# Chapter 5
# The Robot Operating System (ROS1&2): Programming Paradigms and Deployment


**David St-Onge and Damith Herath**

## 5.1 Learning Objectives

The objective at the end of this chapter is to be able to:

- to know how to use (run and launch) ROS nodes and packages;
- to understand the messaging structure, including topics and services;
- to know about some of the core modules of ROS, including the Gazebo simulator, ROSbags, MoveIt! and the navigation stack.

## 5.2 Introduction

We expect most readers of this book to aim at the development of a new robot or at adapting one for specific tasks. As we mentioned in the introduction, the content of this book covers all of the required grounds to know "what has to be done" with an overview of several ways to address "how can it be done". If you do not know it already, you will quickly understand through this book that robot design calls to many different disciplines. The amount of knowledge needed to deploy a robotic system can sometimes feel overwhelming. However, many individual problems were solved already, including software ecosystems to simulate and then deploy our robots seamlessly. Advanced toolset and libraries are certainly integrated in the proprietary solution stack of the main robotic system manufacturers (such as ABB RobotStudio and DJI UAV simulator), but can everybody benefit of the last decades of public research for their own robots? This is a recurrent issue in many fields, and several libraries have been created in specific domains, such as to gather vision algorithms

D. St-Onge (✉)
Department of Mechanical Engineering, ÉTS Montréal, Montreal, Canada
e-mail: david.st-onge@etsmtl.ca

D. Herath
Collaborative Robotics Lab, University of Canberra, Canberra, Australia
e-mail: Damith.Herath@Canberra.edu.au



D. Herath and D. St-Onge (eds.), *Foundations of Robotics*,
https://doi.org/10.1007/978-981-19-1983-1_5

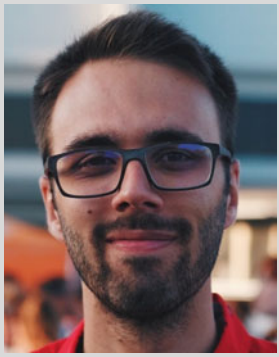
(OpenCV) and machine learning algorithms (TensorFlow). The Robot Operating System (ROS) is an open-source solution addressing this critical sharing need for robotic sensing, control, planning, simulation, and deployment. Not to be confused with a library, it is a software ecosystem (the concept of an operating system might be too strong) facilitating the integration, maintenance, and deployment of new functionalities and hardware from simulations to physical deployment. While ROS can run code the same from several popular languages, in order to use it you will need good knowledge of the infrastructure's underlying concepts (and honestly quite a bit of practice). ROS is renowned to have a steep learning curve and even more so for developers not familiar with software engineering. This chapter aims at giving you an overview of ROS and setting the bases to use it without being specific to any version (only few code examples are provided).

Since ROS is made to run predominantly on Linux operating system, we will end the chapter with a quick overview of Linux fundamental tools useful for roboticists and ROS developers.

**An Industry Perspective**

**Alexandre Vannobel, Team Lead, Kortex Applications Team**

Kinova inc.



I have a bachelor's degree in biomedical engineering from Polytechnique Montréal. I was especially interested in software development through my studies, most especially newer technologies such as AI, robotics, and cloud computing. I had the chance to work as an intern for one summer at Kinova. Needless to say, I learned a lot about robots during those four months I never really learned the basics of robotics in a classroom. It was more of a learn-by-doing experience (and it still is).

Learning the details and intricacies of ROS, Gazebo, and MoveIt was certainly a challenge! I have also been responsible for interfacing our robots with this framework, and there were some development and integration issues, as the goals and objectives of people who create robots and those who use robots do

sometimes differ. It is of importance in those cases to consider what users want and how they want to use the robot, but also to consider implementation costs and time of features.
I have witnessed the acceleration of ROS2's development in the last few months/years, and I think this is where the field is going. ROS1 is a centralized framework made to "unite" all of the robotics paradigms and tools in one big system, but it suffers from a lot of legacy design choices that make the industry really refractory from using it, starting with communication layers and the lack of real-time support. I think ROS2, which was designed with the same paradigms as ROS1 but with an emphasis on addressing those issues will bring the industrial and the research worlds closer.

## 5.3 Why ROS?

Before you dive into ROS usage, you must understand its roots, as they motivated several design decisions along the way, up to the need to redefine the whole ecosystem for industrial and decentralized applications in ROS2. ROS is a big part of the recent advances in robotics and its history is as important as any of the works presented in Chap. 1.

It all started with two PhDs students at Stanford: Eric Berger and Keenan Wyrobek. Early in their research, around 2005, they both needed a robotic platform to deploy and test their scientific contributions: the design of an intrinsically safe personal robot (Wyrobek et al., 2008). In their search for the best robotic platform, they ended up talking to several researchers, each developing their own hardware and software. The amount of duplicated work stunned them. They will later argue that 90% of the roboticists work involve re-writing code and building prototypes, as illustrated in Fig. 5.1. They made it their mission to change how things worked by developing a new common software stack and a versatile physical robot, the PR1. The fund-raising and marketing of their idea are out of scope here, but let us just mention that they had to work hard in order to gain some credibility (Wyrobek, 2017). While still at Stanford, they made the first PR1 prototype, alongside its modular software stack (inspired from *Switchyard* by Nate Koenig) and validated its versatility with a student coding competition and an in-house demonstration (a living room cleaning robot).

Berger and Wyrobek's vision of a universal operating system for robots definitely stroke right in the ambitious work of Scott Hassan (Silicon Valley billionaire). At the time, Scott Hassan was directing a research laboratory, Willow Garage, focused on autonomous vehicles. Over time, ROS (Willow Garage new name for Berger–Wyrobek–Koenig-inspired software stack) and PR became the main activity of Willow Garage, involving investments of several millions of dollars. These considerable resources clearly contributed to the rapid growth of ROS, namely by financially supporting a great team of engineers. However, there was already a hand-


**Fig. 5.1** Comic commissioned at Willow Garage, from Jorge Cham, to illustrate the wasted time in robotics R&D

ful of open-source projects for robotics at the time, including Player/Stage (Gerkey et al., 2003), the Carnegie Mellon Navigation Toolkit (CARMEN) (Montemerlo et al., 2003), Microsoft Robotics Studio (Jackson, 2007), OROCOS (Bruyninckx, 2001), YARP (Metta et al., 2006), and more recently the Lightweight Communications and Marshalling (LCM) (Huang et al., 2010), as well as other systems (Kramer and Scheutz, 2007). These systems provide common interfaces that allow code sharing and reuse, but did not survive as strong as ROS did. Money itself could not ensure ROS success, they needed a community.

