration

Additionally, you can *package your own device application as a snap and deploy it along COS for robotics snaps* (page 2).

cos-registration-agent

The `cos-registration-agent` snap is the single component directly talking to the `cos-registration-server`. It's making sure the device configuration is propagated to the server. It's reading its configuration from the `rob-cos-demo-configuration`. Additionally, it exposes some data to the `rob-cos-data-sharing`.

foxglove-bridge

The `foxglove-bridge` snap is meant to directly communicate with the `foxglove-studio-k8s`. The snap is packaging the official `ros-foxglove-bridge`[601]. It is reading its configuration from the `rob-cos-demo-configuration`.

ros2-exporter-agent

The `ros2-exporter-agent` snap is recording ROS 2 bags and sending them to the `ros2bag-fileserver-k8s`. The snap takes care of recording bag, sending them to the server and then clean old ROS bags. It is reading its configuration from the `rob-cos-demo-configuration`. Additionally, the snap reads credentials from the `rob-cos-data-sharing`.

rob-cos-data-sharing

The `rob-cos-data-sharing` snap is an almost empty snap. It is simply used to share data between different snaps, from the `cos-registration-agent` to the `ros2-exporter-agent`. The data currently shared with the `rob-cos-data-sharing` are: a UID file as well as an SSH public and private key.

rob-cos-grafana-agent

The `rob-cos-grafana-agent` snap is packaging the official `grafana-agent`[602]. The snap is used to send data from the system as well as logs to different applications (Prometheus, Loki, etc.). The `grafana-agent` is configured in Flow mode[603]. It reads its configuration from the `rob-cos-demo-configuration`.

[601] https://github.com/foxglove/ros-foxglove-bridge
[602] https://grafana.com/docs/agent/latest/
[603] https://grafana.com/docs/agent/latest/flow/

`rob-cos-demo-configuration`

The `rob-cos-demo-configuration` snap is an example snap providing the configuration to all the COS for robotics snaps. The snap is meant to be used as a reference but could be used to try COS for robotics on your devices. You can find details about how-to write your own configuration file in the documentation: *Write configuration snap for COS for robotics* (page 143).

Security considerations for COS for robotics

Warning:

Beta Notice: COS for robotics is currently in `beta`. Content and features may change, and some functionality may be incomplete or experimental. Feedback is welcome as we continue to improve.

This page outlines the security considerations to keep in mind when operating COS for robotics, particularly when TLS encryption is enabled. Understanding these limitations and risks will help you make informed decisions when planning and maintaining your deployment.

Handle CA distribution carefully

The root CA certificate must be manually distributed to every device and operator laptop, as shown in the *Enable TLS encryption in COS for robotics* (page 159). Take care to transfer the CA file over a secure channel, verify its integrity before installing it, and never skip certificate validation to work around distribution issues — doing so defeats the purpose of TLS entirely. In large fleets or organisations with many operators, the overhead of maintaining this process securely is worth considering when choosing a TLS provider. Refer to the Security with X.509 certificates[604] topic on Charmhub for guidance.

Protect access to the Juju TLS model

The CA private key managed by the `self-signed-certificates` charm is the root of trust for the entire deployment. Any operator with write access to the `tls` Juju model can issue certificates that will be trusted by all registered devices. Apply the principle of least privilege to Juju user permissions and restrict access to the `tls` model to authorised operators only.

[604] https://charmhub.io/topics/security-with-x-509-certificates

Understand the scope of system-wide CA trust on devices

Installing the CA certificate into `/usr/local/share/ca-certificates/` and running `update-ca-certificates` extends trust to **every process** running on that device, not only the COS agents. Any certificate signed by this CA will be accepted as valid by the entire OS. This is the intended behaviour for this deployment, but it means the CA private key must be kept secure. If the CA is compromised, all devices that have installed it must be considered at risk.

Device leaf certificates are not automatically renewed

The `self-signed-certificates` charm handles automatic renewal of server-side certificates (Traefik, Grafana). Device leaf certificates, however, are issued at registration time and are not automatically renewed. By default, device leaf certificates are valid for 90 days. If a device leaf certificate expires, the device will need to re-register with the server to obtain a new one. Monitor certificate validity in long-running deployments, and consider adjusting the default validity period to suit your fleet's operational cycle.

