CHAPTER 4

Kick-Starting Robot Programming Using ROS

The last three chapters discussed the prerequisites for programming a robot using the Robot Operating System (ROS). We discussed the basics of Ubuntu Linux, bash commands, the basic concepts of C++ programming, and the basics of Python programming. In this chapter, we start working with ROS. Before discussing ROS concepts, let's discuss robot programming and how we do it. After this, we learn more about ROS, how to install ROS, and its architecture.

After this, we look at ROS concepts, ROS command tools, and ROS examples to demonstrate ROS capabilities. After that, we discuss the basics of ROS GUI tools and the Gazebo simulator. In the end, we learn how to set up a TurtleBot 3 simulator in ROS.

What Is Robot Programming?

As you know, a robot is a machine with sensors, actuators (motors), and a computing unit that behaves based on user controls, or it can make its own decisions based on sensors inputs. We can say the brain of the robot is a computing unit. It can be a microcontroller or a PC. The decision making

L. Joseph, *Robot Operating System for Absolute Beginners*,
https://doi.org/10.1007/978-1-4842-3405-1_4

and actions of the robot completely depends on the program running the robot's brain. This program can be firmware running on a microcontroller, or C/C++ or Python code running on a PC or a single board computer, like the Raspberry Pi. Robot programming is the process of making the robot work from writing a program for the robot's brain (i.e., the processing unit).

Figure 4-1 shows a general block diagram of a robot, including the part where it programs.

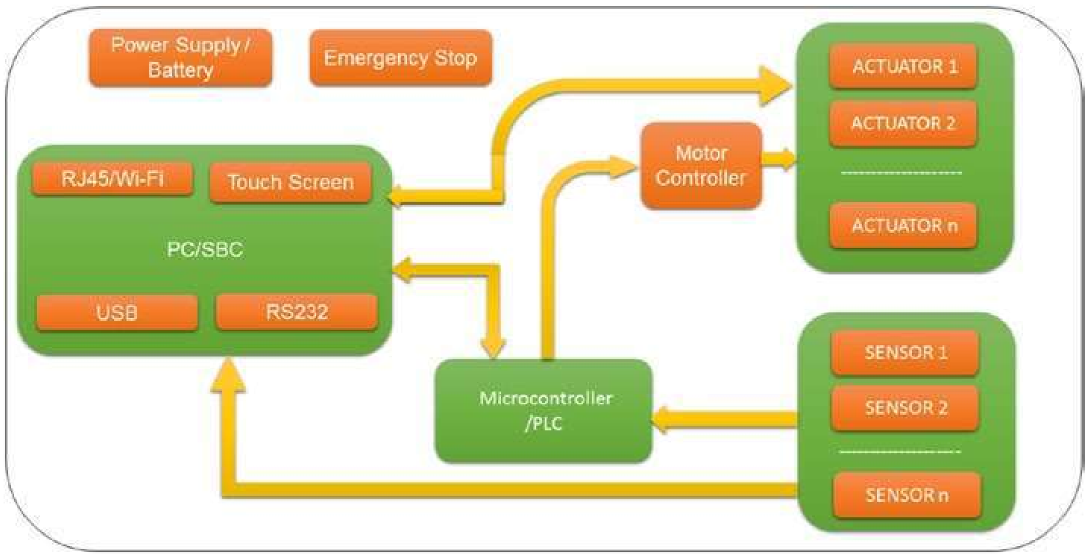


Figure 4-1. *General block diagram of a robot*

The main components of any robot are the actuators and the sensors. Actuators move a robot's joints, providing rotary or linear motion. Servo, Stepper, and DC gearmotors are actuator brands. Sensors provide the robot's state and environment. Examples of robot sensors include wheel encoders, ultrasonic sensors, and cameras.

Actuators are controlled by motor controllers and interface with a microcontroller/PLC (programmable logic controller). Some actuators are directly controlled through a PC's USB. Sensors also interface with a microcontroller or PC. Ultrasonic sensors and infrared sensor interface with a microcontroller. High-end sensors like cameras and laser scanners can interface directly with the PC. There is a power supply/battery to power all the robotic components. There is an emergency stop push-button to stop/reset the robot's operation. The two major parts in which to program inside a robot is a PC and a microcontroller/PLC. PLCs are mainly using in industrial robots.

In short, we can say robot programming is programming the PC/SBC and microcontroller/PCL inside robot for performing a specific application using actuators and feedback from various sensors. The robot applications includes pick and place of object, moving robot from A to B. A variety of programming languages can program robots. C/C++, Python, Java, C #, and so forth are used with PCs. Microcontrollers use Embedded C, the Wiring language (based on C++), which is used in Arduino, and Mbed programming (`https://os.mbed.com`). Industrial robot applications use SCADA or vendors' proprietary programming languages, such as ABB and KUKA. This programming is done from the industrial robot's teach pendant. RAPID is the programming language used in ABB industrial robots to automate robotic applications.

Robotic programming creates intelligence in the robot for self-decision making, implementing controllers like PID to move joints, automating repeated tasks, and creating robotic vision applications.

Why Robot Programming Is Different

Robot programming is a subset of computer programming. Most robots have a "brain" that can make decisions. It can be a microcontroller or a PC. The differences between robot programming and conventional programming are the input and output devices. The input devices include robot sensors, teach pendants, and touch screens, and the output devices include LCD displays and actuators.

Any of the programming languages can program robots, but good community support, performance, and prototyping time make C++ and Python the most commonly used.

The following are some of the features needed for programming a robot.

- *Threading*: As seen in the robot block diagram, there are a number of sensors and actuators in a robot. We may need a multithreaded compatible programming language in order to work with different sensors and actuators in different threads. This is called *multitasking*. Each thread can communicate with each other to exchange data.
- *High-level object-oriented programming*: As you already know, object-oriented programming languages are more modular and code can easily reused. The code maintenance is also easy compared to non-object-oriented programming languages. These qualities create better software for robots.
- *Low-level device control*: The high-level programming langauges can also access low-level devices such as GPIO (general purpose input/output) pins, Serial ports, parallel ports, USB, SPI, and I2C. Programming languages like C/C++ and Python can work with low-level devices, which is why these languages prefer single-board computers like the Raspberry Pi and Odroid.
- *Ease of prototyping*: The easiness in prototyping a robot algorithm is definitely a choice in the selection of programming language. Python is a good choice in prototyping robot algorithms quickly.
- *Interprocess communication*: A robot has lot of sensors and actuators. We can use multithreading architecture or write an independent program for doing each task; for example, one program takes images from a camera and detects a face, another program sends

data to an embedded board. These two programs can communicate with each other to exchange data. This feature creates multiple programs instead of a multithreading system. The multithreading system is more complicated than running multiple programs in parallel. Socket programming is an example of interprocess communication.

- *Performance*: If we work with high-bandwidth sensors, such as depth cameras and laser scanners, the computing resources needed to process the data is obviously high. A good programming language can only allocate appropriate computing resource without loading the computing resource. The C++ langauges is a good choice to handle these kind of scenerio.
- *Community support*: When choosing any programming language for robot programming, make sure that there is enough community support for that language, including forums and blogs.
- *Availability of third-party libraries*: The availability of third-party libraries can make our development easy; for example, if we want to do image processing, we can use libraries like OpenCV. If your programming language has OpenCV support, it is easier to do image processing applications.
- *Existing robotics software framework support*: There are existing robotics software frameworks such as ROS to program robots. If your programming language has ROS support, it is easier to prototype a robot application.

Getting Started with ROS

So far, we have discussed robot programming and how it is different from other computer programming. In this section, we look at a unique software platform for programming robots: the Robot Operating System, or ROS (`www.ros.org`).

ROS is a free and open source robotics software framework that is used in both commercial and research applications. The ROS framework provides the following robot-programming capabilities.

- *Message passing interface between processes*. ROS provides a message passing interface to communicate between two programs or processes. For example, a camera processes an image and finds coordinates in the image, and then these coordinates are sent to a tracker process. The tracker process does the tracking of the image by using motors. As mentioned, this is one of the features needed to program a robot. It is called *interprocess communication* because two processes are communicating with each other.
- *Operating system–like features*. As the name says, ROS is not a real operating system. It is a meta operating system that provides some operating system functionalities. These functionalities include multithreading, low-level device control, package management, and hardware abstraction. The hardware abstraction layer enables programmers to program a device. The advantage is that we can write code for a sensor that works the same way with different vendors. So, we don't need to rewrite the code when we use a new sensor. Package management helps users organize software in units called *packages*. Each package has

source code, configuration files, or data files for a specific task. These packages can be distributed and installed on other computers.

- *High-level programming language support and tools*. The advantage of ROS is that it supports popular programming languages used in robot programming, including C++, Python, and Lisp. There is experimental support for languages such as C #, Java, Node.js, and so forth. The complete list is at `http://wiki.ros.org/Client%20Libraries`. ROS provides client libraries for these languages, meaning the programmer can get ROS functionalities in the languages mentioned. For example, if a user wants to implement an Android application that is using ROS functionality, the rosjava client library can be used. ROS also provides tools to build robotics applications. With these tools, we can build many packages with a single command. This flexibility helps programmers spend less time in creating build systems for their applications.
- *Availability of third-party libraries*. The ROS framework is integrated with most popular third-party libraries; for example, OpenCV (`https://opencv.org`) is integrated for robotic vision, and PCL (`http://pointclouds.org`) is integrated for 3D robot perception. These libraries make ROS stronger, and the programmer can build powerful applications on top of it.
- *Off-the-shelf algorithms*. This is a useful feature. ROS has implemented popular robotics algorithms such as PID (`http://wiki.ros.org/pid`); SLAM (Simultaneous Localization and Mapping) (`http://wiki.ros.org/gmapping`); and path planners such as A*, Dijkstra

(http://wiki.ros.org/global_planner), and AMCL (Adaptive Monte Carlo Localization) (http://wiki.ros.org/amcl). The list of algorithm implementations in ROS continuous. The off-the shelf algorithms reduce development time for prototyping a robot.

- *Ease in prototyping*. One advantage of ROS is off-the-shelf algorithms. Along with that, ROS has packages that can be easily reused with any robot; for example, we can easily prototype our own mobile robot by customizing an existing mobile robot package available in the ROS repository. We can easily reuse the ROS repository because most of the packages are open source and reusable for commercial and research purposes. So, this can reduce robot software development time.

- *Ecosystem/community support*. The main reason for the popularity and development of ROS is community support. ROS developers are all over the world. They actively develop and maintain ROS packages. The big community support includes developers asking questions related to ROS. ROS Answers is a platform for ROS-related queries (https://answers.ros.org/questions/). ROS Discourse is an online forum in which ROS users discuss various topics and publish news related to ROS (https://discourse.ros.org).

- *Extensive tools and simulators*. ROS is built with many command-line and GUI tools to debug, visualize, and simulate robotics applications. These tools are very useful for working with a robot. For example, the Rviz (http://wiki.ros.org/rviz) tool is used for visualization with cameras, laser scanners, inertial measurement units, and so forth. For working with robot simulations, there are simulators such as Gazebo (http://gazebosim.org).

The ROS Equation

The ROS project can be defined in a single equation, as shown in Figure 4-2.

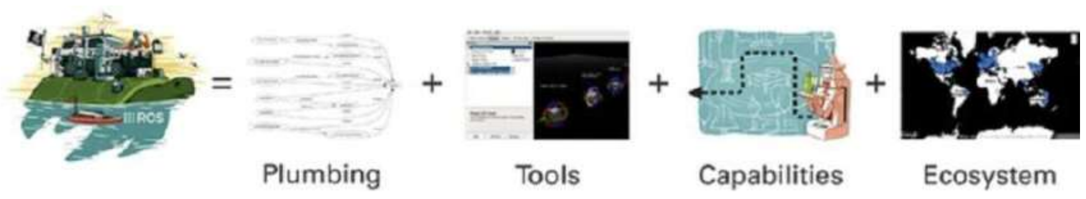


Figure 4-2. *The ROS equation*

The plumbing is the same as the message passing interface.
ROS has many other capabilities, which we explore in upcoming sections.

