CPMR 3rd Edition: Software infrastructure

Gregory Dudek and Michael Jenkin

March 24, 2023

There have been a number of efforts to construct "globally accepted" software to support autonomous systems including [2], but ROS has emerged as the standard middleware for robot systems. ROS exists in two primary branches, ROS 1 and ROS 2, with different versions in each branch. Many of the examples given in this text rely on a working version of ROS 2 Foxy Fitrzoy, and examples are provided showing it running natively in Ubuntu. This is not the only way of running ROS 2[1], but it is perhaps the most straightforward. Given the large installed base of ROS 1 environments, here and in the code base associated with this book we also provide much of the code tailored for ROS 1 Noetic running under Ubuntu. But the focus is on ROS 2.

Just as ROS standardized the way in which robot software infrastrcture was being developed, OpenCV[1] has standardized the way in which computer vision algorithms are developed and deployed. Bindings exist for ROS and OpenCV for a number of different languages. In this book we will concentrate on software running under Python 3. The following sections provide a very brief introduction to ROS 1 and ROS 2 and image processing/computer vision in OpenCV.

1 A very brief guide to ROS

ROS is a middleware, rather than an operating system. It provides a set of tools that simplify the task of writing asynchronous code modules that interact with each other, the user, and the underlying hardware infrastructure. The ROS installation process is relatively straightforward, and instructions on how to perform this installation can be found on `wiki.ros.com` for your specific operating system. This text assumes that a full desktop install has been performed.

In addition to installing the basic building blocks of ROS, the full install will also include Gazebo (`gazebosim.org`), a robot simulator that simulates the physics of a robot interacting with its environment, and rviz2, a 3D visualization tool bundled with ROS that provides a graphical representation of the state of the robot and its sensor. In ROS 1 Gazebo was more tightly integrated into the ROS ecosystem and the visualization tool is called rviz. In ROS 2 this tight integration between Gazebo and ROS is relaxed somewhat, which introduces a number of differences in the way in which Gazebo and ROS interact between the two versions.

From a software point of view, ROS is a collection of processes communicating through a message passing framework. Processes can be distributed over multiple computers provided that they are connected through a standard networking infrastructure. ROS communications are strongly typed and come in various flavours; messages, services, and actions. ROS processes are encapsulated as either nodes, which are bound to a single process in the OS, or in ROS 1 as nodelets, where multiple ROS nodelets are tied to a single underlying process in the OS.

[1] Here, and elsewhere in this text we will use the term ROS to refer to ROS 2 and will only refer to ROS 1 and ROS 2 when referring to the specific ROS version

rosbag	Interface to ros bag files.
roscd	A 'cd' that maps ros project names to the directory in the file system.
rosclean	Clean up log files created by ROS.
roscore	Starts up the ROS core stack.
rosrun	Run a ROS executable.
roslaunch	Launch a ROS launch file.
rosmsg	Interact ROS message (msg) definitions.
rosnode	Interact with a ROS node.
rosparam	Interct with the ROS parameter server.
rossrv	Interact with ROS service (srv) definitions.
rosservice	Interact with a ROS service.
rostopic	Interact with running ROS topics.

Table 1: Commonly used ROS 1 commands

ROS provides a number of supports for the construction of software to support complex robot system. This includes the use of parameters to simplify and organize passing parameters to the various processes, and the concept of a name space to avoid collisions between similar aspects of the system and to encourage code re-use. In terms of the implementation language, there exist ROS-bindings for a range of different languages, although C++ and Python are the most mature.

The ROS command line environment makes considerable use of properties of the shell. It is thus critical that the default ROS modifications to the actions of your shell be performed. This is normally accomplished by modifying your `.bashrc` (or whatever the appropriate shell rc file is for your environment) using templates provided as part of the ROS install process.

Associated with this text is collection of Python-based ROS 1 and ROS 2 packages. The package associated with this appendix provides a "hello world" point robot (a `block_robot`) that integrates into the standard way in which robots operate in ROS 1. The implementation is intentionally simple and straightforward, rather than utilizing "best practices" in terms of developing a large-scale working ROS-based robot. Some of the questions at the end of this chapter help to introduce these techniques. The sections below provide a very brief tutorial for ROS 1 and ROS 2.

2 ROS 1

Table 1 summarizes the most common ROS 1 tools and provides a short one-line description of their function. This list is not exhaustive.

The `cpmr_apb` package provided with the text has a very simple structure (see Listing 1) and follows the basic structure of a ROS 1 package. The package maps onto the standard tree structure with the root directory of the package containing files that provide information for the catkin build system and details about the package more generally. Sub-directories provide source files, launch files, script files, etc. There is great flexibility as to the naming of these directories and what goes in where, but commonly launch files are placed in the launch directory, C++ source files in the src directory, Python files in the script directory, and model files in the urdf or model directory.

