Mastering ROS for Robotics Programming

Mastering ROS for Robotics Programming discusses advanced concepts in robotics and how to program using ROS. Over the course of the book, you will learn how to build models of complex robots and simulate and interface with the robot using the ROS MoveIt! motion planning library and ROS navigation stacks.

One of the essential ingredients of robots is the vision sensor, and an entire chapter is dedicated to the vision sensor, its interfacing with ROS, and its programming. This book discusses the hardware interfacing and simulation of complex robot to ROS and also discusses about ROS Industrial (Package used for interfacing industrial robots).

Finally, you will get to know the best practices to follow when programming using ROS.

Who this book is written for

If you are a robotics enthusiast or researcher who wants to learn more about building robot applications using ROS, this book is for you. In order to learn from this book, you should have a basic knowledge of ROS, GNU/Linux, and C++ programming concepts. The book will also be good for programmers who want to explore the advanced features of ROS.

What you will learn from this book

- Create a robot model of a seven DOF arm and a differential-wheeled robot
- Work with motion planning of a Seven-DOF arm using MoveIt!
- Implement autonomous navigation in differential drive robot using SLAM and AMCL
- Dig deep into the Plugin lib, ROS nodelets, and Gazebo plugins
- Interface I/O boards, such as Arduino, sensors, and actuators, with ROS
- Work with ABB and Universal arm simulation
- Explore the ROS framework using the latest version of ROS

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 *Introduction to ROS and Its Package Management*
- A synopsis of the book’s content
- More information on **Mastering ROS for Robotics Programming**
Lentin Joseph is an author, entrepreneur, electronics engineer, robotics enthusiast, machine vision expert, embedded programmer, and the founder and CEO of Qbotics Labs (http://www.qboticslabs.com) from India. He completed his bachelor's degree in electronics and communication engineering at the Federal Institute of Science and Technology (FISAT), Kerala. For his final year engineering project, he made a social robot that can interact with people. The project was a huge success and was mentioned in many forms of visual and print media. The main features of this robot were that it can communicate with people and reply intelligently and has some image processing capabilities such as face, motion, and color detection. The entire project was implemented using the Python programming language. His interest in robotics, image processing, and Python started with that project.

After his graduation, for 3 years he worked in a start-up company focusing on robotics and image processing. In the meantime, he learned famous robotic software platforms such as Robot Operating System (ROS), V-REP, Actin (a robotic simulation tool), and image processing libraries such as OpenCV, OpenNI, and PCL. He also knows robot 3D designing and embedded programming on Arduino and Tiva Launchpad.

After 3 years of work experience, he started a new company called Qbotics Labs, which mainly focuses on research to build up some great products in domains such as robotics and machine vision. He maintains a personal website (http://www.lentinjoseph.com) and a technology blog called technolabsz (http://www.technolabsz.com). He publishes his works on his tech blog. He was also a speaker at PyCon2013, India, on the topic *Learning Robotics using Python*.
Lentin is the author of the book *Learning Robotics Using Python* (refer to [http://learn-robotics.com](http://learn-robotics.com) to know more) by Packt Publishing. The book was about building an autonomous mobile robot using ROS and OpenCV. The book was launched in ICRA 2015 and was featured in the ROS blog, Robohub, OpenCV, the Python website, and various other such forums.

Lentin was a finalist in the ICRA 2015 challenge, HRATC ([http://www2.isr.uc.pt/~embedded/events/HRATC2015/Welcome.html](http://www2.isr.uc.pt/~embedded/events/HRATC2015/Welcome.html)).
Preface

Mastering ROS for Robotics Programming is an advanced guide of ROS that is very suitable for readers who already have a basic knowledge in ROS. ROS is widely used in robotics companies, universities, and robotics research institutes for designing, building, and simulating a robot model and interfacing it into real hardware. ROS is now an essential requirement for Robotic engineers; this guide can help you acquire knowledge of ROS and can also help you polish your skills in ROS using interactive examples. Even though it is an advanced guide, you can see the basics of ROS in the first chapter to refresh the concepts. It also helps ROS beginners. The book mainly focuses on the advanced concepts of ROS, such as ROS Navigation stack, ROS MoveIt!, ROS plugins, nodelets, controllers, ROS Industrial, and so on.

You can work with the examples in the book without any special hardware; however, in some sections you can see the interfacing of I/O boards, vision sensors, and actuators to ROS. To work with this hardware, you will need to buy it.

The book starts with an introduction to ROS and then discusses how to build a robot model in ROS for simulating and visualizing. After the simulation of robots using Gazebo, we can see how to connect the robot to Navigation stack and MoveIt!. In addition to this, we can see ROS plugins, controllers, nodelets, and interfacing of I/O boards and vision sensors. Finally, we can see more about ROS Industrial and troubleshooting and best practices in ROS.

What this book covers

Chapter 1, Introduction to ROS and Its Package Management, gives you an understanding of the core underlying concepts of ROS and how to work with ROS packages.

Chapter 2, Working with 3D Robot Modeling in ROS, discusses the design of two robots; one is a seven-DOF (Degree of Freedom) manipulator and the other is a differential drive robot.
Preface

Chapter 3, Simulating Robots Using ROS and Gazebo, discusses the simulation of seven-DOF arms, differential wheeled robots, and ROS controllers that help control robot joints in Gazebo.

Chapter 4, Using the ROS MoveIt! and Navigation Stack, interfaces out-of-the-box functionalities such as robot manipulation and autonomous navigation using ROS MoveIt! and Navigation stack.

Chapter 5, Working with Pluginlib, Nodelets, and Gazebo Plugins, shows some of the advanced concepts in ROS, such as ROS pluginlib, nodelets, and Gazebo plugins. We will discuss the functionalities and application of each concept and can practice one example to demonstrate its working.

Chapter 6, Writing ROS Controllers and Visualization Plugins, shows how to write a basic ROS controller for PR2 robots and robots similar to PR2. After creating the controller, we will run the controller using the PR2 simulation in Gazebo. We can also see how to create plugin for RViz.

Chapter 7, Interfacing I/O Boards, Sensors, and Actuators to ROS, discusses interfacing some hardware components, such as sensors and actuators, with ROS. We will see the interfacing of sensors using I/O boards, such as Arduino, Raspberry Pi, and Odroid-C1, with ROS.

Chapter 8, Programming Vision Sensors using ROS, Open-CV, and PCL, discusses how to interface various vision sensors with ROS and program it using libraries such as Open Source Computer Vision (Open-CV) and Point Cloud Library (PCL).

Chapter 9, Building and Interfacing Differential Drive Mobile Robot Hardware in ROS, helps you to build autonomous mobile robot hardware with differential drive configuration and interface it with ROS. This chapter aims at giving you an idea of building a custom mobile robot and interfacing it with ROS.

Chapter 10, Exploring the Advanced Capabilities of ROS-MoveIt!, discusses the capabilities of MoveIt! such as collision avoidance, perception using 3D sensors, grasping, picking, and placing. After that, we can see the interfacing of a robotic manipulator hardware with MoveIt!

Chapter 11, ROS for Industrial Robots, helps you understand and install ROS-Industrial packages in ROS. We can see how to develop an MoveIt! IKFast plugin for an industrial robot.

Chapter 12, Troubleshooting and Best Practices in ROS, discusses how to set the ROS development environment in Eclipse IDE, best practices in ROS, and troubleshooting tips in ROS.
Introduction to ROS and Its Package Management

This is an introductory chapter that gives you an understanding of the core underlying concepts of ROS and how to work with ROS packages. We will also go through the ROS concepts such as ROS master, nodes, parameter server, topic, message, service, and actionlib to refresh your memory of the concepts you already know.

The basic building blocks of the ROS software framework are ROS packages. We will see how to create, build, and maintain a ROS package. We will also see how to create a wiki page for our package on the ROS website to contribute to the ROS community.

In this chapter, we will cover the following topics:

- Why should we learn ROS?
- Why we prefer ROS for robot
- Why we do not prefer ROS for robot
- Understanding the ROS file system level
- Understanding the ROS computation graph level
- Understanding ROS nodes, messages, topics, services, bags
- Understanding ROS Master
- Using ROS Parameter
- Understanding ROS community level
- Running ROS Master and ROS Parameter server
- Creating a ROS package
- Working with ROS topics
Introduction to ROS and Its Package Management

- Adding custom `msg` and `srv` files
- Working with ROS services
- Working with ROS actionlib
- Creating launch files
- Applications of topics, services, and actionlib
- Maintaining the ROS package
- Releasing your ROS package
- Creating a wiki page for your ROS package

**Why should we learn ROS?**

The Robot Operating System (ROS) is a trending robot application development platform that provides various features such as message passing, distributed computing, code reusing, and so on.

The ROS project was started in 2007 with the name Switchyard by Morgan Quigley (http://wiki.osrfoundation.org/morgan) as part of the Stanford STAIR robot project. The main development of ROS happened at Willow Garage (https://www.willowgarage.com/).

The ROS community is growing very fast and there are many users and developers worldwide. Most of the high-end robotics companies are now porting their software to ROS. This trend is also visible in industrial robotics, in which companies are switching from proprietary robotic application to ROS.

The ROS industrial movement has gained momentum in the past few years owing to the large amount of research done in that field. ROS Industrial can extend the advanced capabilities of ROS to manufacturing. The increasing applications of ROS can generate a lot of job opportunities in this field. So after some years, knowledge in ROS will be an essential requirement for a robotics engineer.

**Why we prefer ROS for robots**

Imagine that we are going to build an autonomous mobile robot. Here are some of the reasons why people choose ROS over other robotic platforms such as Player, YARP, Orocos, MRPT, and so on:
• **High-end capabilities:** ROS comes with ready to use capabilities, for example, SLAM (Simultaneous Localization and Mapping) and AMCL (Adaptive Monte Carlo Localization) packages in ROS can be used for performing autonomous navigation in mobile robots and the MoveIt package for motion planning of robot manipulators. These capabilities can directly be used in our robot software without any hassle. These capabilities are its best form of implementation, so writing new code for existing capabilities are like reinventing wheels. Also, these capabilities are highly configurable; we can fine-tune each capability using various parameters.