> In Silicon Valley, people are in a secret place working on something that may or may not ever see the light of day. They may or may not ever be able to talk about it. It's a very different experience to be able to - as we do here - all day, every day, just write code and put it out in the world—Brian Gerkey, chief executive officer at Open Robotics (Huet, 2017)

The ROS community is nowadays clearly what makes ROS unique,[1] powerful, and impossible to avoid when working in robotics. Following Berker testimony (Wyrobek, 2017), they built that community over three strategies:

1. They secured the support of the other major players in open-source robotics by involving them from the start in the definition of what ROS must be. These people became early ambassadors of ROS.
2. They started a wide internship program, hosting PhD students, postdoctoral fellows, professors, and industry engineers from all over the world, all contributing to ROS and then using it in their own work. Berker mentions that Willow Garage was hosting at some point more interns than employees, counting hundreds of them.
3. They gave away 11 of their first PR2 prototypes, running exclusively on ROS, to major research laboratories around the world as the result of a competitive call. The new owners had to commit to contribute significantly to the ROS code base and to provide a proof that their institution allows them to share their research publicly.

Unfortunately, after Willow Garage skyrocketed ROS popularity and usage worldwide, the company was dissolved in 2013. It was never meant to be the end of ROS and PR2: the hardware customer service was taken over by Clearpath Robotics and the open-source software development by a new entity, the Open Source Robotics Foundation (non-profit). Under OSRF, they developed the first set of ROS distributions (*Distro*), from Medusa Hydro (2013) to Melodic Morenia (2018), but the foundation was growing with more requests for commercial contracts. In 2017, it splits to create the Open Source Robotics Corporation (known as Open Robotics,[2]) while the foundation still maintains the ROS code base. Open robotics released the last version of ROS1, Noetic, the first to be based on Python 3 (all previous versions used Python 2) and a whole new version, ROS2.

## 5.4 What Is ROS?

Now you may wonder if ROS is not just a glorified library. . . What is so special about it? The minimal answer is twofold: 1. It provides mechanisms for code maintenance and extensibility (adding new features), and 2. it **connects** a large community. The ROS wiki provides a more complete answer:[3]

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

[1] ROS users' world map: http://metrorobots.com/rosmap.html.

[2] https://www.osrfoundation.org/welcome-to-open-robotics/.

[3] http://wiki.ros.org/ROS/Introduction.

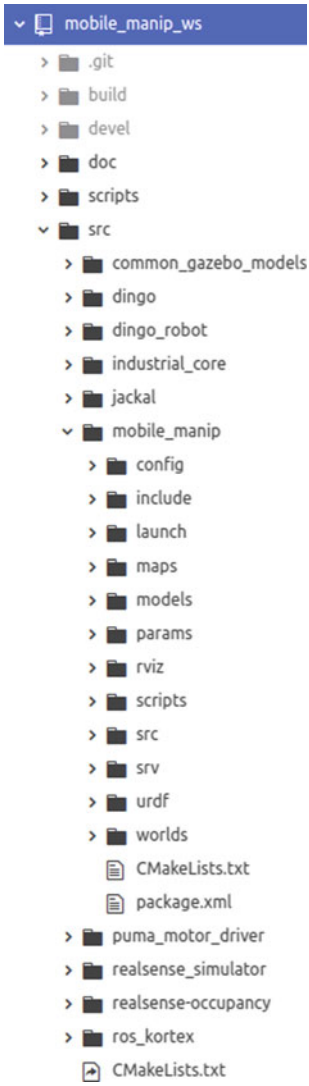



**Fig. 5.2** ROS workspace folder structure from the assignments detailed in Chap. 18

We will stick to our two-item list and just scratch the surface of some core concepts of software engineering to understand a bit better how they unfold in ROS. Implementing software engineering best practices is at the core of ROS, from a modular architecture to a full code-building workflow. The concept of an *operating system* may be a bit stretched as ROS is closer to a *middleware*: the abstract interface to the hardware (*POSIX of robots*).

As users (i.e., not developers) of ROS, we usually do not need to know the details of the building structure, but it is mandatory to learn the basics in order to know how

to properly use it. ROS provides a meta-builder, a uniform set of tools to build code in several languages for several different computer's environments and architecture. In ROS1, this is done by `catkin` (formerly by `rosbuild`); while in ROS2 `ament` takes over. In the end, they are really similar things, both just wrappers around *CMake* (Cross-platform Makefile system).[4] If you are a Python developer, you may think this kind of structure is unnecessary, but that is notwithstanding how it contributes to the portability and modularity of ROS. Portability here refers to the deployment of your code easily in different environments, as long as it follows the ROS building structure. Different environments can be for other users, new robots, but also in order to be seamlessly compatible with a testing environment. We will discuss more in details the simulation infrastructure provided by ROS in Sect. 5.6.3. Using the ROS build tools helps integrate your code with the rest of the ROS ecosystem. The meta-builder will generate the custom messages (topics), services, and actions your node requires (described in Sect. 5.5.1) and make them available to other executable (alike libraries). It will also add several paths and files to the environment in order to execute your code and quickly find your files (e.g., configuration files). The meta-builder will organize your work space over `build`, `devel` and `src` folders, as shown in Fig. 5.2.

Let us have a quick look at this ROS folder structure. In a glimpse, the `build` folder will host all the final files generated from the meta-builder while the `devel` folder keeps track of the files generated by the process for testing and debugging purpose. The `devel` folder will also include the essential `setup.bash` file, which, when sourced (`#source devel/setup.bash`), adds the location of the packages built to your ROS environment. Sourcing the system ROS (`#source /opt/<ROS Distro>/setup.bash`) and your local work space is mandatory to run any executable using ROS commands. This is usually part of any ROS installation procedures, both for maintained packages and third-party ones. The folder `src` is the one you will end up using the most. It contains a separated folder for each `package` of your work space. Software in ROS is organized in *packages*. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages is to provide their intended functionality in an easy-to-consume manner so that software can be easily reused.