Robot Programming Before and After ROS

Let's look at the changes to the robotics programming community since the ROS project began.

The History of ROS

The following are some ROS project historic milestones.

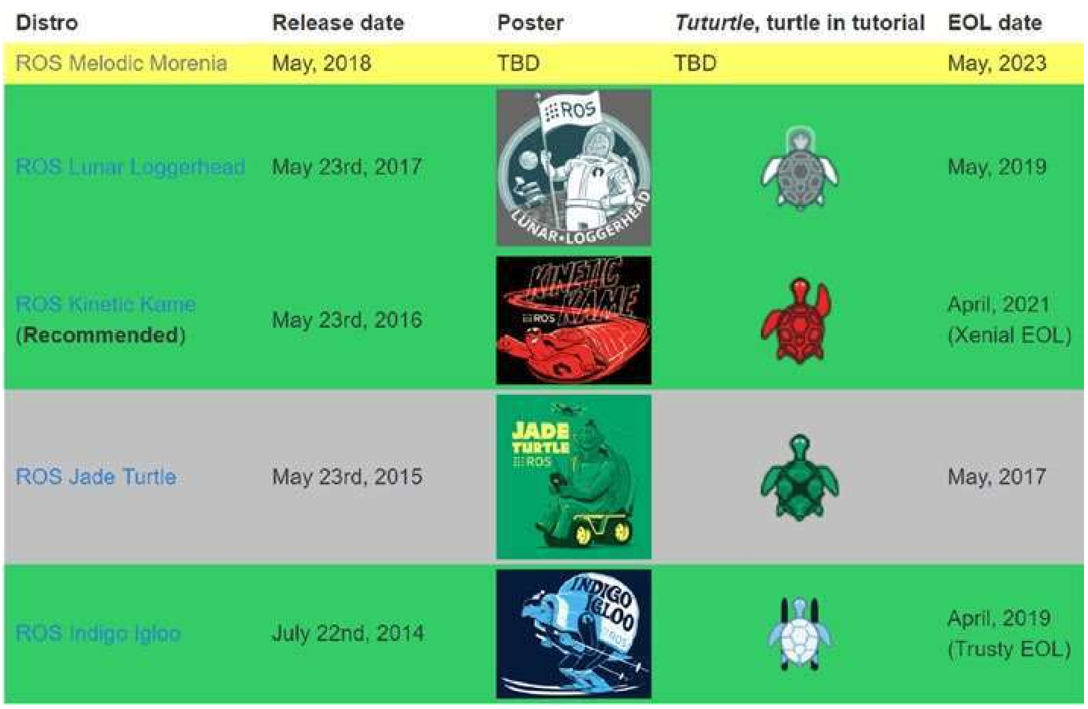
- The ROS project started at Stanford University in 2007, led by roboticist Morgan Quigly (http://people.osrfoundation.org/morgan/). In the beginning it was a group of software developed for robots at Stanford.

- Later in 2007, a robotics research startup called Willow Garage (`http://www.willowgarage.com/`) took over the project and coined the name ROS, which stands for Robot Operating System.
- In 2009, ROS 0.4 was released, and a working ROS robot called PR2 was developed.
- In 2010, ROS 1.0 was released. Many of its features are still in use.
- In 2010, ROS C Turtle was released.
- In 2011, ROS Diamondback was released.
- In 2011, ROS Electric Emys was released.
- In 2012, ROS Fuerte was released.
- In 2012, ROS Groovy Galapagos was released.
- In 2012, the Open Source Robotics Foundation (OSRF) takes over the ROS project.
- In 2013, ROS Hydro Medusa was released.
- In 2014, ROS Indigo Igloo was released, this was the first long-term support (LTS) release, meaning updates and support is provided for a long period of time (typically five years).
- In 2015, ROS Jade Turtle was released.
- In 2016, ROS Kinetic Kame was released. It is the second LTS version of ROS.
- In 2017, ROS Lunar Loggerhead was released.
- In May 2018, the twelfth version of ROS, Melodic Morenia, was released.

The timeline of the ROS project and a more detailed history is available in at www.ros.org/history/.

Each version of ROS is called a ROS distribution. You may be aware of the Linux distribution, such as Ubuntu, Debian, Fedora, and so forth.

Figure 4-3 shows the complete list of ROS distribution releases (http://wiki.ros.org/Distributions).



Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Melodic Morenia	May, 2018	TBD	TBD	May, 2023
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame (Recommended)	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)

Figure 4-3. *ROS distributions*

If you are looking for the latest ROS features, you can choose new distributions, and if you are looking for stable packages, you can choose LTS. In Figure 4-3, the recommended distribution is ROS Kinetic Kame. In this book, the examples use Kinetic Kame.

ROS is now developed and maintained by the Open Robotics, previously known as the Open Source Robotics Foundation (www.osrfoundation.org).

Before and After ROS

There was active development in robotics before the ROS project, but there was no common platform and community for developing robotics applications. Each developer created software for their own robot, which in most cases, couldn't be reused for any other robot. Developers had to rewrite code from scratch for each robot, which takes a lot of time. Also, most of the code was not actively maintained, so there was no support for the software. Also, developers needed to implement standard algorithms on their own, which took more time to prototype the robot.

After the ROS project, things changed. Now there is a common platform for developing robotics applications. It is free and open source for commercial and research purposes. Off-the-shelf algorithms are readily available, so there is no longer a need to code. There is big community support, which makes development easier. In short, the ROS project changed the face of robotics programming.

Why Use ROS?

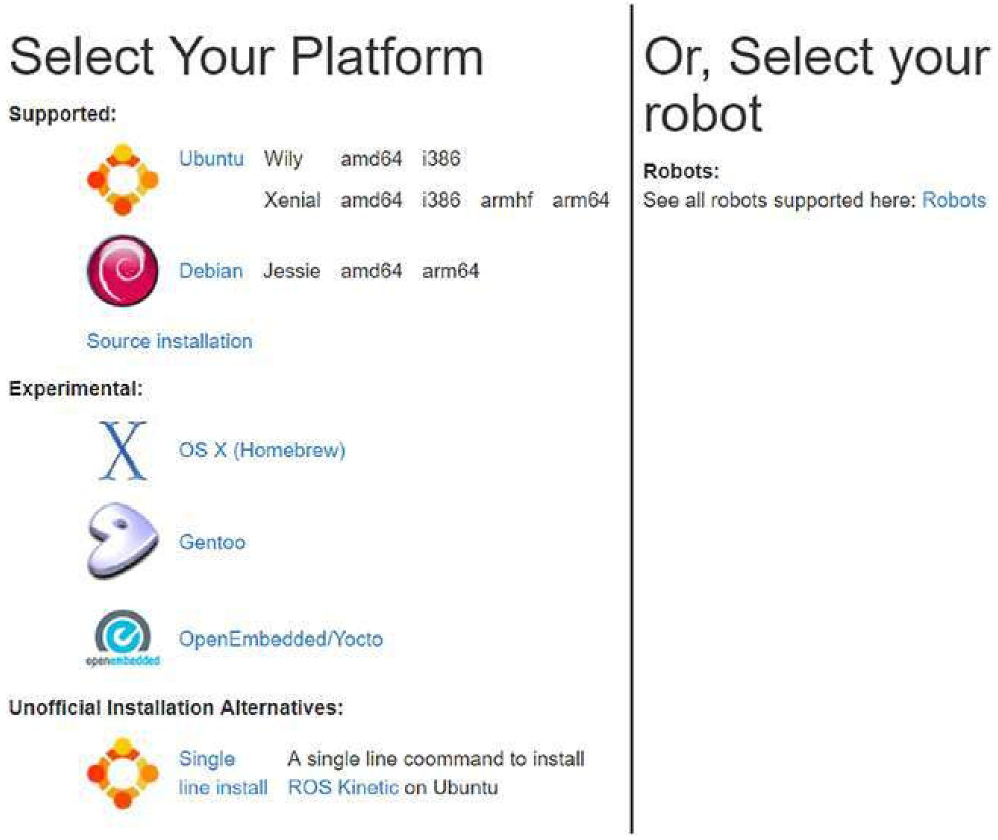
This is common question that developers ask when looking for a platform to program ROS. Although ROS has many features, there are still areas in which ROS can't be used or is not recommended to use. In the case of a self-driving car, for example, we can use ROS to make a prototype, but developers do not recommend ROS to make the actual product. This is due to various issues, such as security, real-time processing, and so forth. ROS may not be a good fit in some areas, but in other areas, ROS is an absolute fit. In corporate robotics research centers and at universities, ROS is an ideal choice for prototyping. And ROS is used in some robotics products after lot of fine-tuning (but not self-driving cars).

A project called ROS 2.0 is developing a much better version of the existing ROS in terms of security and real-time processing (`https://github.com/ros2/ros2/wiki`). ROS 2.0 may become a good choice for robotics products in the future.

Installing ROS

This is an important step in ROS development. Installing ROS on your PC is a straightforward process. Before installing, you should be aware of the various platforms that support ROS.

Figure 4-4 shows various operating systems on which you can install ROS. As discussed, ROS is not an operating system, but it needs a host operating system to work.



Select Your Platform

Supported:

Ubuntu	Wily	amd64	i386		
	Xenial	amd64	i386	armhf	arm64
Debian	Jessie	amd64	arm64		

Source installation

Experimental:

OS X (Homebrew)

Gentoo

OpenEmbedded/Yocto

Unofficial Installation Alternatives:

Single line install — A single line coommand to install ROS Kinetic on Ubuntu

Or, Select your robot

Robots:

See all robots supported here: Robots

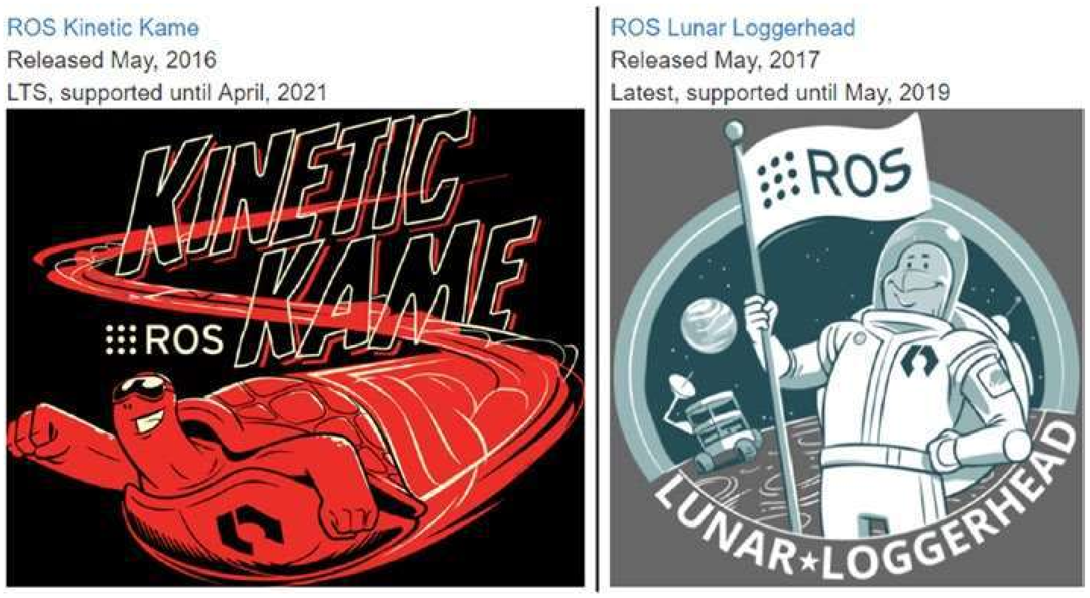
Figure 4-4. *Operating systems that support ROS*

Ubuntu Linux is the most preferred OS for installing ROS. As you can see in Figure 4-4, ROS supports Ubuntu 32-bit, 64-bit, ARM 32-bit, and ARM 64-bit. This means ROS can run on PC/desktops and on single-board computers like Raspberry Pi (http://raspberrypi.org), Odroid (www.hardkernel.com/main/main.php), and NVIDIA TX1/TX2 (www.nvidia.com/en-us/autonomous-machines/embedded-systems/). Debian Linux (www.debian.org) has good ROS support.