In order to use ROS 1, one must instantiate the ROS 1 system. The `roscore` command does this. As is standard for many ROS 1 commands, the command does not fork a process and return control to the console, rather it is more common to run the command in the background.

Once ROS 1 is running, it is possible to query the ROS 1 environment as to the currently

```
cpmr_apb/
    CMakeLists.txt
    launch
    package.xml
    script
    urdf

cpmr_apb/launch
    gazebo.launch
    rviz.launch

cpmr_apb/script
    populate.py
    depopulate.py

cpmr_apb/urdf
    blockrobot.urdf
]
```

Listing 1: ROS 1 package structure

running nodes

```
rosnode list
```

and the current set of topics being published by those nodes

```
rostopic list
```

But in the spirit of "hello world"-like programs everywhere, we begin by building a very simple ROS environment to illustrate ROS' basic features.

ROS can communicate with simulated robots as well as real robots. In either case it can be useful to have a representation of the robot. ROS provides this through a URDF (Unified Robot Description Format) representation. This is essentially a XML description of the robot that is used by a number of different tools within ROS. Listing 2 provides the description of a very simple mobile robot, the `block_robot`.

As described in Listing 2 through 3, the block_robot consists of a small block with a small cylinder mounted on top of it. Many robots can be described in terms of rigid links (with some geometry) connected by joints (with some possibly limited range of motion). Within the urdf file these aspects are provided by the `<link>` and `<joint>` elements. Full details of the aspects of the urdf framework can be found in various ROS 1 texts (including [6]) as well as in various tutorials on `www.ros.org`. Many of the tags in Listing 2 can be found within `<gazebo>` tags. These elements are used by the `gazebo` simulator to provide a simulation of the robot in the environment. Assuming `roscore` is running, and the necessary environment variables have been properly set, then

```
roslaunch cpmr_apb gazebo.launch
```

will begin to simulate the robot in an empty world similar to the view seen in Figure 1. This command launched the `gazebo.launch` launch file in the `cpmr_apb` package. The viewing camera

```
<?xml version="1.0"?>
<robot name="block_robot">
  <material name="red"> <color rgba="1␣0␣0␣1"/> </material>
  <material name="green"> <color rgba="0␣1␣0␣1"/> </material>
  <link name="base_footprint"> </link>
  <gazebo reference="base_link"> <material>Gazebo/Green</material> </gazebo>
  <link name="base_link">
    <visual>
      <geometry> <box size="0.10␣0.10␣0.05" /> </geometry>
      <material name="green"/>
    </visual>
    <collision>
      <geometry> <box size="0.10␣0.10␣0.05" /> </geometry>
    </collision>
    <inertial>
      <mass value="4.5" />
      <inertia ixx="1.0e-3" iyy="1.0e-3" izz="7.5e-3"
               ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>

  <gazebo reference="laser_link">
    <material>Gazebo/Red</material>
  </gazebo>
  <link name="laser_link">
    <visual>
      <geometry> <cylinder length="0.05" radius="0.05" /> </geometry>
      <material name="red"/>
    </visual>
    <collision>
      <geometry> <cylinder length="0.05" radius="0.05" /> </geometry>
    </collision>
    <inertial>
      <mass value="0.01" />
      <inertia ixx="8.33e-6" iyy="8.33e-6" izz="1.25e-7"
               ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>
```

Listing 2: ROS 1 Block Robot URDF (Part I)

```
<joint name="base_joint" type="fixed">
  <origin xyz="0.0␣0.0␣0.025" rpy="0.0␣0.0␣0.0" />
  <parent link="base_footprint" />
  <child link="base_link" />
  <axis xyz="0␣0␣1" />
</joint>

<joint name="laser_joint" type="fixed">
  <origin xyz="0.0␣0.0␣0.05" rpy="0.0␣0.0␣0.0" />
  <parent link="base_link" />
  <child link="laser_link" />
  <axis xyz="0␣0␣1" />
</joint>

<gazebo reference="laser_link">
  <static>true</static>
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>10</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>360</samples>
          <resolution>1</resolution>
          <min_angle>-3.1415</min_angle>
          <max_angle>3.1240</max_angle>
        </horizontal>
      </scan>
```

Listing 3: ROS 1 Block Robot URDF (Part II)

```
      <range>
        <min>0.20</min>
        <max>10.0</max>
        <resolution>0.01</resolution>
      </range>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
      <topicName>scan</topicName>
      <frameName>laser_link</frameName>
    </plugin>
  </sensor>
</gazebo>

<gazebo>
  <plugin name="object_controller" filename="libgazebo_ros_planar_move.so">
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <odometryRate>20.0</odometryRate>
    <robotBaseFrame>base_footprint</robotBaseFrame>
  </plugin>
</gazebo>

</robot>
```