Each of the package's folders must respect a structure, as shown in Fig. 5.2, with the subfolders: `include`, `launch`, `src` as well as optional ones related to the use of Python code (`script`) and simulation (`models`, `urdf`, `worlds`). The `include` and `src` folders are part of common C/C++ code structure, the first for headers (declarations) and the second for content (definitions). `launch` contains the launched files discussed in Sect. 5.5.2. The `src` folder contains one or more `nodes`. The nodes are executable with dedicated functionalities and specific inputs and outputs (when applicable). The work space shown in Fig. 5.2 is extracted from the assignments in Project Chap. 18. It combines third-party packages from Intel for the cameras (`realsense-occupancy`), from

[4] https://cmake.org/.

Kinova for the Gen3 lite arm (`ros_kortex`), from Clearpath for the wheeled base (`dingo`, `dingo_robot`, `jackal`, `puma_motor_driver`) and packages specific to the assignments (`mobile_manip`, `realsense_simulator`, `common_gazebo_models`).

To deploy a ROS work space, you must follow the ROS installation instructions,[5] and then either copy a third-party node (clone a Git repository) in order to work on it, or make your own fresh work space.[6] In both cases, you will end up writing code inside the package folder, for instance inside `mobile_manip_ws/src/mobile_manip` shown in Fig. 5.2. Inside of your package, if a node (an executable file in `script` or C/C++ code in `src`) is new, you need to add it to the `CMakelist.txt` building configuration file at the root of your work space for your meta-builder to be aware of the node existence. When setting up a new ROS environment, be aware that there is a compatibility matrix to fit each ROS distribution with Linux distributions.[7]

Now that we have a better idea of how the meta-builder works to provide portability (dealing with different environments) and modularity (packages and nodes), we can look into how modularity help **connects** the ROS community. ROS developers can share their nodes on any online platform (e.g., GitHub), or make it official by including it to a ROS distribution (indexed). A ROS indexed package must follow perfectly the ROS structure standard as well as programming best practices (unit tests, well commented, etc.). After a bit of training, it becomes easy to download, build, and run nodes made by any contributor around the world. This helped strengthen a community, one so enthusiast that it creates its own annual event, entitled *ROSCon* (*ROSWorld* in 2021 for the Virtual version), gathering hundreds of developers and users. ROS community is growing pretty fast, with new groups emerging, such as ROS Industrial[8] focused on developing industry-relevant capability in ROS. Where the community can easily exchange, their software must also be able to communicate. A large part of the modularity of ROS is provided by its communication infrastructure. A library of message types, extendable, guarantees the data format is compatible between all users nodes. The messages, i.e., simple data structures, can then be called in the form of topics or as part of services, as will be explained in Sect. 5.5.1.

### *5.4.1 ROS1&2: ROSCore Versus DDS*

ROS distributions are frequently released with major updates (enhancements). Since 2017, the core of ROS was revisited, leading to the release of a first stable ROS2 distribution in 2020, Foxy Fitzroy. The last distribution of ROS1, Noetic, will be

[5] https://wiki.ros.org/ROS/Installation.

[6] https://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment#Create_a_ROS_Workspace.

[7] https://www.ros.org/reps/rep-0003.html#platforms-by-distribution.

[8] https://rosindustrial.org/.

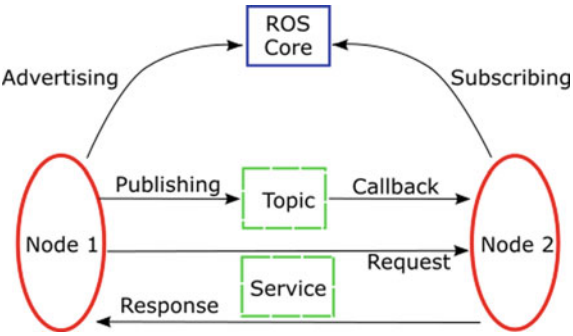



**Fig. 5.3** ROS Core role: the librarian connecting the nodes' topics and services

officially supported until May 2025 and may very well be active longer than that, but at some point all ROS users are expected to transit to ROS2. We quickly mentioned the new building mechanism of ROS2, `ament`, and we will discuss some format changes (e.g., launch files) in the upcoming sections, but the main difference is at the core, the `roscore`. In ROS1, `roscore` is a collection of nodes and programs that are prerequisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. Launching the `roscore` (either automatically with a launch file or manually with the `roscore` command) starts the ROS Core, i.e., the ROS1 librarian. As shown in Fig. 5.3, the ROS Master (i.e. ROS Core) is the one responsible for indexing all nodes running (the *slaves*) along with their communication modality. In other words, in ROS1, without the ROS Core, the nodes cannot be aware of the others, let alone start to communicate with one another. However, when all nodes are launched and aware of the others, theoretically the ROS Core could be killed without any node noticing.

At a glimpse: `roscore` is dead in ROS2, no more *master* and *slaves*. The communication infrastructure is fundamentally decentralized in ROS2, based on a peer-to-peer strategy, the Data Distribution Service (DDS). Where ROS1 had a critical single point of failure, no node can block the others from running in ROS2. DDS includes packet transport protocol and a distributed *discovery* service to grab information from the other running nodes.[9] This paves the way to facilitating the development and deployment of multi-robot systems, maybe even so-called robotic swarms.

Before getting into the ROS world, you need to pick your version. If you are looking for more existing packages and a more stable API, use ROS1. If you are

[9] For more information: https://design.ros2.org/articles/ros_on_dds.html.

looking for long-term stability, better performance, and newer algorithms, use ROS2. Just do not try to learn both from scratch! If you are still in doubt about which one to go for, ignore ROS1 and use ROS2, since ROS 1 will be going away in a couple of years.