In OS X and other operating systems, ROS is still in the experimental phase, which means that ROS functionalities are not yet available.

Let's move on to installation. If you are using a PC or an ARM board running Ubuntu armhf or arm64, you can follow the procedures at http://wiki.ros.org/ROS/Installation.

When you go to this wiki, it asks which ROS version you need to install. Figure 4-5 is a typical website screenshot.


ROS Kinetic Kame
Released May, 2016
LTS, supported until April, 2021

ROS Lunar Loggerhead
Released May, 2017
Latest, supported until May, 2019

Figure 4-5. *Choosing a ROS distribution*

As mentioned, we are choosing ROS Kinetic Kame because it is LTS and stable. If you want to try the latest ROS features, use Lunar Loggerhead.

After you click the distribution that you want, you get the list of operating systems that support that distribution. The list of ROS Kinetic operating systems are shown in Figure 4-4.

Choose the Ubuntu 16.04 (Xenial) operating system. When you select the operating system, you get a set of instructions. The wiki at `http://wiki.ros.org/kinetic/Installation/Ubuntu` provides direct access to instructions for setting ROS in Ubuntu.

We can install ROS in two ways: through a binary installation or by source compilation. The first method is easy and less time-consuming. Binary installation lets you directly install ROS from prebuilt binaries. With source compilation, you create an executable by compiling ROS source code. This takes more time and is based on your PC's specifications.

In this book, we are doing a binary installation.

The following describes the installation steps.

1) *Configure the Ubuntu repositories*. An Ubuntu repository is where the Ubuntu software is organized, typically on servers in which users can access and install the application. There following are repositories in Ubuntu.

    a) Main: Ubuntu officially supported free and open source software

    b) Universe: Community maintained free and open source software

    c) Restricted: This has proprietary device drivers

    d) Multiverse: Software restricted by copyright and legal issues

    To install ROS, we have to enable access to the entire repository so that Ubuntu can retrieve packages from these repositories. Figure 4-6 shows how to do this. Just search in Ubuntu for "Software & Updates".

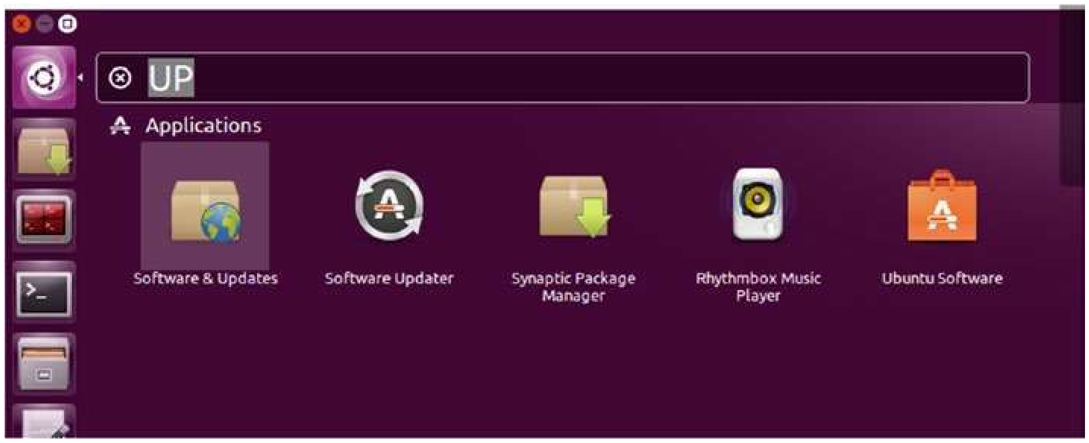
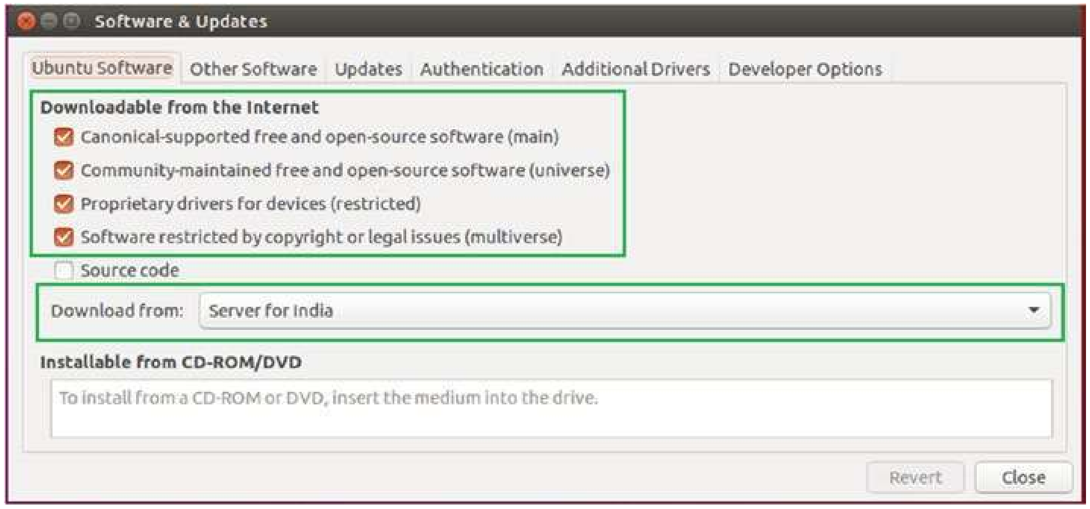


Figure 4-6. *Searching for the Software & Updates application in Ubuntu*

Figure 4-7 shows that you can enable the access of each repository. You can also select the server location. You can either use a server from your country or the Ubuntu main server.



Downloadable from the Internet

- ✓ Canonical-supported free and open-source software (main)
- ✓ Community-maintained free and open-source software (universe)
- ✓ Proprietary drivers for devices (restricted)
- ✓ Software restricted by copyright or legal issues (multiverse)
- ☐ Source code

Download from: Server for India

Installable from CD-ROM/DVD

To install from a CD-ROM or DVD, insert the medium into the drive.

Figure 4-7. *The Software & Updates application in Ubuntu*

OK, you are done with the first step.

2) *Set up your sources.list*. This is an important step in ROS installation. It adds the ROS repository information where the binaries are stored. Ubuntu can fetch the packages after this step is completed. The following is the command used for this.

Note Execute the following commands in a terminal.

```
$ sudo sh -c 'echo "deb http://packages.ros.
org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
```

This command creates a new file called `/etc/apt/sources.list.d/ros-latest.list` and adds the following line to it.

```
"deb http://packages.ros.org/ros/ubuntu xenial main".
```

If we create this file in the `sources.list` folder and add this line, then only Ubuntu package manager can fetch the package list.

Note If you execute `$ lsb_release -sc` in a terminal, you get the output `'xenial'`.

---

3) *Add the keys*. In Ubuntu, if we want to download a binary or a package, we have to add a secure key in our system to authenticate the downloading process. The package that authenticates using these keys is trusted. The following is the command to add the keys.

```
$ sudo apt-key adv --keyserver hkp://
ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

4) *Update the Ubuntu package list*. When we update the list, the packages from the ROS repositories also list. We use the following command to update the Ubuntu repository.

```
$ sudo apt-get update
```

5) *Install ROS Kinetic packages*. After getting the list, we download and install the package using the following command.

```
$ sudo apt-get install ros-kinetic-desktop-full
```

This command installs all the necessary packages in ROS, including tools, simulators, and essential robot algorithms. It takes time to download and install all these packages.

6) *Initialize rosdep*. After installing all packages, we need to install a tool called rosdep, which is useful for installing the dependent packages of a ROS package. For example, a typical ROS package may have a few dependent packages to work properly. rosdep checks whether the dependent packages are available, and if not, it automatically installs them.

The following command installs the rosdep tool.

```
$ sudo rosdep init
$ rosdep update
```

7) *Set the ROS environment*. This is an important step after installing ROS. As discussed earlier, ROS comes with tools and libraries. To access these command-line tools and packages, we have to set up the ROS

environment to access these commands, even though its installed on our system. The following command adds a line in the `.bashrc` file in your home folder, which sets the ROS environment in every new terminal.

```
$ echo "source /opt/ros/kinetic/setup.bash" >>
~/.bashrc
```

Next, enter the following command to add the environment in the current terminal.

```
$ source ~/.bashrc
```

Yes, you are almost done. A small step remains.

8) *Set up dependencies for building the package.* The use of this step can be explained using an example. Imagine that you are working with a robot with more than 100 packages. If you want to set up those packages in a computer, it is difficult to manage all the dependencies needed to install those packages. In that situation, tools like rosinstall are useful. This tool installs all the packages in a single command. In this step, we are literally installing those kinds of tools.

```
$ sudo apt-get install python-rosinstall
python-rosinstall-generator python-wstool
build-essential
```

Congratulations, you are done with installation. You can verify that your installation is correct by using the following command.