• **Tons of tools:** ROS is packed with tons of tools for debugging, visualizing, and performing simulation. The tools such as rqt_gui, RViz and Gazebo are some of the strong open source tools for debugging, visualization, and simulation. The software framework that has these many tools is very rare.

• **Support high-end sensors and actuators:** ROS is packed with device drivers and interface packages of various sensors and actuators in robotics. The high-end sensors include Velodyne-LIDAR, Laser scanners, Kinect, and so on and actuators such as Dynamixel servos. We can interface these components to ROS without any hassle.

• **Inter-platform operability:** The ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.

• **Modularity:** One of the issues that can occur in most of the standalone robotic applications are, if any of the threads of main code crash, the entire robot application can stop. In ROS, the situation is different, we are writing different nodes for each process and if one node crashes, the system can still work. Also, ROS provides robust methods to resume operation even if any sensors or motors are dead.

• **Concurrent resource handling:** Handling a hardware resource by more than two processes is always a headache. Imagine, we want to process an image from a camera for face detection and motion detection, we can either write the code as a single entity that can do both, or we can write a single threaded code for concurrency. If we want to add more than two features in threads, the application behavior will get complex and will be difficult to debug. But in ROS, we can access the devices using ROS topics from the ROS drivers. Any number of ROS nodes can subscribe to the image message from the ROS camera driver and each node can perform different functionalities. It can reduce the complexity in computation and also increase the debug-ability of the entire system.
• **Active community:** When we choose a library or software framework, especially from an open source community, one of the main factors that needs to be checked before using it is its software support and developer community. There is no guarantee of support from an open source tool. Some tools provide good support and some tools don’t. In ROS, the support community is active. There is a web portal to handle the support queries from the users too (http://answers.ros.org). It seems that the ROS community has a steady growth in developers worldwide.

There are many reasons to choose ROS other than the preceding points.

Next, we can check the various reasons why people don’t use ROS. Here are some of the existing reasons.

### Why some do not prefer ROS for robots

Here are some of the reasons why people do not prefer ROS for their robotic projects:

• **Difficulty in learning:** It will be difficult to learn ROS from their default wiki pages. Most users depend on books to start with ROS. Even this book covers only the basics; learning ROS is going to be bit difficult.

• **Difficulties in starting with simulation:** The main simulator in ROS is Gazebo. Even though Gazebo works well, to get started with Gazebo is not an easy task. The simulator has no inbuilt features to program. Complete simulation is done only through coding in ROS. When we compare Gazebo with other simulators such as V-REP and Webots, they have inbuilt functionalities to prototype and program the robot. They also have a rich GUI toolset and support a wide variety of robots and have ROS interfaces too. These tools are proprietary, but can deliver a decent job. The toughness of learning simulation using Gazebo and ROS is a reason for not using it in their projects.

• **Difficulties in robot modeling:** The robot modeling in ROS is performed using URDF, which is an XML based robot description. In short, we need to write the robot model as a description using URDF tags. In V-REP, we can directly build the 3D robot model in the GUI itself, or we can import the mesh. In ROS, we should write the robot model definitions using URDF tags. There is a SolidWorks plugin to convert a 3D model from SolidWorks to URDF. But if we use other 3D CAD tools, there are no options at all. Learning to model a robot in ROS will take a lot of time and building using URDF tags is also time consuming compared to other simulators.
• **Need for a computer**: We always need a computer to run ROS. Small robots that work completely on microcontrollers don't require a ROS system. ROS is only required when we want to perform high-level functionalities such as autonomous navigation and motion planning. In basic robots, there is no need to use ROS if you are not planning higher level functionalities on the robot.

• **ROS in commercial robot products**: When we deploy ROS on a commercial product, a lot of things need to be taken care of. One thing is the code quality. ROS codes follow a standard coding style and keep best practices for maintaining the code too. We have to check whether it satisfies the quality level required for our product. We might have to do additional work to improve the quality of code. Most of the code in ROS is contributed by researchers from universities, so if we are not satisfied with the ROS code quality, it is better to write your own code, which is specific to the robot and only use the ROS core functionalities if required.

We now know where we have to use ROS and where we do not. If ROS is really required for your robot, let's start discussing ROS in more detail. First, we can see the underlying core concepts of ROS. There are mainly three levels in ROS: file system level, computation graph level, and community level. We can have a look at each level in short.

**Understanding the ROS file system level**

Similar to an operating system, ROS files are also organized on the hard disk in a particular fashion. In this level, we can see how these files are organized on the disk. The following graph shows how ROS files and folder are organized on the disk:

![Figure 1: ROS File system level](image-url)
Here are the explanations of each block in the file system

- **Packages**: The ROS packages are the most basic unit of the ROS software. It contains the ROS runtime process (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.

- **Package manifest**: The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The `package.xml` file inside the ROS package is the manifest file of that package.

- **Meta packages**: The term meta package is used for a group of packages for a special purpose. In an older version of ROS such as Electric and Fuerte, it was called stacks, but later it was removed, as simplicity and meta packages came to existence. One of the examples of a meta package is the ROS navigation stack.

- **Meta packages manifest**: The meta package manifest is similar to the package manifest; differences are that it might include packages inside it as runtime dependencies and declare an export tag.

- **Messages (.msg)**: The ROS messages are a type of information that is sent from one ROS process to the other. We can define a custom message inside the `msg` folder inside a package (`my_package/msg/MyMessageType.msg`). The extension of the message file is `.msg`.

- **Services (.srv)**: The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined inside the `srv` folder inside the package (`my_package/srv/MyServiceType.srv`).

- **Repositories**: Most of the ROS packages are maintained using a Version Control System (VCS) such as Git, subversion (svn), mercurial (hg), and so on. The collection of packages that share a common VCS can be called repositories. The package in the repositories can be released using a catkin release automation tool called `bloom`. 
The following screenshot gives you an idea of files and folders of a package that we are going to make in the upcoming sections:

**Figure 2**: List of files inside the exercise package

**ROS packages**

A typical structure of a ROS package is shown here:

**Figure 3**: Structure of a typical ROS package
We can discuss the use of each folder as follows:

- **config**: All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and is a common practice to name the folder `config` to keep the configuration files in it.
- **include/package_name**: This folder consists of headers and libraries that we need to use inside the package.
- **scripts**: This folder keeps executable Python scripts. In the block diagram, we can see two example scripts.
- **src**: This folder stores the C++ source codes. We can see two examples of the source code in the block diagram.
- **launch**: This folder keeps the launch files that are used to launch one or more ROS nodes.
- **msg**: This folder contains custom message definitions.
- **srv**: This folder contains the service definitions.
- **action**: This folder contains the action definition. We will see more about actionlib in the upcoming sections.
- **package.xml**: This is the package manifest file of this package.
- **CMakeLists.txt**: This is the CMake build file of this package.

We need to know some commands to create, modify, and work with the ROS packages. Here are some of the commands used to work with ROS packages:

- **catkin_create_pkg**: This command is used to create a new package
- **rospack**: This command is used to get information about the package in the file system
- **catkin_make**: This command is used to build the packages in the workspace
- **rosdep**: This command will install the system dependencies required for this package

To work with packages, ROS provides a bash-like command called rosbash (http://wiki.ros.org/rosbash), which can be used to navigate and manipulate the ROS package. Here are some of the rosbash commands:

- **roscd**: This command is used to change the package folder. If we give the argument a package name, it will switch to that package folder.
- **roscp**: This command is used to copy a file from a package.
- **rosed**: This command is used to edit a file.
- **rosrun**: This command is used to run an executable inside a package.
The definition of package.xml of a typical package is shown as follows:

```xml
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.0</version>
  <description>The hello world package</description>
  <maintainer email="qboticslab@gmail.com">Lentin Joseph</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/hello_world</url>
  <author email="qboticslab@gmail.com">Lentin Joseph</author>
  <build_depend>/catkin/buildtool</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>ros</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>ros</run_depend>
  <run_depend>std_msgs</run_depend>
  <export>
    <metapackage/>
  </export>
</package>
```

Figure 4: Structure of package.xml

The package.xml file consists of the package name, version of the package, the package description, author details, package build dependencies, and runtime dependencies. The `<build_depend>` tag includes the packages that are necessary to build the source code of the package. The packages inside the `<run_depend>` tag are necessary during runtime of the package node.

**ROS meta packages**

Meta packages are specialized packages in ROS that only contain one file, that is, a package.xml file. It doesn't contain folders and files similar to a normal package.

Meta packages simply group a set of multiple packages as a single logical package. In the package.xml file, the meta package contains an export tag, as shown here:

```xml
<export>
  <metapackage/>
</export>
```

Also, in meta packages, there are no `<buildtool_depend>` dependencies for `catkin`, there are only `<run_depend>` dependencies, which are the packages grouped in the meta package.
The ROS navigation stack is a good example of meta packages. If ROS is installed, we can try the following command, by switching to the navigation meta package folder:

```
$ roscd navigation
```

Open `package.xml` using `gedit` text editor

```
$ gedit package.xml
```

This is a lengthy file; here is a stripped down version of it:

```xml
<package>
  <name>navigation</name>
  <version>1.12.2</version>
  ........
  <buildtool_depend>catkin</buildtool_depend>
  ........
  <run_depend>amcl</run_depend>
  <run_depend>carrot_planner</run_depend>
  ........
  <export>
    <metapackage/>
  </export>
</package>
```

Figure 5: Structure of meta-package package.xml

**ROS messages**

The ROS nodes can publish data having a particular type. The types of data are described using a simplified message description language, also called ROS messages. These datatype descriptions can be used to generate source code for the appropriate message type in different target languages.

The data type description of ROS messages are stored in `.msg` files in the `msg` subdirectory of a ROS package.