### *5.4.2 ROS Industrial*

While we will be limiting our discussions to ROS 1 & ROS 2 in this book, it is worth noting that another flavor of ROS exists called ROS Industrial or ROS-I for short. [10] As the name suggests, ROS-I is a concerted effort to bring the best of ROS to industrial-scale robotics. While, in general, research robotics systems such as the PR2 follow an open-source ethos, most commercial robotic systems use closed and proprietary software. This makes it extremely difficult to develop cross-platform projects using them or adapt existing commercial hardware systems outside their intended ecosystems. Frustrated by this situation, Shaun Edwards, in 2012, created the initial ROS-I repository in collaboration with Yaskawa Motoman Robotics company and Willow Garage while he was at Southwest Research Institute to facilitate the adoption of ROS in manufacturing and automation. Since then, many commercial robotic platforms have been integrated within ROS-I. Core developments of ROS-I are independently managed through several industrial consortia that require a paid membership to participate. A good understanding of ROS should set you up for a relatively easy transition to ROS-I if you eventually venture into commercial robotics.

## 5.5 Key Features from the Core

The following sections will give an overview of the main features included in ROS. While the focus is on ROS1 (the assignments presented in Project Chap. 18 run on Noetic), the concepts are shared with ROS2, but some format differences are discussed when applicable.

### *5.5.1 Communication Protocols*

Whether it is decentralized (ROS2) or centralized (ROS1), the communication between nodes is structured in *messages*.[11] Figure 5.4 shows the `Odometry` mes-

[10] https://rosindustrial.org/.

[11] http://wiki.ros.org/Messages.

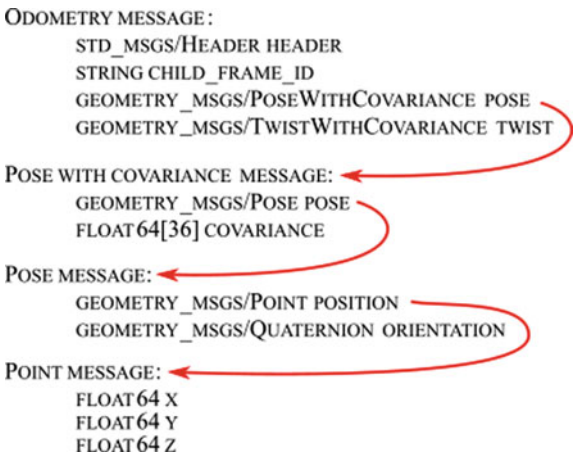



**Fig. 5.4** Content of ROS topic Odometry

sage with some of the message types it contains. Several message libraries come along with a ROS installation, but developers can also generate custom messages for their node. At run time, the availability of these data structure can be advertised over *topics*. Topics are barely names, i.e., labels, put on a given data structure (message) from a given node. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Topics are one of the main ways in which data is exchanged between nodes and therefore between different parts of the system (between robots and with a monitoring ground station). In order to share information, a node needs to *advertise* a topic and then *publish* content (messages) into it. The first part is done in the initialization part of the node's code, while the latter is done each time new data must be shared, commonly inside the code's main loop at a fixed frequency. On the other side, the node(s) that needs a topic's content will *subscribe* to it. The subscriber will associate a callback function triggered for each new incoming message.

ROS comes with a really handy debugging tool for topics, the terminal command `rostopic` (`ros2 topic` in ROS2). It can be used to show all available topics from the nodes running: `rostopic list` (`ros2 topic list`), to print the content (message) of a given topic: `rostopic echo odom` (`ros2 topic echo odom`) and to show the publishing frequency of a topic: `rostopic hz odom` (`ros2 topic hz odom`).

Topics are connectionless communication (classic publisher/subscriber system) in the sense that the publisher of the message does not know if any other node is listening. ROS also provides with a connection-oriented protocol (synchronous RPC calls), the *services*. Services have a client and a server, and both will acknowledge the information received by the other at each transaction. Topics and services use the same containers (message types) for information, but are better suited to different applications. For instance, topics are useful to stream the reading from a sensor, while services are better suited to share the configuration of a node or change a

```xml
<launch>
  <arg name="use_sim_time" default="true" />
  <arg name="gui" default="false" />
  <arg name="headless" default="true" />
  <arg name="world_name" default="$(find robot_manip)/worlds/jackal_maze.world" />

  <!-- Launch Gazebo with the specified world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="0" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
    <arg name="headless" value="$(arg headless)" />
    <arg name="world_name" value="$(arg world_name)" />
  </include>

  <!-- Spawn Jackal -->
  <include file="$(find robot_manip)/launch/include/spawn_jackal.launch" >
    <arg name="init_pose" value="-x 2.0 -y -4.0 -z 1.0" />
    <arg name="use_state_estimation" value="false" />
    <arg name="control_config" value="$(find robot_manip)/config/jackal_control_tf.yaml" />
  </include>

</launch>
```

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

**Fig. 5.5** Example of a launch file for: left is the XML format for ROS1 and right, the Python format new to ROS2

node's state. Finally, ROS provides the *actions* protocol (asynchronous RPC calls), combining topics and services. A basic action includes a goal service, a result service, and a feedback topic. Its format is well suited to interface with mission planners, such as QGroundControl.[12]

### *5.5.2 Launch and Run*

To deploy a ROS system means to start several executable files, i.e., *nodes*. The most basic command to do so is `rosrun <package name> <node name>` (`ros2 run <package name> <node name>`), which is most often run in a different terminal for each node. However, in ROS1 you need a roscore before any node can be run, so you must use the command `roscore` beforehand. Using this strategy to start the nodes individually will lead to numerous terminal tabs that must be monitored simultaneously. ROS provides another way to launch several nodes altogether: the launch files. In ROS1, using a launch file will also automatically start the `roscore`.

The format of the launch file differs between ROS1 and ROS2, as shown in Fig. 5.5. ROS1 uses an XML file while ROS2 encourages the use of Python scripts (ROS2 still supports XML format). Nevertheless, both serve to call nodes with parameters and can nest other launch files. Calling several nodes simultaneously is great, but what happens if you need twice the same node, for instance to process images from two cameras? You can always use the `rosrun` command to launch nodes afterward that are not in the launch file; they will connect to the same ecosystem automatically. However, a powerful feature of launch files is the `group` tag to force nodes into a given namespace: the same node can then be launched several times in different parallel namespaces without interfering with one another. This is essential to simulate multi-robot systems.