Listing 4: ROS 1 Block Robot URDF (Part III)

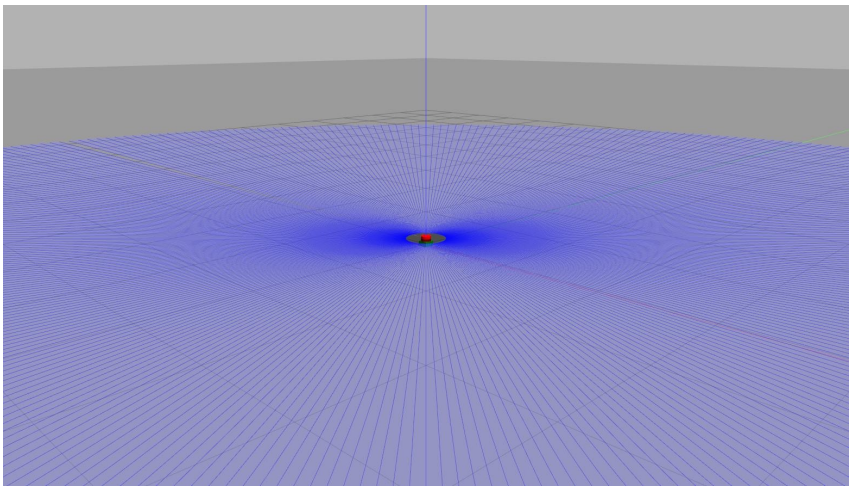


Figure 1: Gazebo simulation of the block_robot

can be moved around the environment, but the critical observations here are that the robot is simulated and drawn using the colours and shapes given in the robot's definition, as well as being equipped with a laser scanner as described in the urdf given in Listing 2-4.

A review of Listing 2 also reveals that for the various parts of the robot the mass and moment of inertia are specified. These parameters are used by `gazebo` in simulating forces acting on the robot, including gravity.

The urdf definition names the links and defines the joints that connect them. In this robot all of the joints are fixed. In other robots this might not be the case and joints can provide controlled motion between connected links. ROS 1 defines coordinate frames associated with each link, and uses transformations (links) to describe the relationship between these frames. This collection of frames defines how the robot goes together and its pose relative to the world. These transformation frames allow coordinates provided in one frame to be understood in another. ROS 1 provides a number of tools to review this structure. For example,

```
rosrun rqt_tf_tree rqt_tf_tree
```

will execute the ROS node `rqt_tf_tree` from the `rqt_tf_tree` package producing the tf tree shown in Figure 2.

Note that the `odom` to `base_footprint` transformation is broadcast by `gazebo` while the other transformations are broadcast by `rsp`, and that all of the transformations are connected. `gazebo` is now properly simulating the robot and its sensors. There are a number of nodes running to make this happen, and running `rosnode list` provides

```
/gazebo
/gazebo_gui
/rosout
/rsp
```

and running `rostopic list` shows that these nodes are providing a number of messages throughout the ROS environment

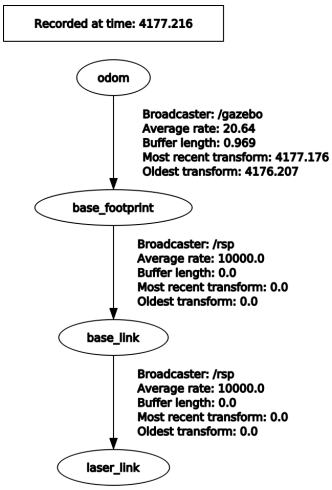


Figure 2: Transformation frame tree for the block_robot

```
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/odom
/rosout
/rosout_agg
/scan
/tf
/tf_static
```

One can examine the structure of these messages using, for example, `rostopic info /tf_static` will show that `/tf_static` is of type `tf2_msgs/TFMessage` and `rosmsg info tf2_msgs/TFMessage` has the structure shown in Listing 5 which describes the transformation between two frames.

Given that we have a simulator of the robot we should be able to move the simulated robot around. `gazebo` listens for `cmd_vel` messages which provide a translational and rotational velocity to the robot and then utilizes its physics engine to move the robot. There exist many tools to generate such messages, The node `teleop_twist_keyboard.py` in the `teleop_twist_keyboard` package provides the necessary `cmd_vel` messages based on keyboard inputs. You should be able to drive the robot around using the keyboard. That the `teleop_twist_kayboard.py` node is properly communicating with the `gazebo` node by sending `cmd_vel` messages can be confirmed using the `rqt_graph` tool using which produces the output shown in Figure 3.