```
$ rosversion -d
```

If you are getting `'kinetic'` as the output, you are all set with the installation.

Robots and Sensors Supporting ROS

Figure 4-8 shows some of the popular robots that use ROS. A complete list of robots working in ROS is at http://robots.ros.org.

The following are the robots shown in Figure 4-8.

a) Pepper (www.ald.softbankrobotics.com/en/robots/pepper): A service robot used for assisting people in a variety of ways.

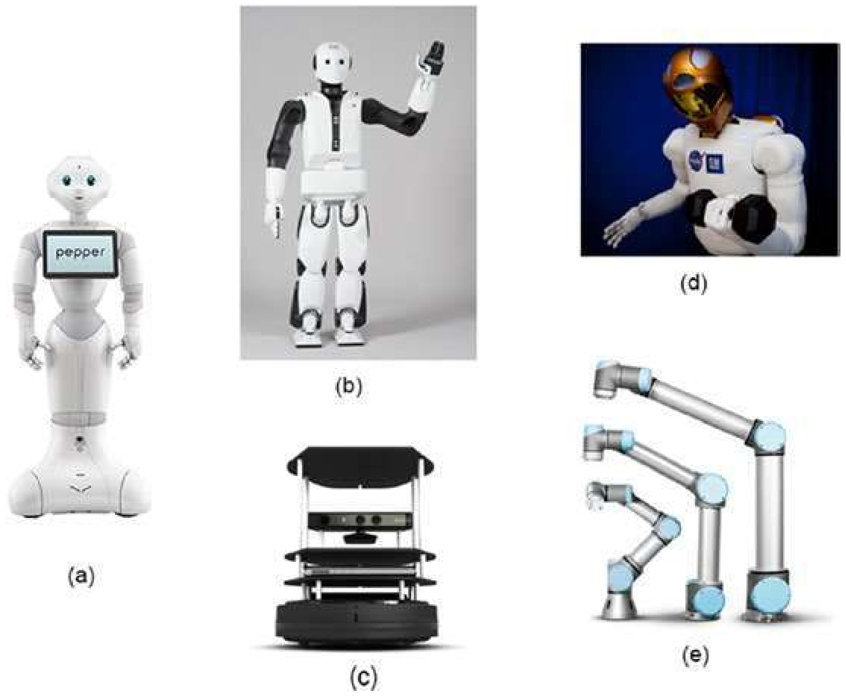


Figure 4-8. *Robots that work in ROS*

b) REEM-C (http://pal-robotics.com/en/products/reem-c/): A full-size humanoid robot that is mainly used for research purposes.

c) TurtleBot 2 (www.turtlebot.com/turtlebot2/): A simple mobile robot platform that is mainly used for research and educational purposes.

d) Robonaut 2 (https://robonaut.jsc.nasa.gov/R2/): A NASA robot designed to automate various tasks on the International Space Station.

e) Universal Robot arm: (www.universal-robots.com/products/ur5-robot): One of the popular semi-industrial robots widely used for automating various tasks in manufacturing.

There are also sensors supported by ROS. A complete list of these sensors is available at http://wiki.ros.org/Sensors (see Figure 4-9).

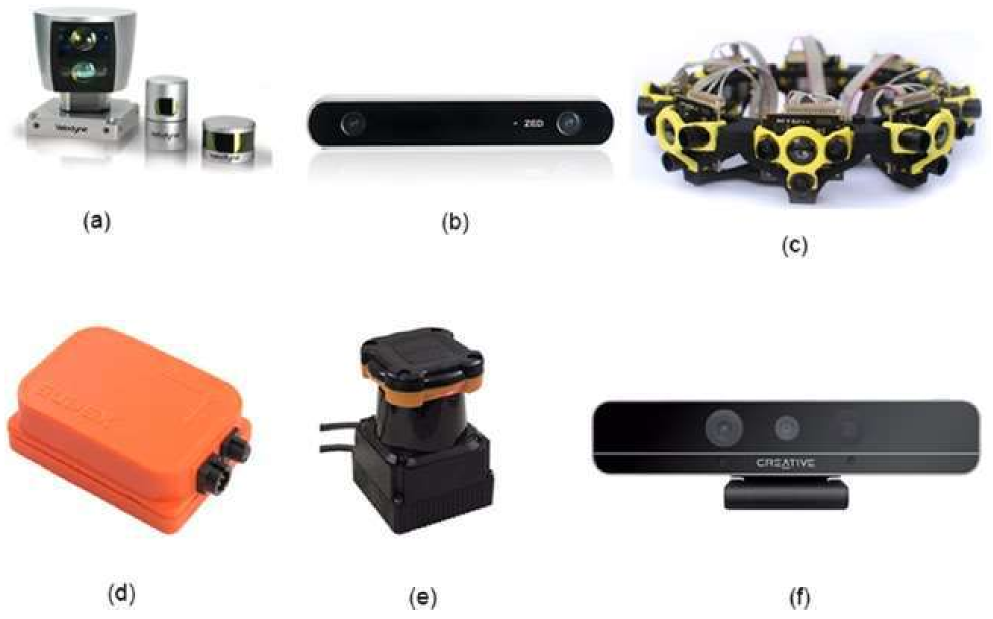


Figure 4-9. *Popular sensors that support ROS*

The following describes each sensor shown in Figure 4-9.

a) Velodyne (http://velodynelidar.com): Popular LIDARs mainly used in self-driving cars.

b) ZED Camera (www.stereolabs.com): A popular stereo depth camera.

c) TeraRanger (www.terabee.com): A new sensor for depth sensing in 2D and 3D.

d) Xsense MTi IMU (www.xsens.com/products/): An accurate IMU solution.

e) Hokuyo Laser (www.hokuyo-aut.jp/): A popular laser scanner.

f) Intel RealSense (https://realsense.intel.com: A 3D depth sensor for robot navigation and mapping.

Popular ROS Computing Platforms

Figure 4-10 shows a few commonly used ROS-compatible computing platforms.

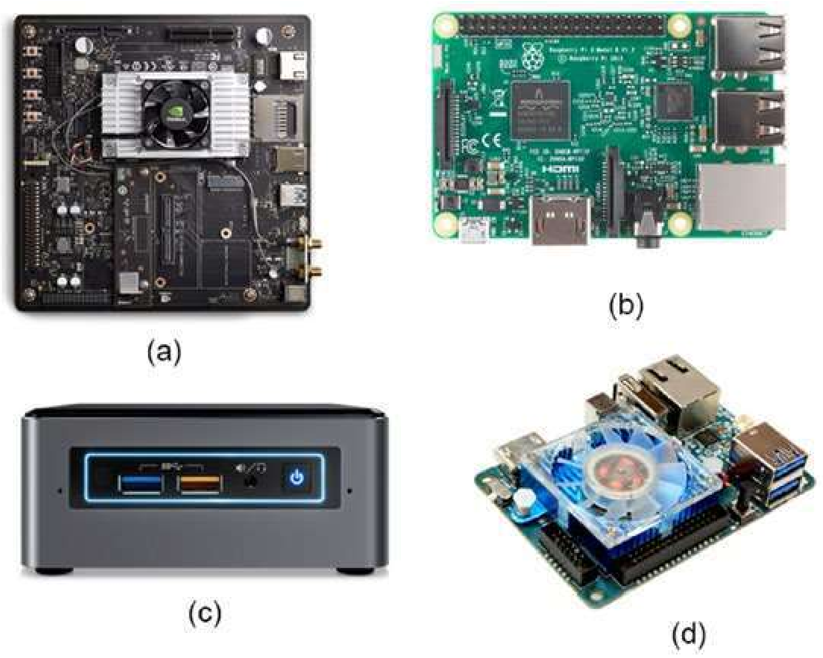


Figure 4-10. *Popular computing units that run ROS*

a) NVDIA TX1/TX2 (`www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/`): Capable of running deep learning applications and computational intensive applications. The board has an ARM-based 64-bit processor that can run Ubuntu. This platform is very popular in autonomous robotics applications, especially drones.

b) Raspberry Pi 3 (`www.raspberrypi.org/products/raspberry-pi-3-model-b/`): Very popular single-board computers for education and research. Robotics is a key area.

c) Intel NUC (`www.intel.com/content/www/us/en/products/boards-kits/nuc.html`): Based on a x86_64 platform, which is basically a miniature version of a desktop computer.

d) Odroid XU4 (`www.hardkernel.com/main/main.php`): The Odroid series boards are similar to Raspberry Pi, but it has better configuration and performance. It is based on the ARM architecture.

ROS Architecture and Concepts

We have discussed ROS, its features, and how to install it. In this section we go deep into ROS architecture and its important concepts. Basically, ROS is a framework to communicate between two programs or process. For example if program A wants to send a data to program B, and B wants to send data to program A, we can easily implement it using ROS. So the question is whether we implement it using socket programming directly. Yes, we can, but if we build more and more programs, it gets complex, so ROS is a good choice for interprocess communication.

Do we really need interprocess communication in a robot? Can we program a robot without it? The answer to the first question is explained in Figure 4-11.

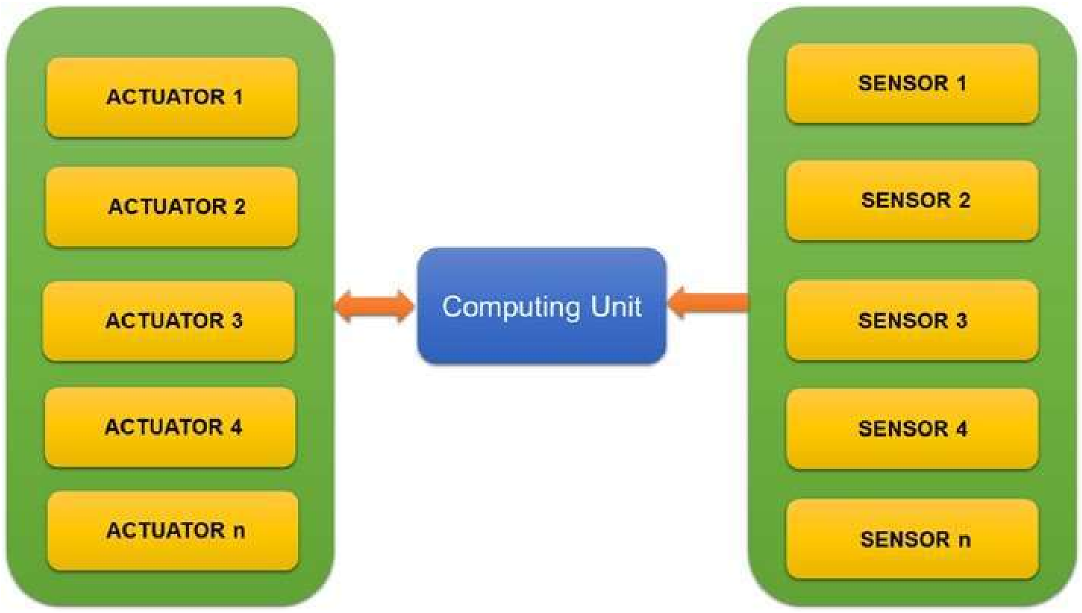


Figure 4-11. *A typical robot block with actuators and sensors*

A robot may have many sensors and actuators, as well as a computing unit. How we can control many actuators and process so much sensor data? Can we do it in a single program? Yes, but that is not a good way of doing it. The better way is, we can write independent programs to handle sensor data and controlling actuators, and often we may need to exchange data between these programs. This is the situation where we use ROS.

So can we program a robot without ROS? Yes, but the complexity of software increases according to the number of actuators and sensors.

Let's see how the communication is happening between two programs in ROS. Figure 4-12 illustrates a basic block diagram of ROS.

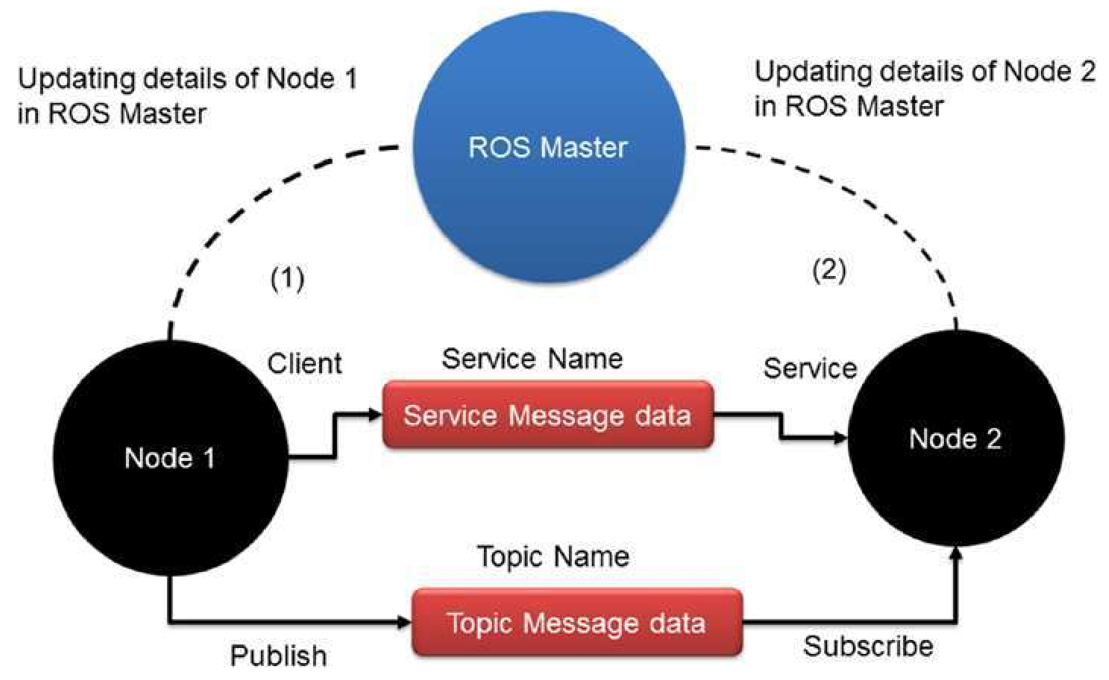


Figure 4-12. *ROS Communication block diagram*

Figure 4-12 shows two programs marked as node 1 and node 2. When any of the programs start, a node communicates to a ROS program called the ROS master. The node sends all its information to the ROS master, including the type of data it sends or receives. The nodes that are sending a data are called *publisher nodes*, and the nodes that are receiving data are called *subscriber nodes*. The ROS Master has all the publisher and subscriber information running on computers. If node 1 sends particular data called "A" and the same data is required by node 2, then the ROS master sends the information to the nodes so that they can communicate with each other.

The ROS nodes can send different types of data to each other, which includes primitive data types such as integer, float, string, and so forth. The different data types being sent are called *ROS messages*. With ROS messages, we can send data with a single data type or multiple data with different data types. These messages are sent through a message bus or path called *ROS topics*. Each topics has a name; for example, a topic named "chatter" sends a string message.

When a ROS node publishes a topic, it sends a ROS topic with a ROS message, and it has data with the message type.

In Figure 4-12, the ROS topic is publishing and subscribing node 1 and node 2. This process starts when the ROS master exchange the node details to each other.

Next, let's go through some important concepts and terms that are used when working with ROS. They can be classified as three categories: the ROS file system, ROS computation concepts, and the ROS community.

The ROS File System

The ROS file system includes packages, meta packages, package manifests, repositories, message types, and services types.

ROS packages are the individual units, or the *atomic units,* of ROS software. All source code, data files, build files, dependencies, and other files are organized in packages. A ROS meta package groups a set of similar packages for a specific application. A ROS meta package does not have any source files or data files. It has the dependencies of similar packages. A ROS meta package organizes a set of packages.

A *package manifest* is an XML file placed inside a ROS package. It has all the primary information of a ROS package, including the name of the package, description, author, dependencies, and so forth. A typical package.xml is shown next.