The message definition can consist of two types: fields and constants. The field is split into field types and field name. Field types is the data type of the transmitting message and field name is the name of it. The constants define a constant value in the message file.

Here is an example of message definitions:

```plaintext
int32 number
string name
float32 speed
```
Here, the first part is the field type and second is the field name. The field type is the data type and the field name can be used to access the value from the message. For example, we can use `msg.number` for accessing the value of the number from the message.

Here is a table to show some of the built-in field types that we can use in our message:

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Serialization</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool(1)</code></td>
<td>unsigned 8-bit int</td>
<td><code>uint8_t(2)</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>int8</code></td>
<td>signed 8-bit int</td>
<td><code>int8_t</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>uint8</code></td>
<td>unsigned 8-bit int</td>
<td><code>uint8_t</code></td>
<td><code>int (3)</code></td>
</tr>
<tr>
<td><code>int16</code></td>
<td>signed 16-bit int</td>
<td><code>int16_t</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>uint16</code></td>
<td>unsigned 16-bit int</td>
<td><code>uint16_t</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>int32</code></td>
<td>signed 32-bit int</td>
<td><code>int32_t</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>uint32</code></td>
<td>unsigned 32-bit int</td>
<td><code>uint32_t</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>int64</code></td>
<td>signed 64-bit int</td>
<td><code>int64_t</code></td>
<td><code>long</code></td>
</tr>
<tr>
<td><code>uint64</code></td>
<td>unsigned 64-bit int</td>
<td><code>uint64_t</code></td>
<td><code>long</code></td>
</tr>
<tr>
<td><code>float32</code></td>
<td>32-bit IEEE float</td>
<td><code>float</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>float64</code></td>
<td>64-bit IEEE float</td>
<td><code>double</code></td>
<td><code>float</code></td>
</tr>
<tr>
<td><code>string</code></td>
<td>ascii string(4)</td>
<td><code>std::string</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td><code>time</code></td>
<td>secs/nsecs unsigned 32-bit ints</td>
<td><code>ros::Time</code></td>
<td><code>rospy.Time</code></td>
</tr>
<tr>
<td><code>duration</code></td>
<td>secs/nsecs signed 32-bit ints</td>
<td><code>ros::Duration</code></td>
<td><code>rospy.Duration</code></td>
</tr>
</tbody>
</table>

A special type of ROS message is called message headers. Headers can carry information such as time, frame of reference or `frame_id`, and sequence number. Using headers, we will get numbered messages and more clarity in who is sending the current message. The header information is mainly used to send data such as robot joint transforms (TF). Here is an example of the message header:

```cpp
uint32 seq
time stamp
string frame_id
```

The `rosmag` command tool can be used to inspect the message header and the field types. The following command helps to view the message header of a particular message:

```bash
$ rosmag show std_msgs/Header
```
Introduction to ROS and Its Package Management

This will give you an output like the preceding example message header. We can see more about the `rosmsg` command and how to work with custom message definitions in the upcoming sections.

The ROS services

The ROS services are a type request/response communication between ROS nodes. One node will send a request and wait until it gets a response from the other. The request/response communication is also using the ROS message description.

Similar to the message definitions using the `.msg` file, we have to define the service definition in another file called `.srv`, which has to be kept inside the `srv` subdirectory of the package. Similar to the message definition, a service description language is used to define the ROS service types.

An example service description format is as follows:

```
#Request message type
string str
---
#Response message type
string str
```

The first section is the message type of request that is separated by `---` and in the next section is the message type of response. In these examples, both Request and Response are strings.

In the upcoming sections, we can see how to work with ROS services.

Understanding the ROS computation graph level

The computation in ROS is done using a network of process called ROS nodes. This computation network can be called the computation graph. The main concepts in the computation graph are ROS Nodes, Master, Parameter server, Messages, Topics, Services, and Bags. Each concept in the graph is contributed to this graph in different ways.

The ROS communication related packages including core client libraries such as roscpp and rospython and the implementation of concepts such as topics, nodes, parameters, and services are included in a stack called ros_comm (http://wiki.ros.org/ros_comm).
This stack also consists of tools such as rostopic, rosparm, rosservice, and rosnodes to introspect the preceding concepts.

The ros_comm stack contains the ROS communication middleware packages and these packages are collectively called ROS Graph layer.

![Structure of the ROS Graph layer](image)

The following are abstracts of each graph's concepts:

- **Nodes**: Nodes are the process that perform computation. Each ROS node is written using ROS client libraries such as roscpp and rospy. Using client library APIs, we can implement different types of communication methods in ROS nodes. In a robot, there will be many nodes to perform different kinds of tasks. Using the ROS communication methods, it can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all functionality. Being a simple structure, ROS nodes are easy to debug too.

- **Master**: The ROS Master provides name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.

- **Parameter Server**: The parameter server allows you to keep the data to be stored in a central location. All nodes can access and modify these values. Parameter server is a part of ROS Master
• **Messages**: Nodes communicate with each other using messages. Messages are simply a data structure containing the typed field, which can hold a set of data and that can be sent to another node. There are standard primitive types (integer, floating point, Boolean, and so on) and these are supported by ROS messages. We can also build our own message types using these standard types.

• **Topics**: Each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic. When a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence. We can even subscribe a topic that might not have any publisher. In short, the production of information and consumption of it are decoupled. Each topic has a unique name, and any node can access this topic and send data through it as long as they have the right message type.

• **Services**: In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction. ROS Services are used in these cases. We can define a service definition that contains two parts; one is for requests and the other is for responses. Using ROS Services, we can write a server node and client node. The server node provides the service under a name, and when the client node sends a request message to this server, it will respond and send the result to the client. The client might need to wait until the server responds. The ROS service interaction is like a remote procedure call.

• **Bags**: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms. Bags are very useful features when we work with complex robot mechanisms.

The following graph shows how the nodes communicate with each other using topics. The topics are mentioned in a rectangle and nodes are represented in ellipses. The messages and parameters are not included in this graph. These kinds of graphs can be generated using a tool called rqt_graph (http://wiki.ros.org/rqt_graph).
Understanding ROS nodes

ROS nodes are a process that perform computation using ROS client libraries such as roscpp and rospy. One node can communicate with other nodes using ROS Topics, Services, and Parameters.

A robot might contain many nodes, for example, one node processes camera images, one node handles serial data from the robot, one node can be used to compute odometry, and so on.

Using nodes can make the system fault tolerant. Even if a node crashes, an entire robot system can still work. Nodes also reduce the complexity and increase debug-ability compared to monolithic codes because each node is handling only a single function.

All running nodes should have a name assigned to identify them from the rest of the system. For example, /camera_node could be a name of a node that is broadcasting camera images.

There is a rosbash tool to introspect ROS nodes. The rosnodes command can be used to get information about a ROS node. Here are the usages of rosnodes:

- `$ rosnodes info [node_name]`: This will print the information about the node
- `$ rosnodes kill [node_name]`: This will kill a running node
- `$ rosnodes list`: This will list the running nodes
- `$ rosnodes machine [machine_name]`: This will list the nodes running on a particular machine or a list of machines
- `$ rosnodes ping`: This will check the connectivity of a node
- `$ rosnodes cleanup`: This will purge the registration of unreachable nodes

Figure 7: Graph of communication between nodes using topics
We will see example nodes using the roscpp client and will discuss the working of ROS nodes that use functionalities such ROS Topics, Service, Messages, and actionlib.

**ROS messages**
ROS nodes communicate with each other by publishing messages to a topic. As we discussed earlier, messages are a simple data structure containing field types. The ROS message supports standard primitive datatypes and arrays of primitive types.

Nodes can also exchange information using service calls. Services are also messages, the service message definitions are defined inside the srv file.

We can access the message definition using the following method. For example, to access std_msgs/msg/String.msg, we can use std_msgs/String. If we are using the roscpp client, we have to include std_msgs/String.h for the string message definition.

In addition to message data type, ROS uses an MD5 checksum comparison to confirm whether the publisher and subscriber exchange the same message data types.

ROS has inbuilt tools called rosmsg to get information about ROS messages. Here are some parameters used along with rosmsg:

- `$ rosmsg show [message]`: This shows the message description
- `$ rosmsg list`: This lists all messages
- `$ rosmsg md5 [message]`: This displays md5sum of a message
- `$ rosmsg package [package_name]`: This lists messages in a package
- `$ rosmsg packages [package_1] [package_2]`: This lists packages that contain messages

**ROS topics**
ROS topics are named buses in which ROS nodes exchange messages. Topics can anonymously publish and subscribe, which means that the production of messages is decoupled from the consumption. The ROS nodes are not interested to know which node is publishing the topic or subscribing topics, it only looks for the topic name and whether the message types of publisher and subscriber are matching.

The communication using topics are unidirectional, if we want to implement request/response such as communication, we have to switch to ROS services.
The ROS nodes communicate with topics using TCP/IP-based transport known as TCPROS. This method is the default transport method used in ROS. Another type of communication is UDPROS, which has low-latency, loose transport, and is only suited for teleoperation.

The ROS topic tool can be used to get information about ROS topics. Here is the syntax of this command:

- `$ rostopic bw /topic`: This command will display the bandwidth used by the given topic
- `$ rostopic echo /topic`: This command will print the content of the given topic
- `$ rostopic find /message_type`: This command will find topics using the given message type
- `$ rostopic hz /topic`: This command will display the publishing rate of the given topic
- `$ rostopic info /topic`: This command will print information about an active topic
- `$ rostopic list`: This command will list all active topics in the ROS system
- `$ rostopic pub /topic message_type args`: This command can be used to publish a value to a topic with a message type
- `$ rostopic type /topic`: This will display the message type of the given topic

**ROS services**

When we need a request/response kind of communication in ROS, we have to use the ROS services. ROS topics can't do this kind of communication because it is unidirectional. ROS services are mainly used in a distributed system.