[12] http://qgroundcontrol.com/.

### 5.5.3 ROS Bags

Now say you developed a new collision avoidance algorithm, based on the data of several sensors. You deploy it on your robot and go for a run with it. No matter how well it goes, you will want to extract performance metrics and assess afterward the issues you faced. This calls for a logging system, luckily ROS provides a robust and versatile one out-of-the-box! The ROS bag format is a logging format for storing ROS messages in files. Files using this format are called bags and have the file extension `.bag`. Bags are recorded, played back, and generally manipulated by tools in the `rosbag` (`ros2 bag`) and `rqt_bag` (no counterpart yet available in ROS2) packages. You can replay your field experiments: republish all sensor data at their real frequency (or simulate different publishing rates), including packet loss or any disturbance from the experiment. `rosbags` also has an API that provides features to quickly parse and analyze or export your data. For instance in Python, it may look like:

```
import rosbag bag = rosbag.Bag('test.bag') for topic, msg, t
in bag.read_messages(topics='odom'):
    print("Odometry is x={}, y={} and z={} at time {} sec".
format(
        msg.pose.pose.position.x, msg.pose.pose.position.y,
msg.pose.pose.position.z,
        t.toSec())
bag.close()
```

Rosbags are key to tuning your algorithms and sharing your time-consuming experimental data with your peers.

### 5.5.4 Transforms and Visualization

Can you imagine a useful, physical robot that does not move or watch something else move? Any useful application in ROS will inevitably have some component that needs to monitor the position of a part, a robot link, or a tool. The ROS way of dealing with relative motion is encompassed in `TF` (transforms). `TF` allows seeking the geometrical transformation between any connected frames, even back through time. It allows you to ask questions like “What was the transform between A and B 10 seconds ago?"

One possible example is a camera, mounted on a robot, tracking markers in the scene, as shown in Fig. 5.6. This example shows the robot odometry frame (the mobile base motion—`camera_odom_frame`), the camera pose (fixed to the base—`camera_fisheye2_frame`), and the frame of the tag detected (`ETS_target`). The tag is detected in the `camera_fisheye2_frame`, and its pose is extracted and transformed directly in `camera_odom_frame` to visualize all frames together.

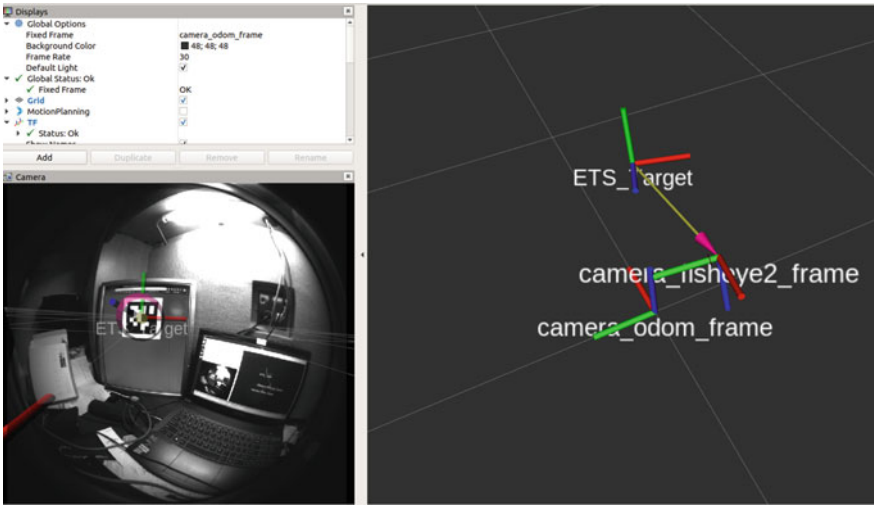



**Fig. 5.6** Visualization in RViz of a fish-eye camera feed and the reference frames resulting from a fiducial marker detection

As long as you have the position and orientation of an object (six degrees of freedom), you can broadcast its `TF` in ROS. For instance in Python, it may look like:

```
import tf2_ros # import the TF library br =
tf2_ros.TransformBroadcaster() # create de broadcaster t =
geometry_msgs.msg.TransformStamped() # create the message
container
# fill the message: t.header.stamp = rospy.Time.now()
t.header.frame_id = "world"
t.child_frame_id = "myrobotframe"
t.transform.translation.x = x
t.transform.translation.y = y
t.transform.translation.z = z
q = tf_conversions.transformations.quaternion_from_euler
(psi, phi, theta)
t.transform.rotation.x = q[0]
t.transform.rotation.y = q[1]
t.transform.rotation.z = q[2]
t.transform.rotation.w = q[3]
br.sendTransform(t) # broadcast the transform
```

Notice in the snippet above the format of the orientation (rotation): ROS, by default, requires to use quaternions. `tf_conversions` library provides the tool to convert rotation matrices and Euler angles to quaternions and back, but for more information about the mathematical representation of the quaternions, read Chap. 6. Often `TF` are used to define the fixed geometrical relations between a robot's parts. You can then

rather easily use the pose of an object detected by a camera mounted somewhere on your robot to feed the wheels motors with appropriate commands, such as "my camera sees the door 2 m ahead, but is positioned 50cm from the wheel axis, so let's go forward by only 1.5 m".

The viewer shown in Figs. 5.6, 5.7, and 5.8 is RViz, short for ROS Visualization. It is a 3D viewer supporting almost all types of topics, namely 2D and 3D LiDAR point clouds, camera stream, and dynamic reference frames motion. The viewer is launched simply with `rosrun rviz rviz` (or simply `rviz`). Then using the graphical interface *Add* button, you can select the topic you want to monitor. While RViz was made to monitor your robot's topics, it can also host *interactive markers* that can be moved in the visualization window and will broadcast their updated position out in ROS. An example used to command a robotic arm is shown in Fig. 5.8.