```
<?xml version="1.0"?>
<package>
  <name>test_pkg</name>
  <version>0.0.1</version>
  <description>The test package</description>

  <maintainer email="qboticslabs@gmail.com">robot</maintainer>

  <license>BSD</license>
  <buildtool_depend>catkin</buildtool_depend>
  .................          ............... .
  <run_depend>catkin</run_depend>
  ....... .                    ............ .
</package>
```

A *ROS repository* is a collection of ROS packages that share a common version control system.

A *message type description* is the definition of a new ROS message type. There are existing data types available in ROS that can be directly used for our application, but if we want to create a new ROS message, we can. A new message type can be defined and stored inside the `msg` folder inside the package.

Similar to message type, a *service type definition* contains our own service definitions. It is stored in the `srv` folder.

Figure 4-13 shows a typical ROS package folder.

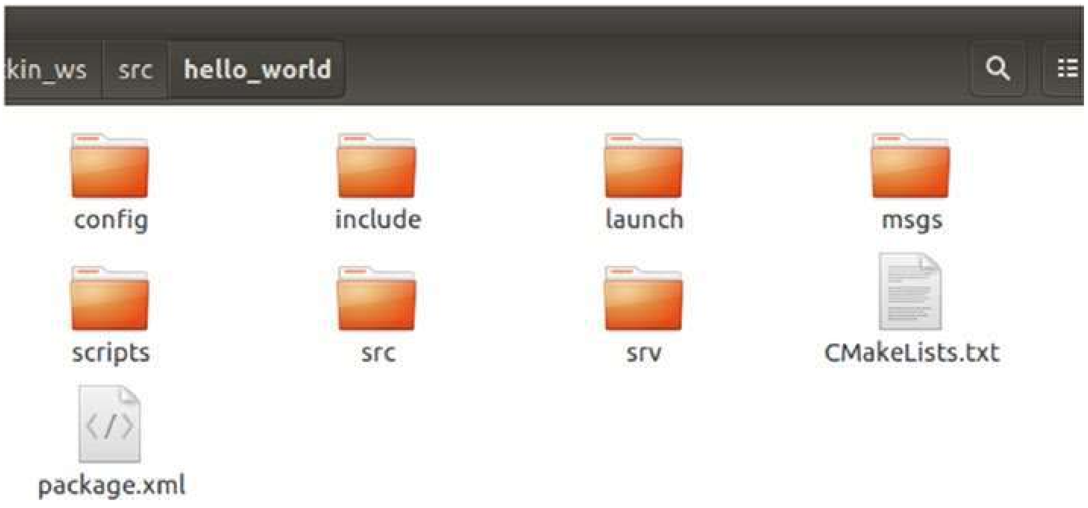


Figure 4-13. *A typical ROS package structure*

ROS Computation Concepts

These are the terms associated with ROS computation concepts.

- *ROS nodes*: Process that use ROS APIs to perform computations.
- *ROS master*: An intermediate program that connects ROS nodes.
- *ROS parameter server*: A program that normally runs along with the ROS master. The user can store various parameters or values on this server and all the nodes can access it. The user can set privacy of the parameter too. If it is a public parameter, all the nodes have access; if it is private, only a specific node can access the parameter.
- *ROS topics*: Named buses in which ROS nodes can send a message. A node can publish or subscribe any number of topics.
- *ROS message*: The messages are basically going through the topic. There are existing messages based on primitive data types, and users can write their own messages.
- *ROS service*: We have already seen ROS Topics, which is having publishing and subscribing mechanism. The ROS Service has Request/Reply mechanism. A service call is a function, which can call whenever a client node sends a request. The node who create a service call is called Server node and who call the service is called client node.

- *ROS bags*: A useful method to save and play back ROS topics. Also useful for logging the data from a robot to process it later.

The ROS Community

The following are terms used to exchange ROS software and knowledge.

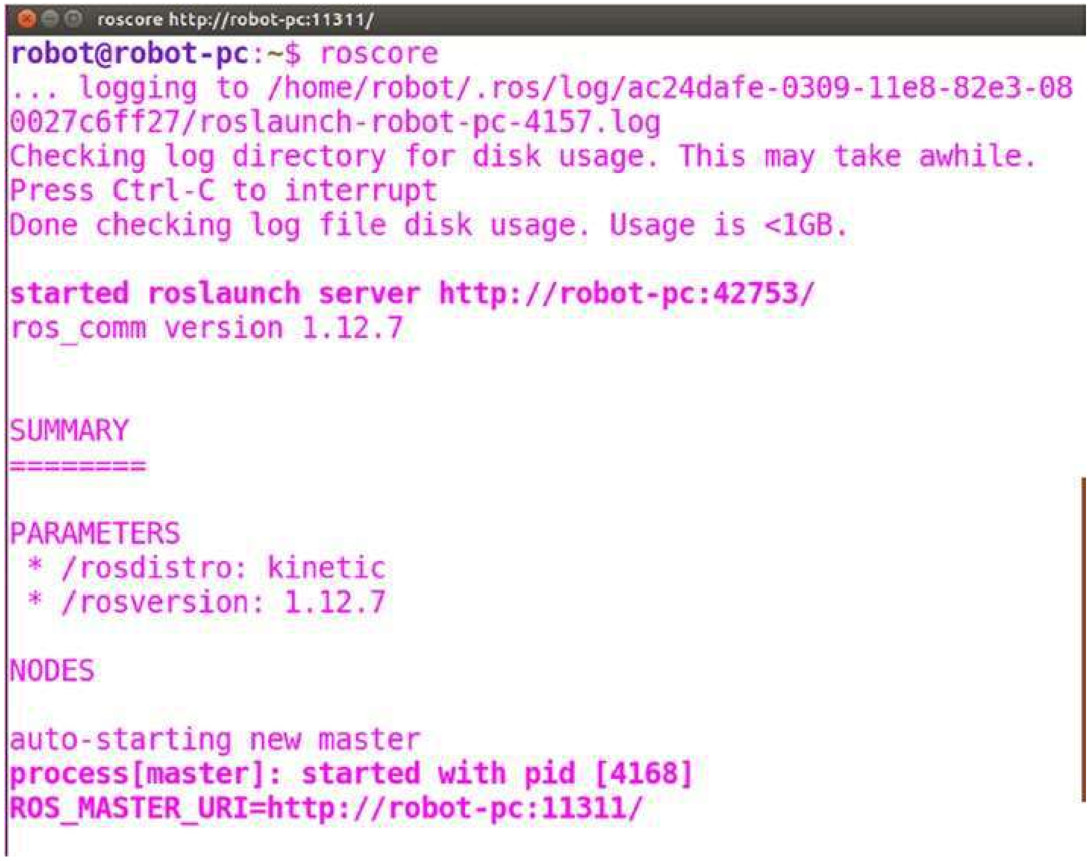
- The *ROS distribution* is a collection of versioned packages.
- The *ROS wiki* has tutorials on how to set up and program ROS.
- *ROS Answers* (`https://answers.ros.org/questions/`) has ROS queries and solutions, similar to Stack Overflow.
- *ROS discourse* (`https://discourse.ros.org`) is a forum in which developers can share news and ask queries related to ROS.

If you want to learn more about ROS concepts, visit `http://wiki.ros.org/ROS/Concepts`.

ROS Command Tools

This section discusses ROS command-line tools. What are these tools for? The tools can make our lives easier. There are different ROS tools that we can use to explore various aspects of ROS. We can implement almost all the capabilties of ROS using these tools. The command-line tools are executed in the Linux terminal; like the other commands in Linux, we get the ROS command tools too.

The `roscore` command is a very important tool in ROS. When we run this command in the terminal, it starts the ROS master, the parameter server, and a logging node. We can run any other ROS program/node after running this command. So run roscore on one terminal window, and use another terminal window to enter the next command to run a ROS node. If you run roscore in a terminal, you may get messages like the ones shown in Figure 4-14.



```
robot@robot-pc:~$ roscore
... logging to /home/robot/.ros/log/ac24dafe-0309-11e8-82e3-08
0027c6ff27/roslaunch-robot-pc-4157.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://robot-pc:42753/
ros_comm version 1.12.7


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES

auto-starting new master
process[master]: started with pid [4168]
ROS_MASTER_URI=http://robot-pc:11311/
```

Figure 4-14. *roscore messages*

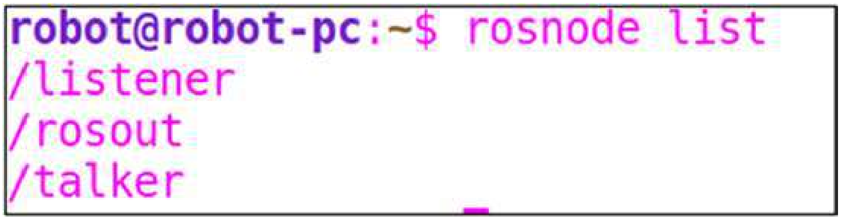
You can see messages in the terminal about starting the ROS master. You also see the ROS master address.

The `rosnode` command explores all the aspects of a ROS node. For example, we can list the number of ROS nodes running on our system. If you type any of the commands, you get complete help for the tool.

The following is a common usage of rosnode.

```
$ rosnode list
```

Figure 4-15 shows the list of nodes running on the system. It is a typical output of `rosnode list`.



```
robot@robot-pc:~$ rosnode list
/listener
/rosout
/talker
```

Figure 4-15. *Output of a rosnode list command*

The `rostopic` command provides information about the topics publishing/subscribing in the system. This command is very useful for listing topics, printing topic data, and publishing data.

```
$ rostopic list
```

If there is a topic called `/chatter`, we can print/echo the topic data using the following command.