The ROS services is defined using a pair of messages. We have to define a request datatype and a response datatype in a srv file. The srv files are kept in a srv folder inside a package.

In ROS services, one node acts as a ROS server in which the service client can request the service from the server. If the server completes the service routine, it will send the results to the service client.

The ROS service definition can be accessed by the following method, for example, if `my_package/srv/Image.srv` can be accessed by `my_package/Image`. 
In ROS services also, there is an MD5 checksum that checks in the nodes. If the sum is equal, then only the server responds to the client.

There are two ROS tools to get information about the ROS service. The first tool is rossrv, which is similar to rosmat, and is used to get information about service types. The next command is rosservice, which is used to list and query about the running ROS services.

The following explain how to use the rosservice tool to get information about the running services:

- `$ rosservice call /service args`: This tool will call the service using the given arguments
- `$ rosservice find service_type`: This command will find services in the given service type
- `$ rosservice info /services`: This will print information about the given service
- `$ rosservice list`: This command will list the active services running on the system
- `$ rosservice type /service`: This command will print the service type of a given service
- `$ rosservice uri /service`: This tool will print the service ROSRPC URI

**ROS bags**

A bag file in ROS is for storing ROS message data from topics and services. The .bag extension is used to represent a bag file.

Bag files are created using the rosbag command, which will subscribe one or more topics and store the message's data in a file as it's received. This file can play the same topics as they are recorded from or it can remap the existing topics too.

The main application of rosbag is data logging. The robot data can be logged and can visualize and process offline.

The rosbag command is used to work with rosbag files. Here are the commands to record and playback a bag file:

- `$ rosbag record [topic_1] [topic_2] -o [bag_name]`: This command will record the given topics into a bag file that is given in the command. We can also record all topics using the -a argument.
- `$ rosbag play [bag_name]`: This will playback the existing bag file.
Here are more details about this command:

http://wiki.ros.org/rosbag/Commandline

There is a GUI tool to handle record and playback of bag files called rqt_bag.
Go to the following link to know more about rqt_bag:

http://wiki.ros.org/rqt_bag

**Understanding ROS Master**

ROS Master is much like a DNS server. When any node starts in the ROS system, it will start looking for ROS Master and register the name of the node in it. So ROS Master has the details of all nodes currently running on the ROS system. When any details of the nodes change, it will generate a callback and update with the latest details. These node details are useful for connecting with each node.

When a node starts publishing a topic, the node will give the details of the topic such as name and data type to ROS Master. ROS Master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, ROS Master will share the node details of the publisher to the subscriber node. After getting the node details, these two nodes will interconnect using the TCPROS protocol, which is based on TCP/IP sockets. After connecting to the two nodes, ROS Master has no role in controlling them. We might be able to stop either the publisher node or the subscriber node according to our wish. If we stop any nodes, it will check with ROS Master once again. This same method is for the ROS services.

The nodes are written using the ROS client libraries such as roscpp and rospy. These clients interact with ROS Master using XMLRPC (XML Remote Procedure Call) based APIs, which act as the backend of the ROS system APIs.

The `ROS_MASTER_URI` environment variable contains the IP and port of ROS Master. Using this variable, ROS nodes can locate ROS Master. If this variable is wrong, the communication between nodes will not take place. When we use ROS in a single system, we can use the IP of localhost or the name `localhost` itself. But in a distributed network, in which computation is on different physical computers, we should define `ROS_MASTER_URI` properly, only then, the remote node could find each other and communicate with each other. We need only one Master, in a distributed system, and it should run on a computer in which all other computers can ping it properly to ensure that remote ROS nodes can access the Master.
The following diagram shows an illustration of how ROS Master interacts with a publishing and subscribing node, the publisher node publishing a string type topic with a "Hello World" message and the subscriber node subscribes to this topic.

When the publisher node starts publishing the "Hello World" message in a particular topic, ROS Master gets the details of the topic and details of the node. It will search whether any node is subscribing the same topic. If there are no nodes subscribing the same topic at that time, both nodes remain unconnected. If the publisher and subscriber nodes run at the same time, ROS Master exchanges the details of the publisher to the subscriber and they will connect and can exchange data through ROS messages.

**Using the ROS parameter**

While programming a robot, we might have to define robot parameters such as robot controller gain such as P, I, and D. When the number of parameters increases, we might need to store it as files. In some situation, these parameters have to share between two or more programs too. In this case, ROS provides a parameter server, which is a shared server in which all ROS nodes can access parameters from this server. A node can read, write, modify and delete parameter values from the parameter server.
We can store these parameters in a file and load them into the server. The server can store a wide variety of data types and can even store dictionaries. The programmer can also set the scope of the parameter, that is, whether it can be accessed by only this node or all the nodes.

The parameter server supports the following XMLRPC datatypes, which include:

- 32-bit integers
- Booleans
- strings
- doubles
- iso8601 dates
- lists
- base64-encoded binary data

We can also store dictionaries on the parameter server. If the number of parameters is high, we can use a YAML file to save it. Here is an example of the YAML file parameter definitions:

```
/camera/name : 'nikon'   #string type
/camera/fps : 30         #integer
/camera/exposure : 1.2   #float
/camera/active : true    #boolean
```

The `rosparam` tool used to get and set the ROS parameter from the command line. The following are the commands to work with ROS parameters:

- `$ rosparam set [parameter_name] [value]`: This command will set a value in the given parameter
- `$ rosparam get [parameter_name]`: This command will retrieve a value from the given parameter
- `$ rosparam load [YAML file]`: The ROS parameters can be saved into a YAML file and it can load to the parameter server using this command
- `$ rosparam dump [YAML file]`: This command will dump the existing ROS parameters to a YAML file
- `$ rosparam delete [parameter_name]`: This command will delete the given parameter
- `$ rosparam list`: This command will list existing parameter names

The parameters can be changed dynamically during the execution of the node that uses these parameters, using the dynamic_reconfigure package (http://wiki.ros.org/dynamic_reconfigure).
Understanding ROS community level
These are ROS resources that enable a new community for ROS to exchange software and knowledge. The various resources in these communities are as follows:

- **Distributions**: Similar to the Linux distribution, ROS distributions are a collection of versioned meta packages that we can install. The ROS distribution enables easier installation and collection of the ROS software. The ROS distributions maintain consistent versions across a set of software.

- **Repositories**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.

- **The ROS Wiki**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

- **Bug ticket system**: If we find a bug in the existing software or need to add a new feature, we can use this resource.

- **Mailing lists**: The ROS-users mailing list is the primary communication channel about new updates to ROS, as well as a forum to ask questions about the ROS software.

- **ROS Answers**: This website resource helps to ask questions related to ROS. If we post our doubts on this site, other ROS users can see this and give solutions.

- **Blog**: The ROS blog updates with news, photos, and videos related to the ROS community (http://www.ros.org/news).

What are the prerequisites to start with ROS?
Before getting started with ROS and trying the code in this book, the following prerequisites should be met:

- **Ubuntu 14.04.2 LTS / Ubuntu 15.04**: We have to use Ubuntu as the operating system for installing ROS. We prefer to stick on to the L.T.S version of Ubuntu, that is, Ubuntu 14.04/14.04.3, or if you want to explore new ROS distribution you can use Ubuntu 15.04.

- **ROS Jade/Indigo desktop full installation**: Install the full desktop installation of ROS. The version we prefer is ROS Indigo, the latest version, Jade, is also supported. The following link gives you the installation instruction of the latest ROS distribution: http://wiki.ros.org/indigo/Installation/Ubuntu.
Running ROS Master and ROS parameter server

Before running any ROS nodes, we should start ROS Master and the ROS parameter server. We can start ROS Master and the ROS parameter server using a single command called `roscore`, which will start the following programs:

- ROS Master
- ROS parameter server
- rosout logging nodes

The rosout node will collect log messages from other ROS nodes and store them in a log file, and will also rebroadcast the collected log message to another topic. The topic /rosout is published by ROS nodes working using ROS client libraries such as roscpp and rospy and this topic is subscribed by the rosout node which rebroadcasts the message in another topic called /rosout_agg. This topic has an aggregate stream of log messages. The command roscore is a prerequisite before running any ROS node. The following screenshot shows the messages printing when we run the roscore command in a terminal.

The following is a command to run roscore on a Linux terminal:

```
$ roscore
```

![Figure 9: Terminal messages while running the roscore command](image)

[23]
The following are explanations of each section when executing `roscore` on the terminal:

- In the first section, we can see a log file is creating inside the `~/.ros/log` folder for collecting logs from ROS nodes. This file can be used for debugging purposes.
- In the second section, the command starts a ROS launch file called `roscore.xml`. When a launch file starts, it automatically starts the `rosmaster` and ROS parameter server. The `roslaunch` command is a Python script, which can start `rosmaster` and the ROS parameter server whenever it tries to execute a launch file. This section shows the address of the ROS parameter server within the port.
- In the third section, we can see the parameters such as `rosdistro` and `rosversion` displayed on the terminal. These parameters are displayed when it executes `roscore.xml`. We can see more on `roscore.xml` and its details in the next section.
- In the fourth section, we can see the `rosmaster` node is started using `ROS_MASTER_URI`, which we defined earlier as an environment variable.
- In the fifth section, we can see the `rosout` node is started, which will start subscribing the `/rosout` topic and rebroadcasting into `/rosout_agg`.

The following is the content of `roscore.xml`:

```xml
<launch>
  <group ns="/">
    <param name="rosversion" command="rosversion roslaunch" />
    <param name="rosdistro" command="rosversion -d" />
    <node pkg="rosout" type="rosout" name="rosout" respawn="true"/>
  </group>
</launch>
```

When the `roscore` command is executed, initially, the command checks the command line argument for a new port number for the `rosmaster`. If it gets the port number, it will start listening to the new port number, otherwise it will use the default port. This port number and the `roscore.xml` launch file will pass to the `roslaunch` system. The `roslaunch` system is implemented in a Python module, it will parse the port number and launch the `roscore.xml` file.