`gazebo` simulates properties that would normally be found on a real robot, and if the robot simulation is properly assembled it will also tell you what a real robot would actually sense in its

```
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

Listing 5: A time-stamped transformation message

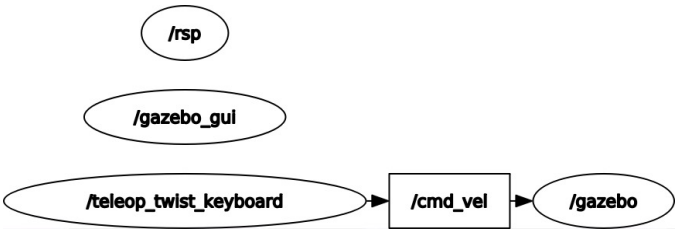


Figure 3: Rosgraph view of the teleoperation of block_robot

environment. The command

`rostopic echo /odom`

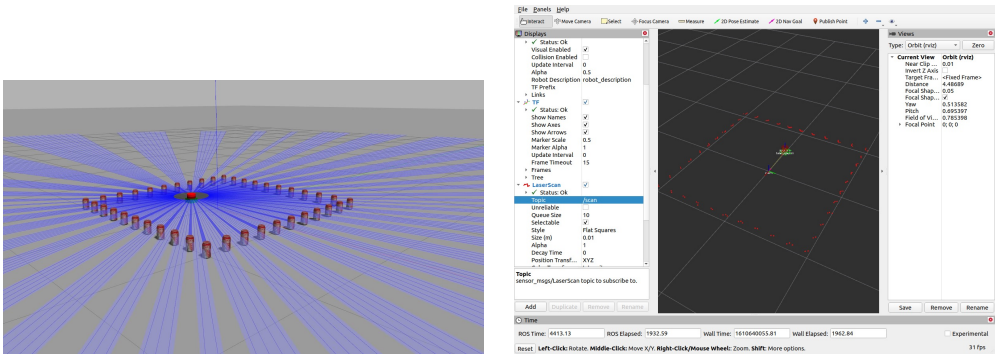
will show `gazebo's` simulation of the block_robot's pose as a function of time, and

`rostopic echo /scan`

will provide the current readings from the laser attached to the robot. There is a launch file in the `cpmr_apb` package called `rviz.launch`. When run, rviz provides an interactive window similar to the one shown in Figure 4(b). rviz needs to know the name of the fixed frame within which to display information. For this example, the appropriate frame is the `odom` frame. By default `rviz` does not display all of the sensor data available to the robot. The `block_robot` robot has a laser whose information is published under the topic `scan`. Use the `Add` button to display a laser scan and have this laser scan link to the `scan` topic. Valid laser returns from the simulated laser will now be shown in `rviz`.

The `cpmr_apb` package contains two ROS node definitions, `populate.py` and `depopulate.py`. To run the ROS `populate.py` node, you would execute

`rosrun mrcup_apb populate.py`



(a) gazebo view (b) rviz view

Figure 4: gazebo and rviz views of a sample environment

catkin_init_workspace	mkdir ~catkin_ws/src -p
	cd ~/catkin_ws/src
	catkin_init_workspace
catkin_create_pkg	cd ~/catkin_ws/src
	catkin_create_pkg package_name std_msgs rospy
catkin_make	cd ~/catkin_ws
	catkin_make
	source devel/setup.bash

Table 2: Sample catkin commands and their normal usage pattern for Python development

This command adds fourty soda cans to the large flat plane that defines the robot's environment. The corresponding ROS node `depopulate.py` removes the soda cans. These are not particularly representative examples of ROS nodes written in Python. See more complete and complex examples in the main chapters of the text. The populated world, as viewed in `gazebo` and `rviz` are shown in Figure 4. Note that for these two nodes to function, the `coke_can` model must be properly installed in Gazebo.

2.0.1 ROS 1 development infrastructure and build tools

ROS 1 relies on the `catkin` build system. There are a number of key commands in the build system and these are summarized in Table 2.

The catkin build system is somewhat more complex for C++ development, but Table 2 summarizes the commands necessary for `Python` development. `catkin_init_workspace` is required once to create the workspace. It requires a directory that will hold the catkin workspace with a src sub-directory to hold the packages being built. Creating a new package within the catkin environment is accomplished with `catkin_create_pkg`. This command requires the package name and the set of top level dependencies upon which this package relies. For `Python` development `rospy` and `std_msgs` are required. For more complex packages there may be other dependencies. Finally, once a new package has been put together it needs to be made with `catkin_make`. Although this would seem to be an unneccessary operation for Python development, it is required to build the

ros2 msg	Interface to ROS 2 messages.
ros2 launch	Launch a ROS 2 launch file
ros2 node	Interface to ROS 2 nodes.
ros2 run	Run a ROS 2 node.
ros2 service	Interface to ROS 2 services.
ros2 topic	Interface to ROS 2 topics.
ros2 pkg	Interface to ROS 2 packages.
ros2 action	Interface to ROS-2 actions.

Table 3: Commonly used ROS 2 commands

necessary representations upon which the `Python` code relies. If this is the first time the package has been made, it is also necessary to re-initialize the environment variables associated with this catkin workspace.

ROS 1 nodes written in Python must be executable (have their execute bit set) and must be written so as to start with a #! or **shebang**. The node will be executed by the shell and should start with