```
$ rostopic echo /chatter
```

If we want to publish a topic with data, we can easily do this command.

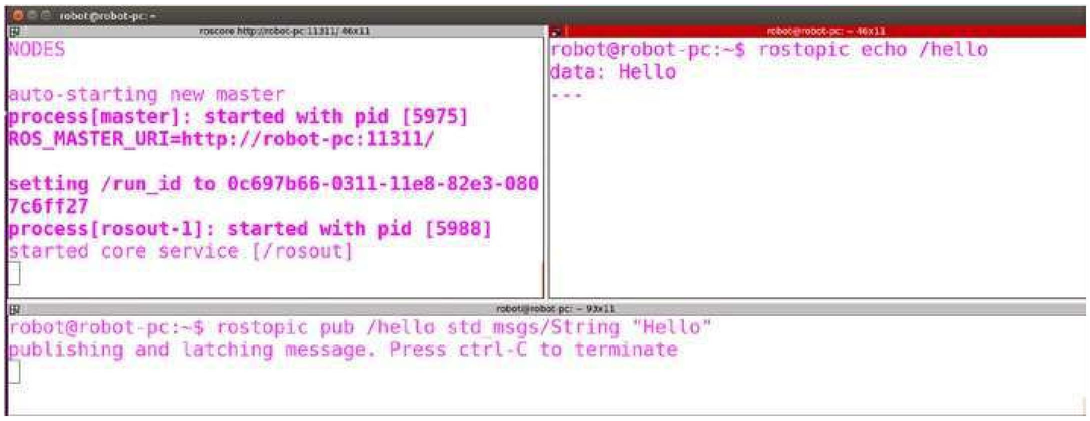
```
$ rostopic pub topic_name msg_type data
```

The following is an example.

```
$ rostopic pub /hello std_msgs/String "Hello"
```

You can echo the same topic after publishing too. Note that if you run these commands in one terminal, roscore should be running.

Figure 4-16 is a screenshot of rostopic echo and publish.



```
NODES

auto-starting new master
process[master]: started with pid [5975]
ROS_MASTER_URI=http://robot-pc:11311/

setting /run_id to 0c697b66-0311-11e8-82e3-080
7c6ff27
process[rosout-1]: started with pid [5988]
started core service [/rosout]
```

```
robot@robot-pc:~$ rostopic echo /hello
data: Hello
---
```

```
robot@robot-pc:~$ rostopic pub /hello std_msgs/String "Hello"
publishing and latching message. Press ctrl-C to terminate
```

Figure 4-16. *Output of rostopic echo and publish*

Figure 4-16 is the Terminator (`https://launchpad.net/terminator`) application in which the screen is split into separate terminal sessions. One session is running roscore. A second session is publishing a topic. A third session is echoing the same topic.

The `rosversion` command checks your ROS version.

The following command retrieves the current ROS version.

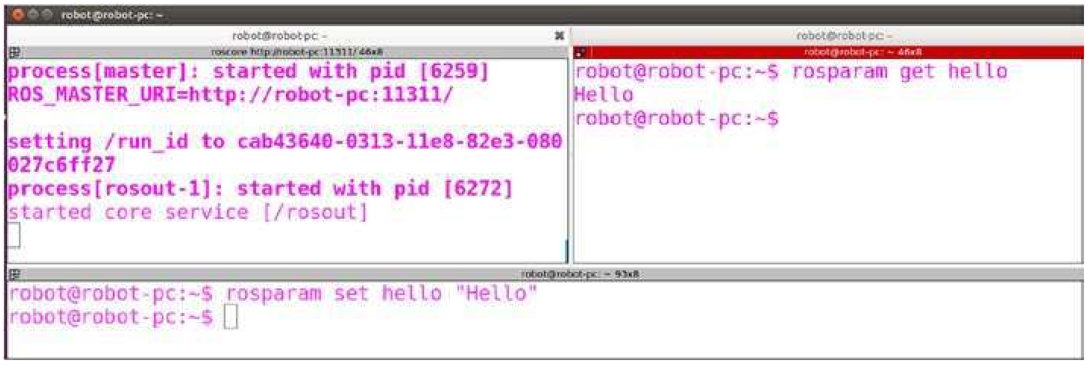
```
$ rosversion -d
Output: kinetic
```

The `rosparam` command gives a list of parameters loaded in the parameter server.

You can use the following command to list the parameters in the system.

```
$ rosparam list
```

Figure 4-17 shows how to set and get a parameter.



```
process[master]: started with pid [6259]
ROS_MASTER_URI=http://robot-pc:11311/

setting /run_id to cab43640-0313-11e8-82e3-080
027c6ff27
process[rosout-1]: started with pid [6272]
started core service [/rosout]
```

```
robot@robot-pc:~$ rosparam get hello
Hello
robot@robot-pc:~$
```

```
robot@robot-pc:~$ rosparam set hello "Hello"
robot@robot-pc:~$
```

Figure 4-17. *Output of rosservice set and get*

You can get the command here.

```
Setting parameter

$ rosparam set parameter_name value
Eg. $ rosparam set hello "Hello"

Getting a parameter

$ rosparam get parameter_name
$ rosparam get hello

Output: "Hello"
```

The `roslaunch` command is also useful in ROS. If you want to run more than ten ROS nodes at time, it is very difficult to launch them one by one. In this situation, we can use `roslaunch` files to avoid this difficulty. ROS launch files are XML files in which you can insert each node that you want to run. Another advantage of the `roslaunch` command is that the `roscore` command executes with it, so we don't need to run an additional roscore command for running the nodes.

The following is the syntax for running a `roslaunch` file. The `'roslaunch'` is the command to run a launch file, along with that we have to mention package name and name of launch file.

```
$ roslaunch ros_pkg_name launch_file_name
```

`roslaunch roscpp_tutorials talker_listener.launch` is an example.

To run a ROS node, you have to use the `rosrun` node. Its usage is very simple.

```
$ rosrun ros_pkg_name node_name
```

`rosrun roscpp_tutorials talker` is an example.

ROS Demo: Hello World Example

This section demonstrates a basic ROS example. The example is already installed in ROS.

There are two nodes: *talker* and *listener*. The talker node publishes a string message. The listener node subscribes it. In this example of the process, the talker publishes a Hello World message and the listener subscribes it and prints it.

Figure 4-18 shows a diagram of the two nodes. As discussed earlier, both nodes need to communicate with the ROS master to get the information from the other node.

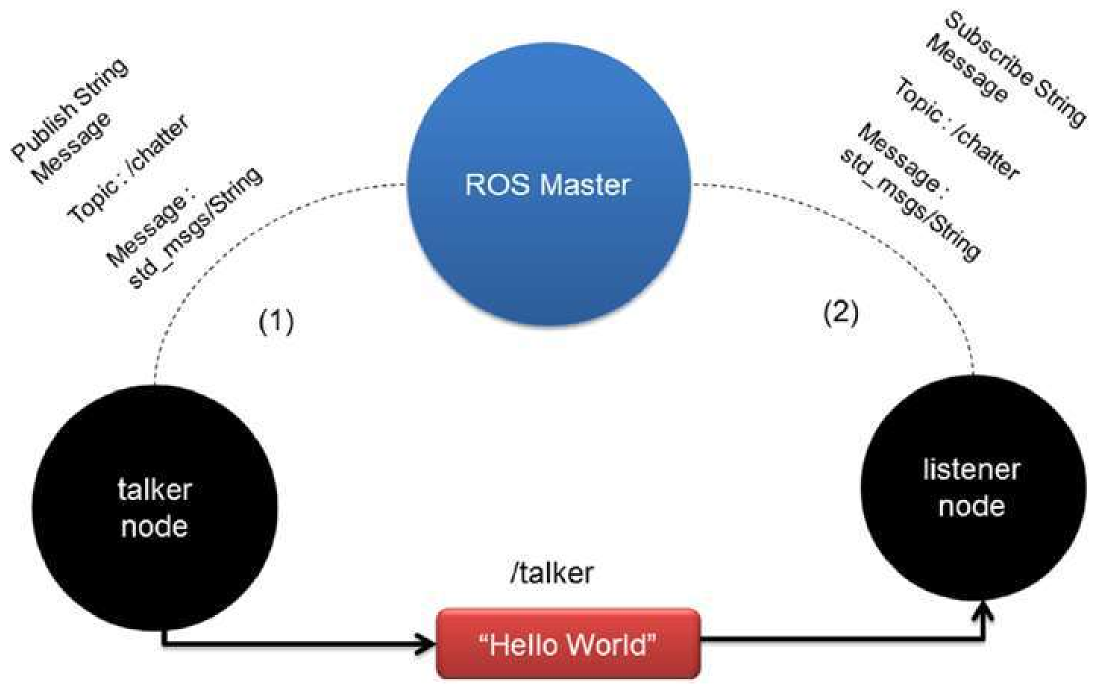


Figure 4-18. *Communication between talker and listener nodes*

Let's start the example by using following command.

The first step in starting any node in ROS is `roscore`.

```
$ roscore
```

Start the talker node by using the following command in another terminal.

```
$ rosrun roscpp_tutorials talker
```

Now you see the messages printing on the terminal screen. If you list the topic by using the following command, you see a new topic called `/chatter`.

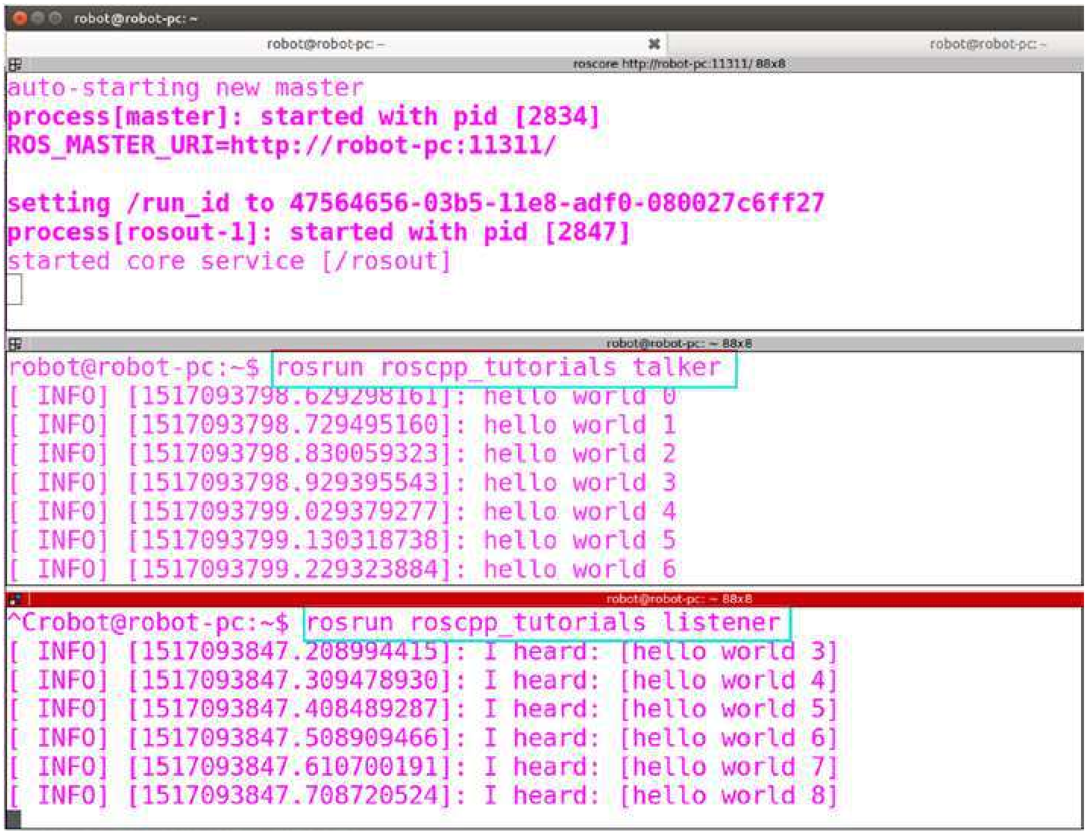
```
$ rostopic list
```

```
Output: /chatter
```

Now start the listener node by using the following command.