In the `roscore.xml` file, we can see the ROS parameters and nodes are encapsulated in a group XML tag with a "/" namespace. The group XML tag indicates that all the nodes inside this tag have the same settings.
The two parameters called rosversion and rosdistro store the output of the rosversion roslaunch and rosversion -d commands using the command tag, which is a part of the ROS param tag. The command tag will execute the command mentioned on it and store the output of the command in these two parameters.

The rosmaster and parameter server are executed inside roslaunch modules by using the ROS_MASTER_URI address. This is happening inside the roslaunch Python module. The ROS_MASTER_URI is a combination of the IP address and port in which rosmaster is going to listen. The port number can be changed according to the given port number in the roscore command.

**Checking the roscore command output**

Let's check the ROS topics and ROS parameters created after running roscore. The following command will list the active topics on the terminal:

```
$ rostopic list
```

The list of topics are as follows, as per our discussion on the rosoout node subscribe /rosout topic, which have all log messages from the ROS nodes and /rosout_agg rebroadcast the log messages:

/rosout
/rosout_agg

The following command lists out the parameters available when running roscore. The following is the command to list the active ROS parameter:

```
$ rosparam list
```

The parameters are mentioned here; they have the ROS distribution name, version, address of roslauch server and run_id, where run_id is a unique ID associated with a particular run of roscore:

/rosdistro
/roslaunch/uris/host_robot_virtualbox__51189
/rosversion
/run_id

The list of the ROS service generated during the running roscore can be checked using the following command:

```
$ rosservice list
```
The list of services running is as follows:

/rosout/get_loggers
/rosout/set_logger_level

These ROS services are generated for each ROS node for setting the logging levels.

After understanding the basics of ROS Master, Parameter server, and roscore we can go to the procedure to build a ROS package. Along with working with the ROS package, we can refresh the concepts of ROS nodes, topics, messages, services, and actionlib.

**Creating a ROS package**

The ROS packages are the basic unit of the ROS system. We can create the ROS package, build it and release it to the public. The current distribution of ROS we are using is Jade/Indigo. We are using the catkin build system to build ROS packages. A build system is responsible for generating 'targets' (executable/libraries) from a raw source code that can be used by an end user. In older distributions, such as Electric and Fuerte, rosbuilder was the build system. Because of the various flaws of rosbuilder, catkin came into existence, which is basically based on CMake (Cross Platform Make). This has lots of advantages such as porting the package into other operating system, such as Windows. If an OS supports CMake and Python, catkin based packages can be easily ported into it.

The first requirement in creating ROS packages is to create a ROS catkin workspace. Here is the procedure to build a catkin workspace.

Build a workspace folder in the home directory and create a src folder inside the workspace folder:

```
$ mkdir ~/catkin_ws/src
```

Switch to the source folder. The packages are created inside this package:

```
$ cd ~/catkin_ws/src
```

Initialize a new catkin workspace:

```
$ catkin_init_workspace
```

We can build the workspace even if there are no packages. We can use the following command to switch to the workspace folder:

```
$ cd ~/catkin_ws
```
The `catkin_make` command will build the following workspace:

```
$ catkin_make
```

After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called overlaying a workspace. We should add the package environment using the following command:

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

This command will source a bash script called `setup.bash` inside the `devel` workspace folder. To set the environment in all bash sessions, we need to add a `source` command in the `.bashrc` file, which will source this script whenever a bash session starts.

This is the link of the procedure [http://wiki.ros.org/catkin/Tutorials/create_a_workspace](http://wiki.ros.org/catkin/Tutorials/create_a_workspace).

1. After setting the `catkin` workspace, we can create our own package that has sample nodes to demonstrate the working of ROS topics, messages, services, and actionlib.

2. The `catkin_create_pkg` command is used to create a ROS package. This command is used to create our package in which we are going to create demos of various ROS concepts.

3. Switch to the `catkin` workspace `src` folder and create the package using the following command:

```
Syntax of catkin_create_pkg : catkin_create_pkg [package_name] [dependency1] [dependency2]
```

4. Here is the command to create the sample ROS package:

```
$ catkin_create_pkg mastering_ros_demo_pkg roscpp std_msgs actionlib actionlib_msgs
```

The dependencies in the packages are as follows:

- **roscpp**: This is the C++ implementation of ROS. It is a ROS client library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. We are including this dependency because we are going to write a ROS C++ node. Any ROS package which uses the C++ node must add this dependency.

- **std_msgs**: This package contains basic ROS primitive data types such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.
actionlib: The actionlib meta-package provides interfaces to create preemptable tasks in ROS nodes. We are creating actionlib based nodes in this package. So we should include this package to build the ROS nodes.

actionlib_msgs: This package contains standard message definitions needed to interact with the action server and action client.

We will get the following message if the package is successfully created:

Figure 10: Terminal messages while creating a ROS package

5. After creating this package, build the package without adding any nodes using the catkin_make command. This command must be executed from the catkin workspace path. The following command shows you how to build our empty ROS package:

```bash
~/catkin_ws$ catkin_make
```

6. After a successful build, we can start adding nodes to the src folder of this package.

The build folder in the CMake build files mainly contains executables of the nodes that are placed inside the catkin workspace src folder. The devel folder contains bash script, header files, and executables in different folders generated during the build process. We can see how to make ROS nodes and build using catkin_make.

Working with ROS topics

Topics are the basic way of communicating between two nodes. In this section, we can see how the topics works. We are going to create two ROS nodes for publishing a topic and subscribing the same. Navigate to the chapter_1_codes/mastering_ros_demo_package/src folder for the codes. demo_topic_publisher.cpp and demo_topic_subscriber.cpp are the two sets of code that we are going to discuss.

Creating ROS nodes

The first node we are going to discuss is demo_topic_publisher.cpp. This node will publish an integer value on a topic called /numbers. Copy the current code into a new package or use this existing file from the code repository.
Here is the complete code:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

int main(int argc, char **argv)
{
  ros::init(argc, argv,"demo_topic_publisher");
  ros::NodeHandle node_obj;
  ros::Publisher number_publisher = node_obj.advertise< std_msgs::Int32 >("/numbers",10);
  ros::Rate loop_rate(10);
  int number_count = 0;
  while (ros::ok())
  {
    std_msgs::Int32 msg;
    msg.data = number_count;
    ROS_INFO("%d",msg.data);
    number_publisher.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++number_count;
  }
  return 0;
}
```

Here is the detailed explanation of the preceding code:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

The ros/ros.h is the main header of ROS. If we want to use the roscpp client APIs in our code, we should include this header. The std_msgs/Int32.h is the standard message definition of integer datatype.

Here, we are sending an integer value through a topic. So we should need a message type for handling the integer data. std_msgs contains standard message definition of primitive datatypes. std_msgs/Int32.h contains integer message definition:

```
  ros::init(argc, argv,"demo_topic_publisher");
```

This code will initialize a ROS node with a name. It should be noted that the ROS node should be unique. This line is mandatory for all ROS C++ nodes:

```
  ros::NodeHandle node_obj;
```
Introduction to ROS and Its Package Management

This will create a `Nodehandle` object, which is used to communicate with the ROS system:

```cpp
ros::Publisher number_publisher =
    node_obj.advertise<std_msgs::Int32>("/numbers",10);
```

This will create a topic publisher and name the topic `/numbers` with a message type `std_msgs::Int32`. The second argument is the buffer size. It indicates that how many messages need to be put in a buffer before sending. It should be set to high if the data sending rate is high:

```cpp
ros::Rate loop_rate(10);
```

This is used to set the frequency of sending data:

```cpp
while (ros::ok())
{
```

This is an infinite while loop, and it quits when we press `Ctrl+C`. The `ros::ok()` function returns zero when there is an interrupt; this can terminate this while loop:

```cpp
    std_msgs::Int32 msg;
    msg.data = number_count;
```

The first line creates an integer ROS message and the second line assigns an integer value to the message. Here, `data` is the field name of the `msg` object:

```cpp
    ROS_INFO("%d",msg.data);
```

This will print the message data. This line is used to log the ROS information:

```cpp
    number_publisher.publish(msg);
```

This will publish the message to the topics `/numbers`:

```cpp
    ros::spinOnce();
```

This command will read and update all ROS topics. The node will not publish without a `spin()` or `spinOnce()` function:

```cpp
    loop_rate.sleep();
```

This line will provide the necessary delay to achieve a frequency of 10Hz.

After discussing the publisher node, we can discuss the subscriber node, which is `demo_topic_subscriber.cpp`. Copy the code to a new file or use the existing file.
Here is the definition of the subscriber node:

```cpp
#include "ros/rosh"
#include "std_msgs/Int32.h"
#include <iostream>

void number_callback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("Received \[%d\]", msg->data);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_topic_subscriber");
    ros::NodeHandle node_obj;
    ros::Subscriber number_subscriber = node_obj.subscribe("/numbers", 10, number_callback);
    ros::spin();
    return 0;
}
```

Here is the code explanation:

```cpp
#include "ros/rosh"
#include "std_msgs/Int32.h"
#include <iostream>

This is the header needed for the subscribers:

```cpp
void number_callback(const std_msgs::Int32::ConstPtr& msg)
{
    ROS_INFO("Received \[%d\]", msg->data);
}
```

This is a callback function that will execute whenever a data comes to the /numbers topic. Whenever a data reaches this topic, the function will call and extract the value and print it on the console:

```cpp
ros::Subscriber number_subscriber =
node_obj.subscribe("/numbers", 10, number_callback);
```

This is the subscriber and here, we are giving the topic name needed to subscribe, buffer size, and the callback function. We are subscribing /numbers topic and we have already seen the callback function in the preceding section:

```cpp
ros::spin();
```
This is an infinite loop in which the node will wait in this step. This code will fasten the callbacks whenever a data reaches the topic. The node will quit only when we press the Ctrl+C key.