```
#!/bin/env/python3
```

There are two very simple nodes given in the `cpmr_apb` package.

ROS 1 distributed over multiple machines Ensuring network connectivity across multiple machines is critical. Every machine must be able to *see* the single `rosmaster`. If nodes cannot, then things will certainly not work as intended. ROS relies on an underlying network infrastructure. ROS relies on environment variables `ROS_IP` (the IP address of the local machine) and `ROS_MASTER_URI` (the IP address of the machine upon which `roscore` is running. A common problem, especially if running on a machine not connected to a name server, is coherence and consistency of these values. Actually giving these values explicit IP addresses within the network is often a good approach. Standard Unix networking tools (e.g., `ping`) can be very effective in debugging ROS-related networking issues.

3 ROS 2

ROS 2 seeks to build on the successes of ROS 1 while addressing some of the less successful design decisions in its predecessor. The basic structure is unchanged in that ROS 2 provides a middleware in which processes are distributed over some local network with a lightweight message passing infrastructure connecting the nodes. ROS 2 provides support for multiple different network passing infrastructures, although to date very few are in wide usage.

Whereas in ROS 1 it was necessary to start the ROS service infrastructure explicitly, in ROS 2 this is now handled implicitly so there is no equivalent `roscore` in ROS 2. Rather than having a collection of command-line tools to interact with the ROS framework, as in ROS 1. In ROS 2 the set of command-line tools is (generally) replaced with a single tool with multiple options. Table 3 summarizes the commonly used command versions

Each of the commands takes a collection of arguments. So, for example to publish "Hello World" on the /hello topic, one could execute

```
ros2 topic pub /hello std_msgs/String "data: Hello World"
```

ROS 2 package structure is slightly different but quite similar to, the package structure used in ROS 1. ROS 2 package structure varies slightly depending on the language in which nodes within the package are implemented. Here we concentrate on nodes implemented in Python. The minimal package structure for a ROS 2 package `mypackage` includes

- `package.xml` basic information and required dependencies for the package.
- `setup.py` defines mappings between console scripts and files and describes other files that must be created in the build directory.
- `setup.cfg` defines install locations.
- `test/mypackage` defines the testing framework.
- `mypackage` defines a directory for the Python files that make up the node.

Listing 6 shows the structure of the `cpmr_apb` package.

```
cpmr_apb/
    cpmr_apb
    launch
    package.xml
    resource
    setup.cfg
    setup.py
    test
    urdf

cpmr_apb/cprm_apmb
    depopulate.py
    __init__.py
    populate.py

cpmr_apb/launch
    gazebo.blockrobot.launch.py

cpmr_apb/resource
    cpmr_apb

cpmr_apb/test
    test_copyright.py
    test_flake8.py
    test_pep257.py

cpmr_apb/urdf
    blockrobot.urdf
```

Listing 6: ROS 2 package structure

Note that many packages will contain other files and directories. The `ros2` tool can be used to create an empty package using the `ament_python` build system

```
ros2 pkg create mypackage --build-type ament_python
```

In ROS 2 the source tree and install tree are separate. Thus even Python ROS 2 packages must be fully built to use them. ROS 2 uses the `colcon` build tool to build the nodes in the packages into the source tree. Executing

```
colcon build
```

The ROS 2 packages in the `src` directory will be built into to the `install` directory. The `ROS 2` search path can be extended to include the newly-built packages using

```
source install/setup.bash
```

for the `bash` shell. It is also possible to build only part of the `src` directory and to use symbolic links from the build directory to the source directory rather than copying the files. This approach is particularly useful in Python builds. For example