```
$ rosrun roscpp_tutorials listener
```

The subscribing begins between the two nodes (see Figure 4-19).



```
auto-starting new master
process[master]: started with pid [2834]
ROS_MASTER_URI=http://robot-pc:11311/

setting /run_id to 47564656-03b5-11e8-adf0-080027c6ff27
process[rosout-1]: started with pid [2847]
started core service [/rosout]
```

```
robot@robot-pc:~$ rosrun roscpp_tutorials talker
[ INFO] [1517093798.629298161]: hello world 0
[ INFO] [1517093798.729495160]: hello world 1
[ INFO] [1517093798.830059323]: hello world 2
[ INFO] [1517093798.929395543]: hello world 3
[ INFO] [1517093799.029379277]: hello world 4
[ INFO] [1517093799.130318738]: hello world 5
[ INFO] [1517093799.229323884]: hello world 6
```

```
^Crobot@robot-pc:~$ rosrun roscpp_tutorials listener
[ INFO] [1517093847.208994415]: I heard: [hello world 3]
[ INFO] [1517093847.309478930]: I heard: [hello world 4]
[ INFO] [1517093847.408489287]: I heard: [hello world 5]
[ INFO] [1517093847.508909466]: I heard: [hello world 6]
[ INFO] [1517093847.610700191]: I heard: [hello world 7]
[ INFO] [1517093847.708720524]: I heard: [hello world 8]
```

Figure 4-19. *talker-listener example*

If you want to run two of the nodes together, use the `roslaunch` command.

```
$ roslaunch roscpp_tutorials talker_listener.launch
```

`roscpp_tutorials` is an existing package in ROS and `talker_listener.launch`.

ROS Demo: turtlesim

This section demonstrates an interesting application for learning ROS concepts. The application is called turtlesim, which is a 2D simulator with a turtle in it. You can move the turtle, read the turtle's current position, and change the turtle's pattern, and so forth using ROS topics, ROS services, and parameters. When working with turtlesim, you get a better idea of how to control a robot using ROS.

The turtlesim application is already installed on ROS. You can start this application by using the following commands.

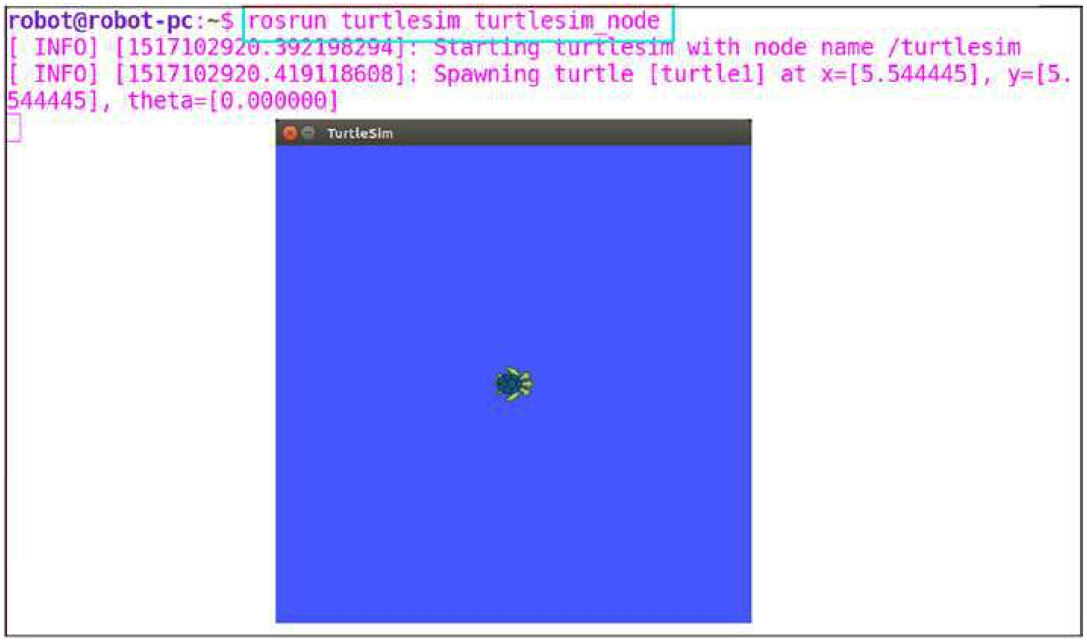
```
Starting roscore
```

```
$ roscore
```

```
Starting Turtlesim application
```

```
$ rosrun turtlesim turtlesim_node
```

A screen like the one shown in Figure 4-20 means that everything is working fine.



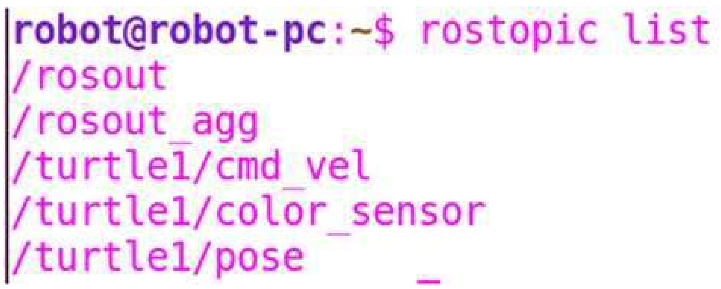
```
robot@robot-pc:~$ rosrun turtlesim turtlesim_node
[ INFO] [1517102920.392198294]: Starting turtlesim with node name /turtlesim
[ INFO] [1517102920.419118608]: Spawning turtle [turtle1] at x=[5.544445], y=[5.
544445], theta=[0.000000]
```

Figure 4-20. *Turtlesim*

Now you can open a new terminal and list the topics by publishing the turtlesim node.

```
$ rostopic list
```

You see the topics shown in Figure 4-21.

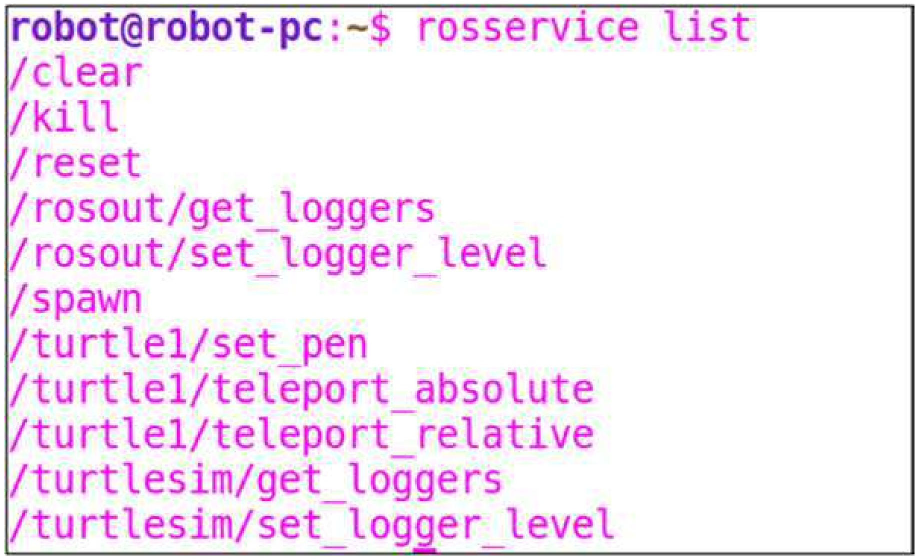


```
robot@robot-pc:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

Figure 4-21. *Turtlesim topics*

Figure 4-22 lists the services created by the turtlesim node. You can list the services by using the following command.

```
$ rosservice list
```

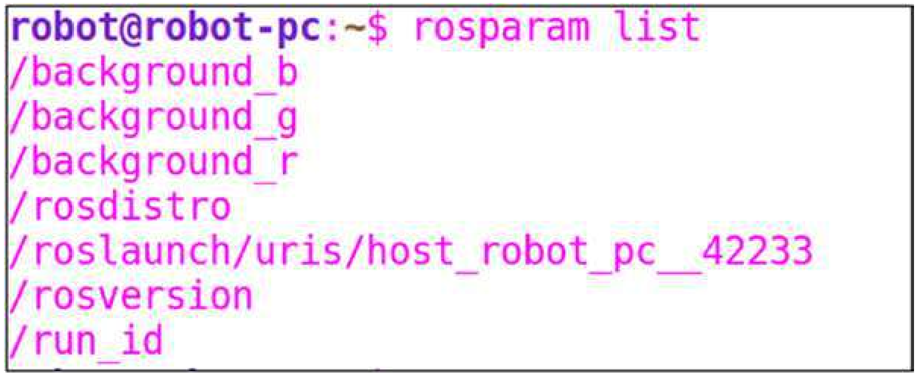


```
robot@robot-pc:~$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Figure 4-22. *List of ROS services*

List the ROS parameters by using the following command (see Figure 4-23).

```
$ rosparam list
```



```
robot@robot-pc:~$ rosparam list
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_robot_pc__42233
/rosversion
/run_id
```

Figure 4-23. *List of ROS parameters*

Moving the Turtle

If you want to move the turtle, start another ROS node by using the following command. This command has to start in another terminal.

```
$ rosrun turtlesim turtle_teleop_key
```

You can control the robot using your keyboard's arrow keys. When you press an arrow key, it publishes velocity to `/turtle1/cmd_vel`, which makes the turtle move (see Figure 4-24).

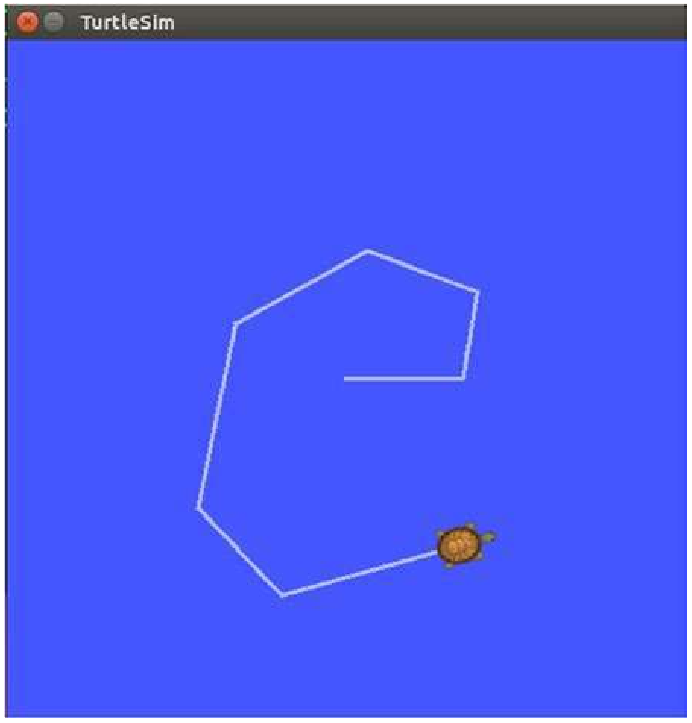


Figure 4-24. *The path that the turtle covers*

If you want to see the back end of these nodes, check the diagram in Figure 4-25. It shows the topic data going to turtlesim.

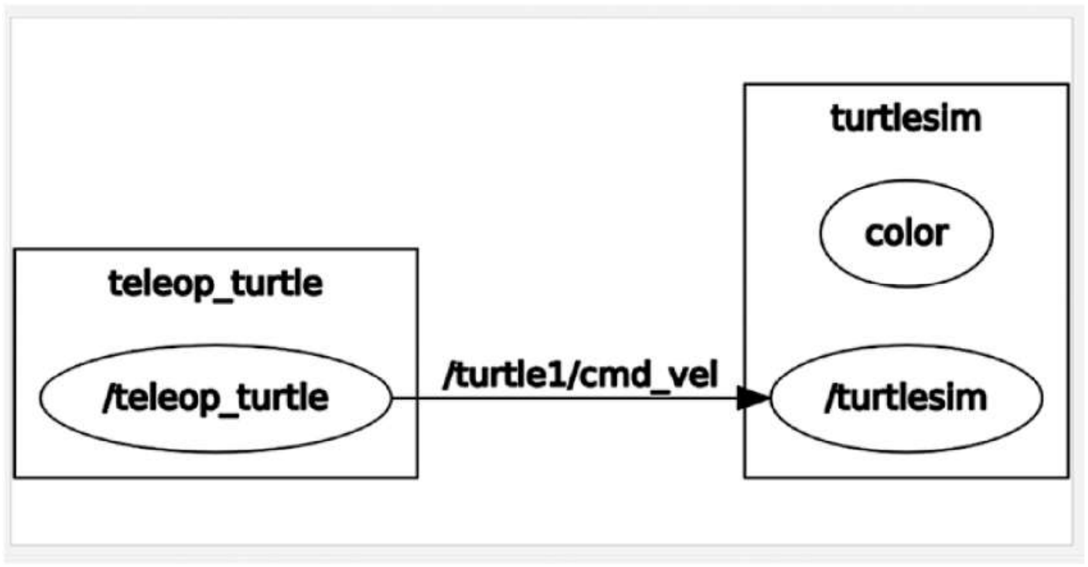


Figure 4-25. *Turtlesim and teleop node back ends*

Moving the Turtle in a Square

This section shows how to move the turtle along a square path. Close all the running nodes by pressing Ctrl+C, and start a new turtlesim session using the following command (see Figure 4-26).