## 5.6 Additional Useful Features

Several community contributions went into the essential toolset of ROS and greatly contribute to its popularity. This section covers a handful of what we consider to be the most important for mobile robots and manipulators. All of these packages are leveraged in at least one of the assignments of Project Chap. 18.

### *5.6.1 ROS Perception and Hardware Drivers*

When dealing with your hardware integration, the same logic applies as for the software parts discussed previously: you do not want to waste time in reproducing what was done already to interface with each component. Manufacturers have their counterpart to this logic: it can be really expensive to develop drivers for several different operating systems and software solutions to accommodate potential clients. ROS acts here again as a standard, connecting the manufacturers to a large community. Hundreds of hardware manufacturers deliver ROS nodes with their products, namely SICK, Clearpath, Kinova, Velodyne, Bosch, and Intel. The driver node made by the manufacturer most often deals only with low-level communication into ROS compatible topics and services. From that point, the meta-package ROS perception helps with filtering, synchronizing, and visualization of the data. For instance, ROS perception includes `pcl_ros` to manage point clouds. It includes filters such as voxel grid filter and pass-through filter, but also geometrical segmentation of the data to extract planes or polygons from the point cloud. An example point cloud published as a ROS topic is shown in Fig. 5.7. For dealing with images (cameras), `cv_bridge` and several other packages bring the powerful features of the open library OpenCV to process images within ROS code. This provides the classic algorithms for contours detection, images filtering (blur, etc.), and histogram generation. From there, many

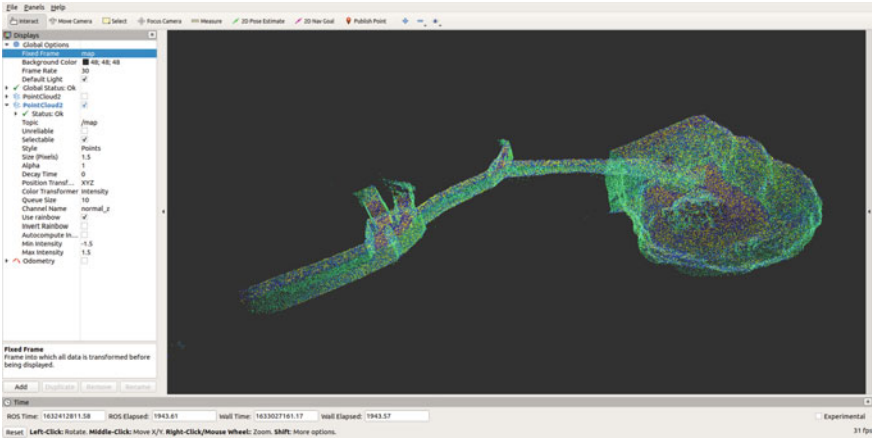


**Fig. 5.7** DARPA subterranean 2021 spot-1 finals map made by CTU-CRAS-NORLAB team

machine learning algorithms have ROS wrappers, such as the powerful You Only Look Once (YOLO)[13] for object recognition.

Finally, ROS perception also contains a package integrating several of the most up-to-date algorithms for simultaneously localization and mapping (SLAM), `gmapping`. Based on either on 2D LiDAR, 3D LiDAR, stereo camera, or a single camera, the package outputs a rough map of the environment explored by the robot without any a priori knowledge of the robot position in the map. These powerful algorithms are nowadays essential to any mobile robot deployment in GPS-denied environment. Several other, and more recent, SLAM solutions are also available on GitHub from research laboratories around the world, but `gmapping` is maintained by OSRF. When the environment is known (a 2D map is available), you may prefer to use the ROS package for adaptive Monte Carlo localization (AMCL). This one uses a particle filter to find the best candidates in position when simulating your laser scan from the map provided. This is the strategy deployed in the assessments 4 and 5 of Project Chap. 18.

### *5.6.2 ROS Navigation and MoveIt!*

Let us assume that the perception stack grants us with the position of the robot and a map of its environment. In order to fulfill any mission, the robot will need to move in this environment, either by finding an optimal trajectory (mobile robot) or by computing an optimal posture for a manipulator to reach a given pose with its tool (i.e., gripper). For mobile robots, conventional methods of indoor path planning

[13] https://github.com/leggedrobotics/darknet_ros.

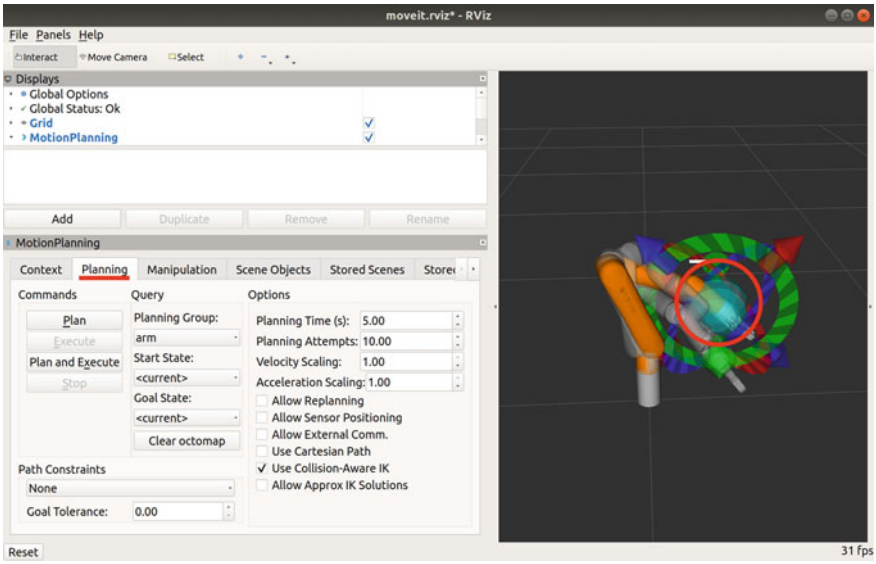



**Fig. 5.8** Kinova Gen3 lite manipulator controlled by interactive markers and MoveIt! planner from RViz

often refer to the optimal path as the shortest path that can be obtained from various algorithms such as A*, Dijkstra's (Palacz et al., 2019) or rapid-exploring random trees (RRT). These algorithms, and a lot more, are available out of the box from public ROS packages.