```
colcon build --symlink-install --packages-select mypackage
```

will only build the package `mypackage` and will install using symbolic links rather than copies.

The `github` repository associated with this text contain a number of packages to support the material presented in the text. If you clone the `github` repository to some directory (e.g., `cpmr_ros2`), and then in that directory execute the `colcon build` command above, this will build all of the code provided with the text. Source the appropriate `setup` file for your shell, and you should have access to the packages provided with the text.

We begin by building a very simple ROS 2 environment to illustrate ROS 2' basic features. (This environment is identical to the ROS 1 environment described above for ROS 1.) ROS 2 can communicate with simulated robots as well as real robots. In either case it can be useful to have a representation of the robot. As in ROS 1, ROS 2 provides this through a URDF (Unified Robot Description Format) representation. This is essentially a XML description of the robot that is used by a number of different tools within ROS 2. As in ROS 1, `gazebo` provides 3D simulation and rendering for robots and environments with links to `ROS`. Unlike in `ROS 1`, in `ROS 2` there is a stronger separation between `gazebo` and `ROS` than is found in `ROS 1`.

Listing 7-8 provides the description of the block robot. This is the ROS 2 version of the robot described in Listing 3-4. The basic structure of the URDF description of the robot is unchanged from ROS 1, although there are minor differences in the gazebo plugin structure.

The `gazebo` view and the `rviz2` view are very similar to the `gazebo` and `rviz` views shown in Figure 4. To launch the gazebo simulation of the `block_robot` in ROS 2

```
ros2 launch cpmr_apb gazebo.launch.py
```

will bring up the simulated block robot. To view the sensor values one uses

```
rviz2
```

This has much of the same functionality of `rviz` in ROS 1.

As in the ROS 1 version of this robot, it is possible to teleoperate the robot using standard teleoperational tools

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard.py
```

and to view the nodes, messages and transformation tree hierarchy using `ros2 node list`, and to view the message passing structure using `rqt_graph`. This produces a somewhat more complex graph structure than in ROS 1 as shown in Figure 5. Obtaining the transformation tree is through `ros2 run tf2_tools view_frames.py` and produces an output very similar to that found in Figure 2.

```
<?xml version="1.0"?>
<robot name="block_robot">
  <material name="red">
    <color rgba="1␣0␣0␣1"/>
  </material>
  <material name="green">
    <color rgba="0␣1␣0␣1"/>
  </material>

  <link name="base_footprint">
  </link>

  <gazebo reference="base_link">
    <material>Gazebo/Green</material>
  </gazebo>
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.10␣0.10␣0.05" />
      </geometry>
      <material name="green"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.10␣0.10␣0.05" />
      </geometry>
    </collision>
    <inertial>
      <mass value="4.5" />
      <inertia ixx="4.6875e-3" iyy="4.6875e-3" izz="7.5e-3" ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>

  <gazebo reference="laser_link">
    <material>Gazebo/Red</material>
  </gazebo>
  <link name="laser_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.05" />
      </geometry>
      <material name="red"/>
    </visual>
```

Listing 7: ROS 2 Block Robot URDF (Part I)

```
  <collision>
    <geometry>
      <cylinder length="0.05" radius="0.05" />
    </geometry>
  </collision>
  <inertial>
    <mass value="0.01" />
    <inertia ixx="8.33e-6" iyy="8.33e-6" izz="1.25e-7" ixy="0" ixz="0" iyz="0" />
  </inertial>
</link>
<joint name="base_joint" type="fixed">
  <origin xyz="0.0␣0.0␣0.025" rpy="0.0␣0.0␣0.0" />
  <parent link="base_footprint" />
  <child link="base_link" />
  <axis xyz="0␣0␣1" />
</joint>

<joint name="laser_joint" type="fixed">
  <origin xyz="0.0␣0.0␣0.05" rpy="0.0␣0.0␣0.0" />
  <parent link="base_link" />
  <child link="laser_link" />
  <axis xyz="0␣0␣1" />
</joint>

<gazebo reference="laser_link">
  <static>true</static>
  <sensor type="ray" name="laser_controller">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>10</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>360</samples>
          <resolution>1</resolution>
          <min_angle>-3.1415</min_angle>
          <max_angle>3.1240</max_angle>
        </horizontal>
      </scan>
```

Listing 8: ROS 2 Block Robot URDF (Part II)

```
      <range>
        <min>0.20</min>
        <max>10.0</max>
        <resolution>0.01</resolution>
      </range>
    </ray>
    <plugin name="laser_controller" filename="libgazebo_ros_ray_sensor.so">
      <ros>
        <namespace>/</namespace>
        <remapping>~/out:=scan</remapping>
      </ros>
      <frameName>laser_link</frameName>
      <output_type>sensor_msgs/LaserScan</output_type>
    </plugin>
  </sensor>
</gazebo>

<gazebo>
  <plugin name="object_controller" filename="libgazebo_ros_planar_move.so">
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <odometryRate>20.0</odometryRate>
    <robotBaseFrame>base_footprint</robotBaseFrame>
  </plugin>
</gazebo>

</robot>
```

Listing 9: ROS 2 Block Robot URDF (Part III)