Building the nodes

We have to edit the `CMakeLists.txt` file in the package to compile and build the source code. Navigate to `chapter_1_codes/mastering_ros_demo_package/CMakeLists.txt` to view the existing `CMakeLists.txt` file. The following code snippet in this file is responsible for building these two nodes:

```cpp
include_directories(
    include
    ${catkin_INCLUDE_DIRS}
    ${Boost_INCLUDE_DIRS}
)

#This will create executables of the nodes
add_executable(demo_topic_publisher src/demo_topic_publisher.cpp)
add_executable(demo_topic_subscriber src/demo_topic_subscriber.cpp)

#This will generate message header file before building the target
add_dependencies(demo_topic_publisher mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_topic_subscriber mastering_ros_demo_pkg_generate_messages_cpp)

#This will link executables to the appropriate libraries
target_link_libraries(demo_topic_publisher ${catkin_LIBRARIES})
target_link_libraries(demo_topic_subscriber ${catkin_LIBRARIES})
```

We can add the preceding snippet to create a new a `CMakeLists.txt` file for compiling the two codes.

The `catkin_make` command is used to build the package.

We can first switch to workspace:

```
$ cd ~/catkin_ws
```

Build `mastering_ros_demo_package` as follows:

```
$ catkin_make mastering_ros_demo_package
```

We can either use the preceding command to build a specific package or just `catkin_make` to build the entire workspace.
This will create executables in `~/catkin_ws/devel/lib/<package name>`.

If the building is done, we can execute the nodes.

First start **roscore**:

```
$ roscore
```

Now run both commands in two shells.

In the running publisher:

```
$ rosrundemonstratingrosdemos pacakge demotopic_publisher
```

In the running subscriber:

```
$ rosrundemonstratingrosdemos pacakge demotopic_subscriber
```

We can see the output as shown here:

![Running topic publisher and subscriber](image)

The following diagram shows how the nodes communicate with each other. We can see the `demotopic_publisher` node publish the `/numbers` topic and subscribe by then `demotopic_subscriber` node.

![Graph of the communication between publisher and subscriber nodes](image)
We can use the `rosnode` and `rostopic` tools to debug and understand the working of two nodes:

- `$ rosnode list`: This will list the active nodes
- `$ rosnode info demo_topic_publisher`: This will get the info of the publisher node
- `$ rostopic echo /numbers`: This will display the value sending through the `/numbers` topic
- `$ rostopic type /numbers`: This will print the message type of the `/numbers` topic

**Adding custom msg and srv files**

In this section, we can see how to create custom messages and services definitions in the current package. The message definitions are stored in a `.msg` file and service definition are stored in a `.srv` file. These definitions inform ROS about the type of data and name of data to be transmitted from a ROS node. When a custom message is added, ROS will convert the definitions into equivalent C++ codes, which we can include in our nodes.

We can start with message definitions.

Message definitions have to be written in the `.msg` file and have to be kept in the `msg` folder, which is inside the package.

We are going to create a message file called `demo_msg.msg` with the following definition:

```plaintext
string greeting
int32 number
```

Until now, we have worked only with standard message definitions. Now, we have created our own definitions and can see how to use them in our code.

The first step is to edit the `package.xml` file of the current package and uncomment the lines `<build_depend>message_generation</build_depend>` and `<run_depend>message_runtime</run_depend>`.

Edit the current `CMakeLists.txt` and add the `message_generation` line as follows:

```plaintext
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib
```

---

[34]
Uncomment the following line and add the custom message file:

```cpp
add_message_files(
    FILES
    demo_msg.msg
)
## Generate added messages and services with any dependencies listed here
generate_messages(
    DEPENDENCIES
    std_msgs
    actionlib_msgs
)
```

After these steps, we can compile and build the package:

```
$ cd ~/catkin_ws/
$ catkin_make
```

To check whether the message is built properly, we can use the `rosmsg` command:

```
$ rosmsg show mastering_ros_demo_pkg/demo_msg
```

If the content shown by the command and the definition are the same, the procedure is correct.

If we want to test the custom message, we can build a publisher and subscriber using the custom message type named `demo_msg_publisher.cpp` and `demo_msg_subscriber.cpp`. Navigate to the `chapter_1_codes/mastering_ros_demo_pkg/src` folder for these codes.

We can test the message by adding the following lines of code in `CMakeLists.txt`:

```cpp
add_executable(demo_msg_publisher src/demo_msg_publisher.cpp)
add_executable(demo_msg_subscriber src/demo_msg_subscriber.cpp)
add_dependencies(demo_msg_publisher mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_msg_subscriber mastering_ros_demo_pkg_generate_messages_cpp)
target_link_libraries(demo_msg_publisher ${catkin_LIBRARIES})
target_link_libraries(demo_msg_subscriber ${catkin_LIBRARIES})
```
Build the package using `catkin_make` and test the node using the following commands.

- Run roscore:
  
  ```
  $ roscore
  ```

- Start the custom message publisher node:
  
  ```
  $ rosrund mastering_ros_demo_pkg demo_msg_publisher
  ```

- Start the custom message subscriber node:
  
  ```
  $ rosrund mastering_ros_demo_pkg demo_msg_subscriber
  ```

The publisher node publishes a string along with an integer, and the subscriber node subscribes the topic and prints the values. The output and graph are shown as follows:

![Running publisher and subscriber using custom message definitions.](image)

The topic in which the nodes are communicating is called `/demo_msg_topic`. Here is the graph view of two nodes:

![Graph of the communication between message publisher and subscriber](image)

Next, we can add `srv` files to the package. Create a new folder called `srv` in the current package folder and add a `srv` file called `demo_srv.srv`. The definition of this file is as follows:

```
string in
---
string out
```
Here, both the Request and Response are strings.

In the next step, we need to uncomment the following lines in package.xml as we did for the ROS messages:

```xml
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Take CMakeLists.txt and add message_runtime in catkin_package():

```xml
catkin_package(  
CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs  
message_runtime  
)
```

We need to follow the same procedure in generating services as we did for the ROS message. Apart from that, we need additional sections to be uncommented as shown here:

```xml
## Generate services in the 'srv' folder  
add_service_files(  
FILES  
demo_srv.srv  
)
```

After making these changes, we can build the package using catkin_make and using the following command we can verify the procedure:

```
$ rossrv show mastering_ros_demo_pkg/demo_srv
```

If we see the same content as we defined in the file, we can confirm it's working.

## Working with ROS services

In this section, we are going to create ROS nodes, which can use the services definition that we defined already. The service nodes we are going to create can send a string message as a request to the server and the server node will send another message as a response.

Navigate to chapter_1_codes/mastering_ros_demo_pkg/src and find nodes with the names demo_service_server.cpp and demo_service_client.cpp.

The demo_service_server.cpp is the server and its definition is as follows:

```cpp
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
```
Introduction to ROS and Its Package Management

```cpp
#include <sstream>
using namespace std;

bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request &req,
                          mastering_ros_demo_pkg::demo_srv::Response &res)
{
    std::stringstream ss;
    ss << "Received Here";
    res.out = ss.str();
    ROS_INFO("From Client [%s], Server says [%s]", req.in.c_str(), res.out.c_str());
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_server");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("demo_service",
                                                  demo_service_callback);
    ROS_INFO("Ready to receive from client.");
    ros::spin();
    return 0;
}
```

Let's see the explanation of the code:

```cpp
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
```

Here, we included `ros/ros.h`, which is a mandatory header for a ROS CPP node. The `mastering_ros_demo_pkg/demo_srv.h` header is a generated header, which contains our service definition and can use this in our code. The `sstream.h` is for getting string streaming classes:

```cpp
bool demo_service_callback(mastering_ros_demo_pkg::demo_srv::Request &req,
                          mastering_ros_demo_pkg::demo_srv::Response &res)
{
```
This is the server callback function executed when a request is received on the server. The server can receive the request from clients having a message type of mastering_ros_demo_pkg::demo_srv::Request and sends the response in the mastering_ros_demo_pkg::demo_srv::Response type:

```cpp
std::stringstream ss;
ss << "Received Here";
res.out = ss.str();
```

In this code, the string data "Received Here" is passing to the service Response instance. Here, out is the field name of the response that we have given in the demo_srv.srv. This response will go to the service client node:

```cpp
ros::ServiceServer service = n.advertiseService("demo_service", demo_service_callback);
```

This creates a service having a name as demo_service and a callback function is executed when a request comes to this service. The callback function is demo_service_callback, which we saw in the preceding section.

Next, we can see how the demo_service_client.cpp is working.

Here is the definition of this code:

```cpp
#include "ros/ros.h"
#include <iostream>
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#include <sstream>
using namespace std;

int main(int argc, char **argv)
{
    ros::init(argc, argv, "demo_service_client");
    ros::NodeHandle n;
    ros::Rate loop_rate(10);
    ros::ServiceClient client = n.serviceClient<mastering_ros_demo_pkg::demo_srv>("demo_service");
    while (ros::ok())
    {
        mastering_ros_demo_pkg::demo_srv srv;
        std::stringstream ss;
        ss << "Sending from Here";
        srv.request.in = ss.str();
        if (client.call(srv))
        {
```
Let's explain the code:

```cpp
ROS_INFO("From Client [%s], Server says [%s]", srv.request.in.c_str(), srv.response.out.c_str());
```

This line creates a service client that has message type `mastering_ros_demo_pkg::demo_srv` and communicates to a ROS service named `demo_service`:

```cpp
mastering_ros_demo_pkg::demo_srv srv;
```

This line will create a new service object instance:

```cpp
std::stringstream ss;
ss << "Sending from Here";
srv.request.in = ss.str();
```

Fill the request instance with a string called "Sending from Here":

```cpp
if (client.call(srv))
```

This will send the service call to the server. If it is sent successfully, it will print the response and request, if it failed, it do nothing:

```cpp
ROS_INFO("From Client [%s], Server says [%s]", srv.request.in.c_str(), srv.response.out.c_str());
```

If the response is received, then it will print the request and the response.
After discussing the two nodes, we can discuss how to build these two nodes. The following code is added to CMakeLists.txt to compile and build the two nodes:

```cmake
add_executable(demo_service_server src/demo_service_server.cpp)
add_executable(demo_service_client src/demo_service_client.cpp)

add_dependencies(demo_service_server mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_service_client mastering_ros_demo_pkg_generate_messages_cpp)

target_link_libraries(demo_service_server ${catkin_LIBRARIES})
target_link_libraries(demo_service_client ${catkin_LIBRARIES})
```

We can execute the following commands to build the code:

```
$ cd ~/catkin_ws
$ catkin_make
```

To start nodes, first execute roscore and use the following commands:

```
$ rosrun mastering_ros_demo_pkg demo_service_server
$ rosrun mastering_ros_demo_pkg demo_service_client
```

![Figure 15: Running ROS service client and server nodes.](image)

---

[41]
We can work with `rosservice` using the `rosservice` command:

- `$ rosservice list`: This will list the current ROS services
- `$ rosservice type /demo_service`: This will print the message type of `/demo_service`
- `$ rosservice info /demo_service`: This will print the information of `/demo_service`

**Working with ROS actionlib**

In ROS services, the user implements a request/reply interaction between two nodes, but consider if the reply takes too much time or the server is not finished with the given work, we have to wait until it completes.

There is another method in ROS in which we can preempt the running request and start sending another one if the request is not finished on time as we expected. Actionlib packages provide a standard way to implement these kinds of preemptive tasks. Actionlib is highly used in robot arm navigation and mobile robot navigation. We can see how to implement an action server and action client implementation.

Like ROS services, in actionlib, we have to specify the action specification. The action specification is stored inside the action file having an extension of `.action`. This file must be kept inside the `action` folder, which is inside the ROS package. The action file has the following parts:

- **Goal**: The action client can send a goal that has to be executed by the action server. This is similar to the request in the ROS service. For example, if a robot arm joint wants to move from 45 degrees to 90 degrees, the goal here is 90 degrees.
- **Feedback**: When an action client sends a goal to the action server, it will start executing a call-back function. Feedback is simply giving the progress of the current operation inside the callback function. Using the feedback definition, we can get the current progress. In the preceding case, the robot arm joint has to move to 90 degrees; in this case, the feedback can be the intermediate value between 45 and 90 degrees in which the arm is moving.
- **Result**: After completing the goal, the action server will send a final result of completion, it can be the computational result or an acknowledgement. In the preceding example, if the joint reaches 90 degrees it achieves the goal and the result can be anything indicating it finished the goal.
We can discuss a demo action server and action client here. The demo action client will send a number as the goal. When an action server receives the goal, it will count from 0 to the goal number with a step size of 1 and with a one second delay. If it completes before the given time, it will send the result, otherwise, the task will be preempted by the client. The feedback here is the progress of counting. The action file of this task is as follows. The action file is named `Demo_action.action`:

```
#goal definition
int32 count
---
#result definition
int32 final_count
---
#feedback
int32 current_number
```

Here, the count value is the goal in which the server has to count from zero to this number. `final_count` is the result, in which the final value after completion of a task and `current_number` is the feedback value. It will specify how much the progress is.

Navigate to `chapter_1_codes/mastering_ros_demo_pkg/src` and you can find the action server node as `demo_action_server.cpp` and action client node as `demo_action_client.cpp`.

**Creating the ROS action server**

In this section, we will discuss `demo_action_server.cpp`. The action server receives a goal value that is a number. When the server gets this goal value, it will start counting from zero to this number. If the counting is complete, it will successfully finish the action, if it is preempted before finishing, the action server will look for another goal value.

This code is a bit lengthy, so we can discuss the important code snippet of this code.

Let's start from the header files:

```cpp
#include <actionlib/server/simple_action_server.h>
#include "mastering_ros_demo_pkg/Demo_actionAction.h"
```

The first header is the standard action library to implement an action server node. The second header is generated from the stored action files. It should include for accessing our action definition:

```cpp
class Demo_actionAction
{
```
This class contains the action server definition:

```cpp
actionlib::SimpleActionServer<mastering_ros_demo_pkg::Demo_actionAction> as;
```

Create a simple action server instance with our custom action message type:

```cpp
mastering_ros_demo_pkg::Demo_actionFeedback feedback;
```

Create a feedback instance for sending feedback during the operation:

```cpp
mastering_ros_demo_pkg::DemoActionResult result;
```

Create a result instance for sending the final result:

```cpp
Demo_actionAction(std::string name) :
    as(nh_, name, boost::bind(&Demo_actionAction::executeCB, this, _1), false),
    action_name(name)
```

This is an action constructor, and an action server is created here by taking an argument such as `NodeHandle`, `action_name`, and `executeCB`, where `executeCB` is the action callback where all the processing is done:

```cpp
as.registerPreemptCallback(boost::bind(&Demo_actionAction::preemptCB, this));
```

This line registers a callback when the action is preempted. The `preemptCB` is the callback name executed when there is a preempt request from the action client:

```cpp
void executeCB(const mastering_ros_demo_pkg::Demo_actionGoalConstPtr &goal)
{
    if(!as.isActive() || as.isPreemptRequested()) return;
}
```

This is the callback definition which is executed when the action server receives a goal value. It will execute callback functions only after checking whether the action server is currently active or it is preempted already:

```cpp
for(progress = 0 ; progress < goal->count; progress++){
    //Check for ros
    if(!ros::ok()){
        return;
    }
```

This loop will execute until the goal value is reached. It will continuously send the current progress as feedback:

```cpp
    if(!as.isActive() || as.isPreemptRequested()){  
        return;
    }
```

This is the code snippet for creating a simple action server instance.
Inside this loop, it will check whether the action server is active or it is preempted. If it occurs, the function will return:

```c
if (goal->count == progress){
    result.final_count = progress;
    as.setSucceeded(result);
}
```

If the current value reaches the goal value, then it publishes the final result:

```c
Demo_actionAction demo_action_obj(ros::this_node::getName());
```

In `main()`, we create an instance of `Demo_actionAction`, which will start the action server.

**Creating the ROS action client**

In this section, we will discuss the working of an action client. `demo_action_client.cpp` is the action client node that will send the goal value consisting of a number which is the goal. The client is getting the goal value from the command line arguments. The first command line argument of the client is the goal value and the second is the time of completion for this task.

The goal value will be sent to the server and the client will wait until the given time, in seconds. After waiting, the client will check whether it completed or not; if it is not complete, the client will preempt the action.

The client code is a bit lengthy, so we will discuss the important sections of the code:

```c
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include "mastering_ros_demo_pkg/Demo_actionAction.h"
```

In action client, we need to include `actionlib/client/simple_action_client.h` to get the action client APIs which are used to implement action clients:

```c
actionlib::SimpleActionClient<mastering_ros_demo_pkg::Demo_actionAction> ac("demo_action", true);
```

This will create an action client instance:

```c
ac.waitForServer();
```

This line will wait for an infinite time if there is no action server running on the system. It will exit only when there is an action server running on the system:

```c
mastering_ros_demo_pkg::Demo_actionGoal goal;
goal.count = atoi(argv[1]);
ac.sendGoal(goal);
```
Create an instance of a goal and send the goal value from the first command line argument:

```cpp
bool finished_before_timeout = ac.waitForResult(ros::Duration(atoi(argv[2])));
```

This line will wait for the result from the server until the given seconds:

```cpp
ac.cancelGoal();
```

If it is not finished, it will preempt the action.

**Building the ROS action server and client**

After creating these two files in the `src` folder, we have to edit the `package.xml` and `CMakeLists.txt` to build the nodes.

The `package.xml` file should contain message generation and runtime packages as we did for ROS service and messages.

We have to include the **Boost** library in `CMakeLists.txt` to build these nodes. Also, we have to add the action files that we wrote for this example:

```cpp
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  actionlib
  actionlib_msgs
  message_generation
}
```

We should pass `actionlib`, `actionlib_msgs`, and `message_generation` in `find_package()`:

```cpp
## System dependencies are found with CMake's conventions
find_package(Boost REQUIRED COMPONENTS system)
```

We should add **Boost** as a system dependency:

```cpp
## Generate actions in the 'action' folder
add_action_files(
  FILES
  Demo_action.action
)
```
We need to add our action file in `add_action_files()`:

```python
## Generate added messages and services with any dependencies listed here
generate_messages(
    DEPENDENCIES
    std_msgs
    actionlib_msgs
)
```

We have to add `actionlib_msgs` in `generate_messages()`:

```python
catkin_package(
    CATKIN_DEPENDS roscpp rospy std_msgs actionlib actionlib_msgs
    message_runtime
)
```

We have to add `Boost` to include the directory:

```python
##Building action server and action client

add_executable(demo_action_server src/demo_action_server.cpp)
add_executable(demo_action_client src/demo_action_client.cpp)

add_dependencies(demo_action_server mastering_ros_demo_pkg_generate_messages_cpp)
add_dependencies(demo_action_client mastering_ros_demo_pkg_generate_messages_cpp)

target_link_libraries(demo_action_server ${catkin_LIBRARIES} )
target_link_libraries(demo_action_client ${catkin_LIBRARIES})
```

After `catkin_make`, we can run these nodes using the following commands:

- Run roscore:
  
  `$ roscore`
- Launch the action server node:
  
  `$rosrun mastering_ros_demo_pkg demo_action_server`
Launch the action client node:

```
$rosrun mastering_ros_demo_pkg demo_action_client 50 4
```

The output of these process is shown as follows:

![Figure 16: Running ROS actionlib server and client](image)

### Creating launch files

The launch files in ROS are a very useful feature for launching more than one node. In the preceding examples, we have seen a maximum of two ROS nodes, but imagine a scenario in which we have to launch 10 or 20 nodes for a robot. It will be difficult if we run each node in a terminal one by one. Instead of that, we can write all nodes inside a XML based file called launch files and using a command called `roslaunch`, we can parse this file and launch the nodes.
The `roslaunch` command will automatically start ROS Master and the parameter server. So in essence, there is no need to start the `roscore` command and individual node; if we launch the file, all operations will be done in a single command.

Let's start creating launch files. Switch to the package folder and create a new launch file called `demo_topic.launch` to launch two ROS nodes that are publishing and subscribing an integer value. We keep the launch files in a `launch` folder, which is inside the package:

```bash
$ roscd mastering_ros_demo_pkg
$ mkdir launch
$ cd launch
$ gedit demo_topic.launch
```

Paste the following content into the file:

```xml
<launch>
  <node name="publisher_node" pkg="mastering_ros_demo_pkg" type="demo_topic_publisher" output="screen"/>

  <node name="subscriber_node" pkg="mastering_ros_demo_pkg" type="demo_topic_subscriber" output="screen"/>
</launch>
```

Let's discuss what is in the code. The `<launch>` tags are the root element in a launch file. All definitions will be inside these tags.

The `<node>` tag specifies the desired node to launch:

```xml
  <node name="publisher_node" pkg="mastering_ros_demo_pkg" type="demo_topic_publisher" output="screen"/>
```

The name tag inside `<node>` indicates the name of the node, `pkg` is the name of the package, and `type` is the name of executable we are going to launch.

After creating the launch file `demo_topic.launch`, we can launch it using the following command:

```bash
$ roslaunch mastering_ros_demo_pkg demo_topic.launch
```
Here is the output we get if the launch is successful:

![Terminal messages](image)

> Figure 17: Terminal messages while launching the demo_topic.launch file

We can check the list of nodes using:

```bash
$ rosnode list
```

We can also view the log messages and debug the nodes using a GUI tool called `rqt_console`:

```bash
$ rqt_console
```

We can see the logs generated by two nodes in this tool as shown here:

![Logging using rqt_console](image)

> Figure 18: Logging using the rqt_console tool

### Applications of topics, services, and actionlib

Topics, services, and actionlib are used in different scenarios. We know topics are a unidirectional communication method, services are a bidirectional request/reply kind of communication, and actionlib is a modified form of ROS services in which we can cancel the executing process running on the server whenever required.
Here are some of the areas where we use these methods:

- **Topics**: Robot teleoperation, publishing odometry, sending robot transform (TF), and sending robot joint states
- **Services**: This saves camera calibration parameters to a file, saves a map of the robot after SLAM, and loads a parameter file
- **Actionlib**: This is used in motion planners and ROS navigation stacks

The complete source code of this project can be cloned from the following Git repository. The following command will clone the project repo:

```$ git clone https://github.com/qboticslabs/mastering_ros_demo_pkg.git```

**Maintaining the ROS package**

Most of the ROS packages are released as open source with the BSD license. There are active developers around the globe who are contributing to the ROS platform. Maintaining packages are an important constraint in all software especially open source application. Open source software is maintained and supported by a community of developers. Creating a version control system for our package is essential if we want to maintain and accept a contribution from other developers. The preceding package is already updated in GitHub and you can view the source code of the project at [https://github.com/qboticslabs/mastering_ros_demo_pkg](https://github.com/qboticslabs/mastering_ros_demo_pkg)

After uploading the code in GitHub, we can see what the procedures are to release our current package to ROS.

**Releasing your ROS package**

After creating a ROS package in GitHub, we can officially release our package. ROS provides detailed steps to release the ROS package using a tool called bloom ([http://ros-infrastructure.github.io/bloom/](http://ros-infrastructure.github.io/bloom/)). Bloom is a release automation tool, designed to make platform-specific releases from the source projects. Bloom is designed to work best with the catkin project.

The prerequisites before releasing the package are as follows:

- Install the bloom tool
- Create a Git repository for the current package
- Create an empty Git repository for the release
The following command will install bloom in Ubuntu:

```
$ sudo apt-get install python-bloom
```

Create a Git repository for the current package. The repository that has the package is called the upstream repository. Here, we already created a repository at https://github.com/qboticslabs/mastering_ros_demo_pkg.

Create an empty repository in Git for the release package. This repository is called the release repository. We have created a package called demo_pkg-release. This package is at https://github.com/qboticslabs/demo_pkg-release.

After meeting these prerequisites, we can start to create the release of the package. Navigate to the mastering_ros_demo_pkg local repository where we push our package code to Git. Open a terminal inside this local repository and execute the following command:

```
$ catkin_generate_changelog
```

The purpose of this command is, it will create a CHANGELOG.rst file inside the local repository. After executing this command it will show this option:

Continue without -all option [y/N]. Give y here

It will create a CHANGELOG.rst in the local repository.

After the creation of the log file, we can update the Git repository by committing the changes:

```
$ git add -A
$ git commit -m 'Updated CHANGELOG.rst'
$ git push -u origin master
```

**Preparing the ROS package for the release**

In this step, we are checking whether the package contains change logs, versions, and so on. The following command makes our package consistent and recommended for a release.

This command should execute from the local repository of the package:

```
$ catkin_prepare_release
```

The command will set a version tag if there is no current version and commit the changes in the upstream repository.
Releasing our package

The following command starts the release. The syntax of this command is as follows:

```
bloom-release --rosdistro <ros_distro> --track <ros_distro> repository_name
```

```
$ bloom-release --rosdistro indigo --track indigo mastering_ros_demo_pkg
```

When this command is executed, it will go to the rosdistro (https://github.com/ros/rosdistro) package repository to get the package details. The rosdistro package in ROS contains an index file, which contains a list of all the packages in ROS. Currently, there is no index for our package because this is our first release, but we can add our package details to this index file called `distributions.yaml`.

The following message will be displayed when there is no reference of the package in rosdistro:

```
<table>
<thead>
<tr>
<th>lenti@lenti-Aspire-4755 /media/lenti/Work/Mastering_Robotics_using_ROS/Chapter-1-Introduction to ROS</th>
<th></th>
</tr>
</thead>
</table>
| New ROS Distro index url: 'https://raw.githubusercontent.com/ros/rosdistro/18cdf2a3f16d7102aedab8628b8a2/index.yaml'
| Specified repository 'mastering_ros_demo_pkg' is not in the distribution file located at 'https://raw.githubusercontent.com/ros/rosdistro/18cdf2a3f16d7102aedab8628b8a2/index.yaml'
| Could not determine release repository url for repository 'mastering_ros_demo_pkg' of distro 'indigo'
| You can continue the release process by manually specifying the location of the RELEASE repository.
| To be clear this is the url of the RELEASE repository not the upstream repository.
| For release repositories on GitHub, you should provide the `https://` url which should end in `.git`
| Here is the url for a typical release repository on GitHub: https://github.com/ros-gbp/rviz-release.git
| >>> Looking for a release of this repository in a different distribution...
| No reasonable default release repository url could be determined from previous releases.
| Release repository url [press enter to abort]: https://github.com/qbolicslabs/deep_pkg-release.git
```
We should give the release repository in the terminal that is marked in red in the preceding screenshot. In this case, the URL was https://github.com/qboticslabs/demo_pkg-release.

In the upcoming steps, the wizard will ask for the repository name, upstream, URL, and so on. We can give these options and finally, a pull request to rosdistro will be submitted, which is shown in the following screenshot:
Chapter 1

The pull request for this package can be viewed at https://github.com/ros/rosdistro/pull/9662.

If it is accepted, it will merge to indigo/distribution.yaml, which contains the index of all packages in ROS.

The following screenshot displays the package as an index in indigo/distribution.yaml:

![distribution.yaml file of ROS Indigo](image)

After this step, we can confirm that the package is released and officially added to the ROS index.

**Creating a Wiki page for your ROS package**

ROS wiki allows users to create their own home pages to showcase their package, robot, or sensors. The official wiki page of ROS is wiki.ros.org. Now, we are going to create a wiki page for our package.

**Downloading the example code**

You can download the example code files from your account at [http://www.packtpub.com](http://www.packtpub.com) for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you. You can also download chapter codes from [https://github.com/qboticslabs/mastering_ros.git](https://github.com/qboticslabs/mastering_ros.git).
The first step is to register in wiki using your mail address. Go to wiki.ros.org, and click on the login button as shown in the screenshot:

![Figure 23: Locating the login option from ROS wiki](image)

After clicking on **Login**, you can register or directly login with your details if you are already registered. After login, press the user name link on the right side of the wiki page as shown in the screenshot:

![Figure 24: Locating the user account button from ROS wiki](image)
After clicking on this link, you will get a chance to create a home page for your package; you will get a text editor with GUI to enter data into. The following screenshot shows you the page we created for this demo package:

![Figure 25: Creating a new wiki page](image)

The wiki page of this package can be viewed at [http://wiki.ros.org/qboticslabs](http://wiki.ros.org/qboticslabs).

**Questions**

- Why should we learn ROS?
- How does ROS differ from other robotic software platforms?
- What is the internal working of `roscore`?
- How do ROS topic and service differ in their operations?
- How do ROS service and actionlib differ in their operations?
Summary

ROS is now a trending software framework among roboticists. Gaining knowledge in ROS is essential in the upcoming years if you are planning to build your career as a robotics engineer. In this chapter, we have gone through the basics of ROS mainly to refresh the concepts if you have already learned ROS. We discussed the necessity of learning ROS and how it excels among the current robotics software platforms. We went through the basic concepts such as ROS Master, Parameter server, and roscore and saw the explanation of the working of roscore. After discussing the internal working of roscore, we discussed each ROS concept, such as ROS topics, services, messages, and actionlib by illustrating examples. After demonstrating the working of each concept, we uploaded the package to GitHub and created a wiki page for the package. In the next chapter, we will discuss ROS robot modeling using URDF and xacro and will design some robot models.
Where to buy this book

You can buy Mastering ROS for Robotics Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.