```
Starting roscore
$ roscore
```

```
Starting turtlesim node
$ rosrun turtlesim turtlesim_node
```

```
Starting the node for drawing square
$ rosrun turtlesim draw_square
```

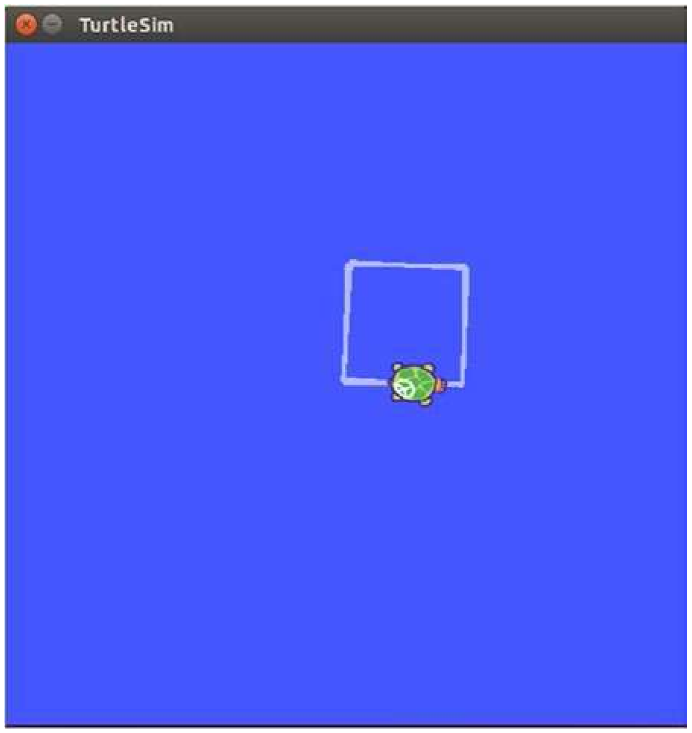


Figure 4-26. *The draw square in turtlesim*

If we want to clear the turtlesim, we can call a service called `/reset`.

```
$ rosservice call /reset
```

This resets the turtle's position.

In the next section, we look at ROS GUI tools.

ROS GUI Tools: Rviz and Rqt

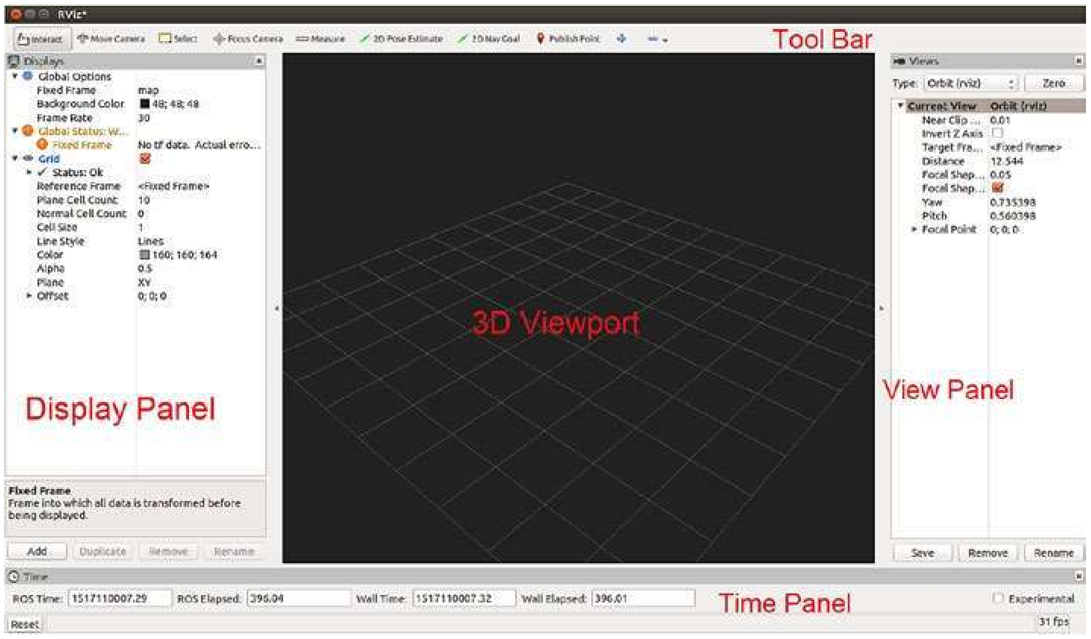
Along with command-line tools, ROS has GUI tools to visualize sensor data. A popular GUI tool is Rviz (see Figure 4-27). Using Rviz, we can visualize image data, 3D point clouds, and robot models, as well as transform data and so forth. This section explores the basics of the Rviz tool, which comes with the ROS installation.

Start Rviz using following the command.

```
Start roscore
$ roscore
```

```
Start rviz
```

```
$ rosrun rviz rviz
```



```
Global Options
  Fixed Frame            map
  Background Color       48; 48; 48
  Frame Rate             30
Global Status: W...
  Fixed Frame            No tf data. Actual erro...
Grid                     ✓
  Status: Ok
  Reference Frame        <Fixed Frame>
  Plane Cell Count       10
  Normal Cell Count      0
  Cell Size              1
  Line Style             Lines
  Color                  160; 160; 164
  Alpha                  0.5
  Plane                  XY
  Offset                 0; 0; 0
```

Fixed Frame
Frame into which all data is transformed before being displayed.

```
Type: Orbit (rviz)       Zero
Current View             Orbit (rviz)
  Near Clip ...          0.01
  Invert Z Axis          ☐
  Target Fra...          <Fixed Frame>
  Distance               12.544
  Focal Shap...          0.05
  Focal Shap...          ✓
  Yaw                    0.735398
  Pitch                  0.560398
  Focal Point            0; 0; 0
```

```
ROS Time: 1517110007.29   ROS Elapsed: 396.04   Wall Time: 1517110007.32   Wall Elapsed: 396.01   ☐ Experimental
31 fps
```

Figure 4-27. Rviz

The following describes sections in Rviz.

- *3D viewport*: The area to visualize the 3D data from sensors, robot transform data, 3D model data, and other kinds of 3D information.
- *Display panel*: Displays various kinds of sensor data.
- *View panel*: Options to view the 3D view port according to the application.
- *Toolbar*: Options for interacting with the 3D viewport, measuring robot position, setting the robot navigation goal, and changing camera view.
- *Time panel*: Features information about the ROS time and elapsed time. This time stamping may be useful for processing the sensor data.
- *Rqt*: Features options to visualize 2D data, logging topics, publishing topics, calling services, and more.

This is how to start the Rqt GUI.

```
Start roscore

$ roscore

Start rqt_gui
$ rosrun rqt_gui rqt_gui
```

You get an empty GUI with some menus. You can add your own plug-ins from the drop-down menu. Figure 4-28 is a screenshot of rqt_gui loaded with a plug-in.

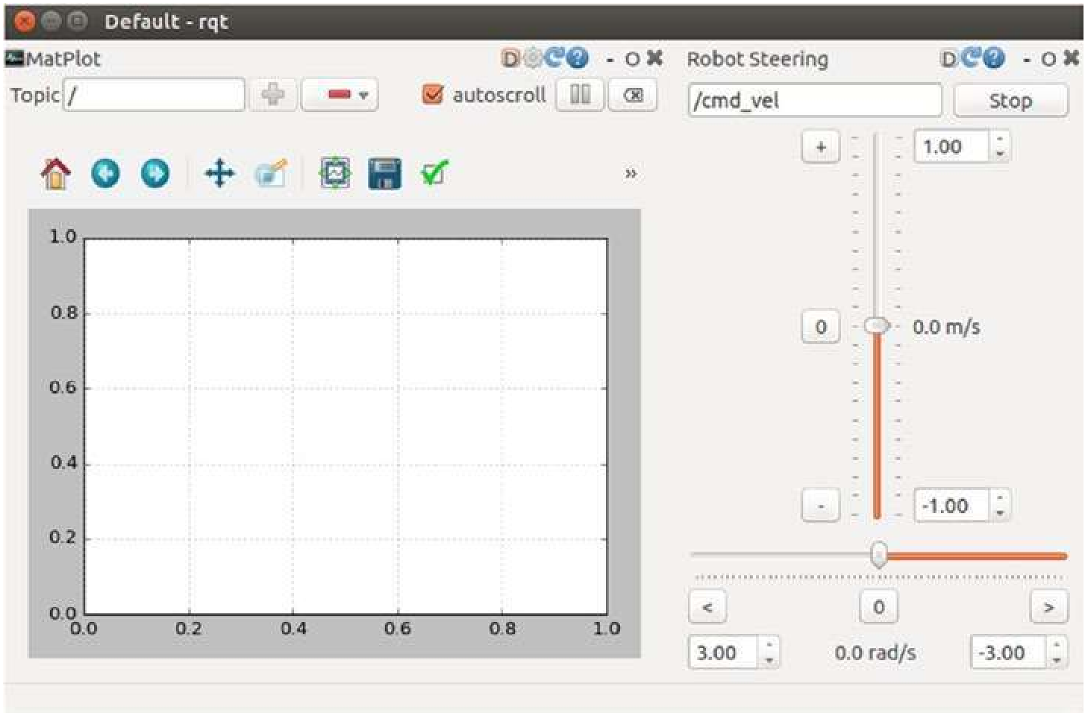


Figure 4-28. *The Rqt GUI*

Summary

This chapter discussed the fundamentals of the Robot Operating System. It started with robot programming and explained why it is different from other software applications. Next, we looked at the different operating system platforms that can install ROS and covered the detailed installation instructions for Ubuntu. We saw different robots and sensors compatible with ROS, and we discussed the ROS architecture. We also looked at important ROS concepts and a simulator called turtlesim. In the end, we became familiar with ROS GUI tools such as Rqt and Rviz.

In the next chapter, we see how to program using ROS and how to create ROS applications using C++ and Python.