For manipulators, many numerical solvers for multibody dynamics have been proposed over the past decades and along with them path planners that either use sampling-based algorithms or optimization-based algorithms. These algorithms and several others were integrated in the Open Motion Planning Library,[14] itself integrated in the MoveIt! ROS planning package.[15]

Figure 5.8 shows MoveIt! in action through RViz using interactive markers. These markers can simply be dragged to the desired goal and then the left menu grants the user access to different planners and their configurable parameters. MoveIt! can also consider static objects in the scene to plan a solution considering collision avoidance. These objects can be added manually or imported from the Gazebo simulator.

### *5.6.3 Gazebo Simulator*

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train artificial intelligence systems using realistic scenarios. Gazebo

[14] https://ompl.kavrakilab.org/.

[15] https://moveit.ros.org/.

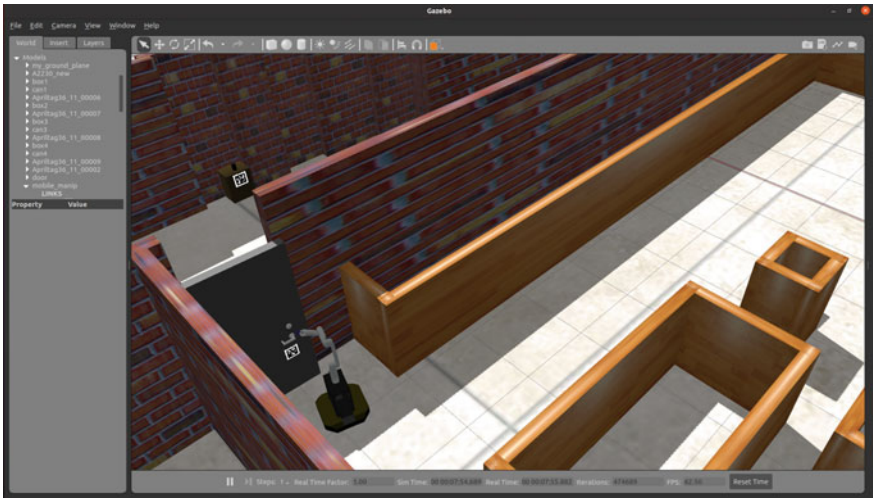


**Fig. 5.9** View from Gazebo simulator with the mobile manipulator of the assignment in Project Chap. 18

offers the ability to accurately and efficiently simulate robots in complex indoor and outdoor environments. It encompasses a robust physics engine, with convenient programmatic and graphical interfaces. Best of all, alike ROS, Gazebo is free, open source, and has a vibrant community.

Gazebo simulator can load any mesh in `obj` or `dae` format and then use it with realistic dynamics to simulate robot motion and collisions. Alike ROS, Gazebo is modular, so the simulation plugins for dynamics, can be customized as well as any sensor data. Several manufacturers provide plugins (e.g., Intel cameras) and models (e.g., Kinova robots) to simulate their hardware within Gazebo. Figure 5.9 shows a simulation environment from the Project Chap. 18, including Intel cameras, the fully actuated Kinova Gen3 lite manipulator, the differential drive Clearpath Dingo mobile base, and a world made out of walls, furniture, and functional doors.

Gazebo is by far the most popular simulator for ROS users, but it lacks realistic rendering and can be pretty heavy to run for a large number of robots (swarms). To address the first limitation, Gazebo is being phased out in favor of Ignition.[16] Nevertheless, developers in vision-based machine learning will prefer more realistic environments such as Unreal[17] and Unity[18] (which has a ROS plugin[19]). For the latter, swarm roboticists will use dedicated simulators, such as ARGoS[20] (which also has a ROS plugin[21]).

[16] https://ignitionrobotics.org/.

[17] https://www.unrealengine.com/.

[18] https://unity.com/.

[19] https://resources.unity.com/unitenow/onlinesessions/simulating-robots-with-ros-and-unity/.

[20] https://www.argos-sim.info/

[21] https://github.com/BOTSlab/argos_bridge/.

## 5.7 Linux for Robotics

We mentioned previously that ROS is not exactly an operating system, but rather a middleware. Still, many people are referring to it as the *Linux for robotics* (Wyrobek, 2017). There is some truth in this name, as ROS is extending the Linux operating system to robotic applications. Until ROS2, it was only able to run properly on Linux. It means that the majority of ROS users must know their way around in a Linux environment.

We will take for granted that you start on a computer already set up with Linux (Dell sells certified computers preloaded with Linux[22]) or that you know how to launch a Linux virtual machine in Windows or OSX (although virtual machines are not recommended for hardware experiments and computer-intense simulations).

As we mentioned earlier, when installing ROS on a Linux system, look into the ROS-Linux compatibility matrix first.[23] In all of Linux distributions, you will need to input some terminal commands to get things done. Knowing the basic commands in a Linux terminal is also rather essential for embedded development, as the most popular on-board computers (e.g., Raspberry Pi, NVidia) will run a version of Linux and can be accessed through a remote terminal session (e.g., ssh). The most essentials terminal commands are as follows:

- `cd`: Change Directory. `cd ..` is used to get to the parent directory.
- `ls`: List Files. `ls -la`: will list all files (hidden ones too) along with the properties (permissions and size).
- `mv`: MoVe file.
- `cp`: CoPy file.
- `rm`: ReMove file.
- `df`: Disk Filesystem (disk usage). `df -h` allows to see the memory usage on all disks in human readable format.
- `reset`: to remove all output from the terminal screen and remove any local environment variables changes.

To edit and compile your ROS code, you want an integrated development environment (IDE) that can help you find the right names and definitions of functions, as well as compile and even debug your code. IDEs are like glasses: you need to try them to find the one that fits you best. A lot of IDEs are available for Python (Atom, Eclipse, PyCharm, etc.) and C/C++ (Visual Code, CLion). Linux experts sometimes prefer the highly configurable text editors such as Sublime, Emacs, and Vim, for which plugins and tutorials are available for ROS. However, the majority of the ROS developers seems to prefer Eclipse, for its user-friendly interface, its support for several programming languages, and its ROS plugin seamlessly integrated. Other more recent options are drawing attention: Microsoft Visual Code, or its open-source ROS version, Roboware, and the web-based ROS Development Studio (RDS). While they

[22] https://www.dell.com/en-us/work/shop/overview/cp/linuxsystems/.