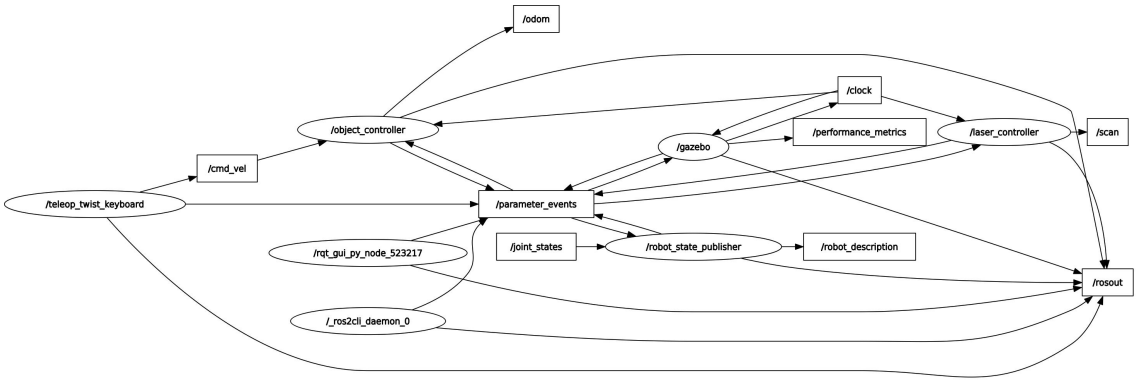


Figure 5: ROS 2 graph structure

4 Nodes, publishers, subscribers, services, actions

In both ROS 1 and ROS 2 nodes are processes that operate in parallel and communicate through a typed message passing framework. A node that generates messages is a publisher. Messages are published on a given topic. A node that receives messages is a subscriber. Messages are broadcast and there can be more than one subscriber for a given message topic. In addition to messages, nodes can provide services which operate using a request/reply paradigm. Services are also typed. ROS also supports an `actionlib` structure that provides a service-like communication structure but which support preemption.

Debugging the ROS infrastructure Given the multi-process, multi-machine asynchronous structure of ROS programs it is not uncommon to identify problems (bugs) when developing ROS code. There is no good set of rules to dealing with bugs in ROS code, but a few observations/recommendations are provided below.

ROS messages not being received by the appropriate node Inconsistent names and name spaces are a common problem in terms of ROS environments. The `rqt` package provides a number of effective tools to (for example) display the node graph which will show the state of message passing in the network.

World and robot parts not appearing where they should be The transformation framework graph should match expectations. If it does not, then things are unlikely to be performing nominally.

ROS log files ROS maintains a very effective logging mechanism, and many bugs can be identified through the logs. That being said, it is important to clear out log files when they are no longer needed.

Nodes not alive The `rosnode` command has many useful options. The `ping` option will ping a given node or collection of nodes to see if this node is responsive. The `list` option will see if the node has actually died.

Networking issues ROS assumes that the network infrastructure is sufficient for the messages being sent to it. For large data items (e.g., video, dense 3D point clouds, etc.) it can be easy to overwhelm the available network infrastructure. This is especially true if the network relies on wireless communications.

5 OpenCV

OpenCV is the Open Source Computer Vision library. It has emerged as a standard mechanism for writing image processing and computer vision code. It has bindings for a number of different languages. Here we will concentrate on bindings for Python 3.

In a digital computer, an image is typically a rectangular collection of pixels. Each pixel has a given location (row, col) where row and col are non-negative integers, and by convention $(0, 0)$ corresponds to the upper left corner of the image. Each location in the image has a type, typically either a floating point value between 0 and 1 or an unsigned 8 bit integer having values 0 (lowest) to 255 (highest) value. A given pixel may have only a single value – this is the case for greyscale

```
import cv2

cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    edges = cv2.Canny(gray, 100, 2)
    cv2.imshow('Edges', edges)
    cv2.waitKey(1)
```

Listing 10: An OpenCV python program to compute Canny edges from the default video camera.

images in which 0 would correspond to a black pixel and 255 would correspond to a white pixel, or multiple values. The most common multiple valued image is an image with three channels (blue, green and red). In Python, OpenCV leverages the numpy library to represent these images. That is, to declare a 480 row by 640 column colour image in OpenCV, and to set all of the values to 0 (that is, to make it black), one could use

```
img = np.zeros((480, 640, 3), np.uint8)
```

assuming numpy has been imported as np, which is typical. Leveraging Numpy provides coders access to a wide range of useful tools to manipulate images.