There is no certificate revocation mechanism for devices

There is currently no certificate revocation list (CRL) or OCSP responder for device leaf certificates. If a device is decommissioned or suspected to be compromised, be aware of the following limitations:

- The registration server is a configuration and listing service only. Removing a device from it does not prevent a compromised device from continuing to publish data, as there is no identity-based access control on data ingestion.
- If the device's private key may have been exposed, the only way to invalidate all issued certificates is to rotate the CA using the `rotate-private-key`[605] action on the `self-signed-certificates` charm. This requires reinstalling the new CA certificate on every device and operator laptop, and re-registering all devices.

Device private key is protected by snap isolation, not encryption at rest

During registration, a private key and a CSR are generated on the device and the private key is stored in the `rob-cos-data-sharing` snap data directory. Snap data directories are owned by root and isolated between snaps through AppArmor and seccomp policies, which provides process-level protection. Physical access to the device remains a risk: consider enabling full-disk encryption on the robot's storage when the threat model requires it.

[605] https://charmhub.io/self-signed-certificates/actions

Foxglove bridge WebSocket is accessible on the local network

The Foxglove bridge listens for WSS (WebSocket over TLS) connections using the device leaf certificate. By default, this port may be reachable by any host on the local network. Use firewall rules to restrict access to the Foxglove bridge port to trusted hosts only.

Ubuntu Core devices cannot use system-wide CA distribution

The `update-ca-certificates` mechanism used in this guide works only on Ubuntu Desktop and Server. Devices running Ubuntu Core do not support this system-wide approach yet, and a suitable solution for Ubuntu Core is not yet available. Take this into account when planning your fleet's operating system.

Ubuntu gives robotics teams a complete, production-grade platform to package, deploy, and maintain robot software at scale.

The Canonical Robotics stack is your end-to-end infrastructure, from first build to global fleet. Ubuntu Core and Snaps give you a reliable, production-grade foundation for packaging and deploying ROS applications. COS for Robotics is your observability suite for monitoring robots in the field. And with ROS ESM, you get guaranteed security maintenance for your ROS environment, long after upstream end-of-life.

Robotics developers shouldn't have to become DevOps engineers. Our open source tools take this complexity off your plate, so you can focus on building great robots.

The Canonical Robotics stack is for robotics developers and integrators who need to ship robot applications reliably on solid open-source foundations.

5. In this documentation

- **First Steps:** *Canonical Robotics Stack overview* (page 212)
- **Packaging & distribution:** *Package a ROS application as a snap* (page 2) | *ROS architectures with snaps* (page 216) | *Deploy a robot with snaps and Ubuntu Core* (page 2) | *Migrate from Docker to snap* (page 109) | *Snapcraft plugins* (page 191) and *extensions* (page 191) for ROS
- **Observability & monitoring:** *Observability for robotics* (page 247) | *Monitor your robot fleet* (page 78) | *Alert rules configuration* (page 250) | *Deploy COS with TLS encryption* (page 159)
- **Security & long term support:** *Harden your robot* (page 176) | *What is ESM for ROS* (page 242) | *Security vulnerability audits* (page 245) | *Check if a CVE is fixed* (page 175)

5.1. How this documentation is organized

Tutorials **Step-by-step tutorials** for a hands-on introduction to Canonical's robotics solutions.	How-to guides **How-to guides** to achieve specific goals with Canonical's robotics stack.
Explanation **Explanations and clarifications** of the core concepts and key topics that underpin Canonical's robotics solutions.	Reference **Technical information** such as specifications, architecture, API documentation, and troubleshooting tips.

6. Project and community

The Canonical Robotics stack is part of the Ubuntu ecosystem. Its products are open source projects that warmly welcome community contributions, suggestions, fixes and constructive feedback.

- Our Code of Conduct[606]
- Community engagement commitment[607]
- Join the Robotics conversation on Ubuntu Discourse[608]
- Join the Snap(craft) forum[609]
- Interactive chat on Matrix.org[610]

Thinking about using the Canonical Robotics stack for your next project? Get in touch![611]

[606] https://ubuntu.com/community/ethos/code-of-conduct
[607] https://documentation.ubuntu.com/core/explanation/community-engagement/
[608] https://discourse.ubuntu.com/c/project/robotics/121
[609] https://forum.snapcraft.io/c/device/10
[610] https://ubuntu.com/community/communications/matrix
[611] https://ubuntu.com/robotics#get-in-touch