[23] https://www.ros.org/reps/rep-0003.html#platforms-by-distribution.

all have pros and cons, they also all do essentially the same thing. If you are looking for an IDE, we suggest VS Code. If you just want a code editor, we like Sublime Text.

## 5.8 Chapter Summary

This chapter introduced the Robotic Operating System, ROS. We first discussed the motivation for its conception by going through its origin and then we gave an overview of its core advantages, leading to its current popularity. The chapter covered both ROS1 and ROS2, with a short stopover on the centralized versus decentralized differences between them. We then covered the essential features from the ROS Core and third-party additions. Finally, we gave essential hints to new Linux users, as this operating system is still the best suited one for ROS development.

## 5.9 Revision Questions

**Question #1**
In ROS1, what is the result of the command `rosrun robot_manip dingo_control`?

1. It launches the `robot_manip` node of the `dingo_control` package, but a `roscore` must have been started beforehand.
2. It launches the `dingo_control` node of the `robot_manip` package, but a `roscore` must have been started beforehand.
3. It launches the `robot_manip` node of the `dingo_control` package and a `roscore` if none is present.
4. It launches the `dingo_control` node of the `robot_manip` package and a `roscore` if none is prese[]nt.

**Question #2**
Associate the following ROS concepts:

1. Topic
2. Service
3. Message

with their definition:

A A link created by a node to post information to those who *subscribe* to it.
B A standardized container for the exchange of information between nodes.
C A blocking communication that awaits the response of the called node.

**Question #3**
Is ROS1 a completely decentralized software ecosystem? Explain why.

**Question #4**
Give the relative path in the ROS workspace to a C++ node source file (`doit.cpp`) of a package named `realsense_occupancy`.

## 5.10 Further Reading

The best way to learn ROS is to play with it. ROS wiki[24] is a great place to start learning more about the core packages. ROS wiki also contains several basic tutorials to practice with topics, services, actions, and launch file either in C++ or in Python. If you are looking for an extension to this chapter, including more explanations on the functionalities of ROS, the open access online book of Jason M. O'Kane, *A Gentle Introduction to ROS*[25] is a perfect resource. For the one that prefers physical books, going in depth in all of the ROS components, along with detailed example, look into the book of Quigley, Gerkey, and Smart, *Programming Robots with ROS*. Unfortunately, there is still a lack of good books specific to ROS2, but the online official documentation is always a great resource.[26]

## References

Bruyninckx, H. (2001). Open robot control software: The orocos project. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, pp. 2523–2528, vol. 3. https://doi.org/10.1109/ROBOT.2001.933002

Gerkey, B. P., Vaughan, R. T., & Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317–323.

Huang, A. S., Olson, E., & Moore, D. C. (2010). LCM: lightweight communications and marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4057–4062. https://doi.org/10.1109/IROS.2010.5649358

Huet, E. (2017). *The not-so-secret code that powers robots around the globe*. Bloomberg The Quint.

Jackson, J. (2007). Microsoft robotics studio: A technical introduction. *IEEE Robotics Automation Magazine, 14*(4), 82–87. https://doi.org/10.1109/M-RA.2007.905745

Kramer, J., & Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. *Autonomous Robots, 22*(2), 101–132. https://doi.org/10.1007/s10514-006-9013-8

Metta, G., Fitzpatrick, P., & Natale, L. (2006). Yarp: Yet another robot platform. *International Journal of Advanced Robotic Systems, 3*(1), 8.

Montemerlo, M., Roy, N., & Thrun, S. (2003). Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, vol. 3, pp. 2436–2441. https://doi.org/10.1109/IROS.2003.1249235

[24] https://docs.ros.org/.

[25] https://www.cse.sc.edu/~jokane/agitr.

[26] such as https://docs.ros.org/en/rolling/.

Palacz, W., Ślusarczyk, G., Strug, B., & Grabska, E. (2019). Indoor robot navigation using graph models based on bim/ifc. In *International Conference on Artificial Intelligence and Soft Computing*, Springer, pp 654–665.

Wyrobek, K. (2017). The origin story of ros, the linux of robotics. In *IEEE Spectrum*.

Wyrobek, K. A., Berger, E. H., Van der Loos, H. M., & Salisbury, J. K. (2008). Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In *2008 IEEE International Conference on Robotics and Automation*, pp. 2165–2170. https://doi.org/10.1109/ROBOT.2008.4543527

**David St-Onge** (Ph.D., Mech. Eng.) is an Associate Professor in the Mechanical Engineering Department at the École de technologie supérieure and director of the INIT Robots Lab (initrobots.ca). David's research focuses on human-swarm collaboration more specifically with respect to operators' cognitive load and motion-based interactions. He has over 10 years' experience in the field of interactive media (structure, automatization and sensing) as workshop production director and as R&D engineer. He is an active member of national clusters centered on human-robot interaction (REPARTI) and art-science collaborations (Hexagram). He participates in national training programs for highly qualified personnel for drone services (UTILI), as well as for the deployment of industrial cobots (CoRoM). He led the team effort to present the first large-scale symbiotic integration of robotic art at the IEEE International Conference on Robotics and Automation (ICRA 2019).

**Damith Herath** is an associate professor in Robotics and Art at the University of Canberra. He is a multi-award winning entrepreneur and a roboticist with extensive experience leading multidisciplinary research teams on complex robotic integration, industrial, and research projects for over two decades. He founded Australia's first collaborative robotics start-up in 2011 and was named one of the most innovative young tech companies in Australia in 2014. Teams he led in 2015 and 2016 consecutively became finalists and, in 2016, a top-ten category winner in the coveted Amazon Robotics Challenge—an industry-focused competition among the robotics research elite. In addition, he has chaired several international workshops on Robots and Art and is the lead editor of the book "Robots and Art: Exploring an Unlikely Symbiosis"—the first significant work to feature leading roboticists and artists together in the field of Robotic Art.