OpenCV provides mechanisms to load images, save images and display images. Assuming OpenCV is imported as cv2, and assuming the file 'test.jpg', exists in the current directory

```
image = cv2.imread('test.jpg')
cv2.imshow("image", image)
cv2.waitKey(0)
```

will bring up a window showing the image. The `waitKey` function will wait until a 'q' is typed on the window if its argument is 0, or wait that number of milliseconds.

OpenCV has a number of functions to read from image streams such as cameras or video files, write image streams to a video file, and so on. Implementations exist for a great number of standard image processing tools. Standard texts on OpenCV as well as online tutorials describe many of them. Space does not permit a review of all of the tools available in OpenCV, but as a simple example consider the small code snippet shown in Listing 10

This code captures frames from the default video camera attached to the computer. Each frame is then converted to a greyscale image and then the Canny edge detector is applied to it.

6 Further reading

There exist a number of good books dedicated to ROS 1, see [6, 4, 5, 3], for example. Much of the documentation of ROS 2 must be found online. ROS has a large on-line repository of tutorials and documentation sites. Many are located under `wiki.ros.org`. You may find it useful to peruse the nodes provided with this text.

7 Problems

1. The `block_robot` URDF description is written in raw URDF. A much more common approach in larger robot projects is to write the definition exploiting the `xacro` macro package provided with ROS. This allows the maintainer of a robot's description to not have to precompute appropriate offsets between links and other values but rather to provide a cleaner and clearer description of the robot. Full details of `xacro` can be found on the ROS wiki server `wiki.ros.org`. Refactor the `block_robot` so as to exploit `xacro` and modify the two launch files so that the robot definition is first run through `xacro` before being loaded into `gazebo` and `rviz, rviz2`. You can find examples of `xacro` for both ROS 1 and ROS 2 in the code repository associated with this text.

2. The `block_robot` is a very unadorned robot. Using either the solid primitives defined in URDF or through the addition of mesh objects created with some 3D modelling software, provide a hat to the top of the robot. Ideally this hat provides a hint as to the "front" of the robot.

3. Obtain a URDF description of some other robot – many robot manufacturers including Clearpath (mobile robots) and Kinova (manipulator robots) provide URDF descriptions of their products. Download such a description and place it into the empty world and drive it around the space using the keyboard teleoperation process described above.

4. There exist a range of other teleoperational inputs (e.g., joysticks) that are well supported in ROS. Drive the robot around with one. You will likely have to customize the standard `ROS` nodes to make this work (well) with your joystick.

5. In the examples provided here a very simple world environment is used. Create a more sophisticated world model and test the `block_robot` within it. World models can be created in `gazebo` directly and `gazebo` comes with access to a large number of object models (e.g., the soft drink can used in the example above) that can be added to the word. But a more rewarding process is to model some real environment that will be used with a real robot. A number of free tools exist online to build components for such models, including tools such as `Google Sketchup` to model complex environments.

6. Add a camera to the `block_robot` mounted on the top of the laser sensor facing along the x axis of the robot and parallel to the robot's base frame. The process of adding a camera is very similar to the LIDAR sensor provided in Listing 1, but with the Gazebo `camera` plugin. Examples of the appropriate XML to define such a sensor can be found in tutorials on `gazebosim.org`. See also the robots described in other chapters of this text.

7. In order to not overly complicate the `block_robot`, the robot is created independent of a namespace and it publishes and subscribes to topics at the highest level of the name hierarchy (.e.g, `scan` and `cmd_vel`). Although this choice reduced the amount of complexity in the code, it is limiting. Augment the existing launch files so that these values are passed as parameters. Test this by launching the robot is in the "br" namespace and with message topics in this namespace as well.

8. Write an OpenCV program in Python that reads in images from a video camera and then draws a random circle on the image before displaying it in a window. Note that there exists a `circle` method that will do the drawing for you.

9. Write an OpenCV program in Python that reads in a jpeg image and then converts it to greyscale and then inverts it in intensity space. That is replaces intensity `x` with intensity `255-x`. Implement this twice. First, by looping over every pixel in the image. Then by leveraging numpy to have the operation broadcast to every pixel in the image. Which approach is faster?

References

[1] J. Howse and J. Minichino. *Learning OpenCV 4 Computer Vision with Python 3*. Packt Publishing, 3rd edition, 2020.

[2] A. S. Huang, Edwin E. Olson, and D. C. Moore. LCM: Lightweight communications and marshalling. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4057–4062, 2010.

[3] A. Koubass. *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Springer International, 2016.

[4] W. Newman. *A Systematic Approach to Learning Robot Programming with ROS*. CRC Press, Boca Raton, FL, 2018.

[5] J. M. O'Kane. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, 2013.

[6] M. Quigley, K. Conley, B. Gerky, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open source robot operating system. In *IEEE International Conference on Robotics and Automation Open Source Software Workshop*, Kobe, Japan, 2009.