1
Introduction to ROS

The first two chapters of this book introduce basic ROS concepts and its package management system in order to refresh your memory about concepts you should already know. In this first chapter, we will go through ROS concepts such as the ROS Master, the ROS nodes, the ROS parameter server, ROS messages and services discussing what we need to install ROS and how to get started with the ROS master.

In this chapter, we will cover the following topics:

- Why should we learn ROS?
- Why should we prefer or should not prefer ROS for robots?
- Getting started with the ROS filesystem level and its computation graph level.
- Understanding ROS framework elements.
- Getting started with the ROS master.

Why should we learn ROS?

Robot Operating System (**ROS**) is a flexible framework, providing various tools and libraries to write robotic software. It offers several powerful features to help developers in such tasks as message passing, distributing computing, code reusing, and implementation of state-of-the-art algorithms for robotic applications.

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 applications 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, a knowledge of 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, **Simultaneous Localization and Mapping** (**SLAM**) and **Adaptive Monte Carlo Localization** (**AMCL**) packages in ROS can be used for performing autonomous navigation in mobile robots, and the `MoveIt` package can be used 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 is like reinventing the wheel. 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 a simulation. The tools, such as rqt_gui, RViz, and Gazebo, are some of the strong open source tools for debugging, visualization, and simulation. A software framework that has these many tools is very rare.
- **Support for 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 communication 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 is that 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 operations even if any sensors or motors are dead.
- **Concurrent resource handling**: Handling a hardware resource via 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 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 some people do not prefer ROS for their robotic projects:

- **Difficulty in learning**: ROS can be difficult to learn. It has a steep learning curve and developers should become familiar with many new concepts to get benefits from the ROS framework.
- **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 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 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.
- **Potential limitations**: Current ROS versions have some limitations. For example, there is a lack of a native real-time application development support or the complexity to implement robust multi-robot distributed applications.
- **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 code follows a standard coding style and keeps 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 the 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 our 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: the filesystem level, computation graph level, and community level. We will briefly have a look at each level.

Understanding the ROS filesystem level

ROS is more than a development framework. We can refer to ROS as a meta-operating system, since it offers not only tools and libraries but even OS-like functions, such as hardware abstraction, package management, and a developer toolchain. Like a real operating system, ROS files are organized on the hard disk in a particular manner, as depicted in the following figure:

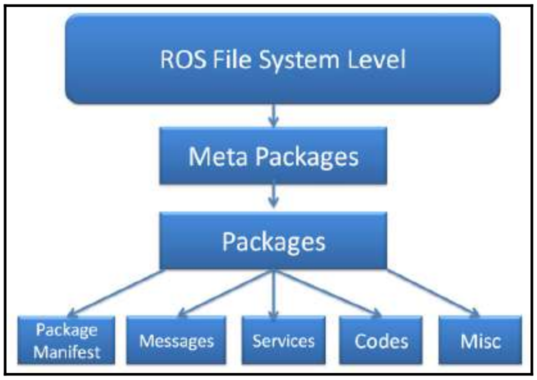


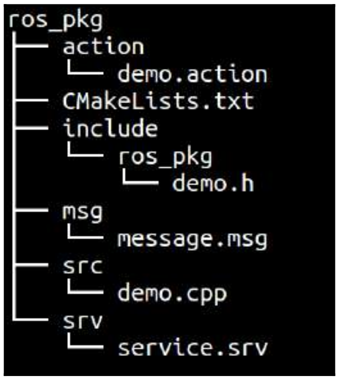
Figure 1: ROS filesystem level

Here are the explanations for each block in the filesystem:

- **Packages**: The ROS packages are the most basic unit of the ROS software. They contain one or more ROS programs (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.

- **Metapackages**: The term metapackage refers to one or more related packages which can be loosely grouped together. In principle, metapackages are virtual packages that don't contain any source code or typical files usually found in packages.
- **Metapackages manifest**: The metapackage manifest is similar to the package manifest, the difference being 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 the files and folders of a package that we are going to create in the upcoming sections:



```
ros_pkg
├── action
│   └── demo.action
├── CMakeLists.txt
├── include
│   └── ros_pkg
│       └── demo.h
├── msg
│   └── message.msg
├── src
│   └── demo.cpp
└── srv
    └── service.srv
```

Figure 2: List of files inside the exercise package

ROS packages

A typical structure of an ROS package is shown here:

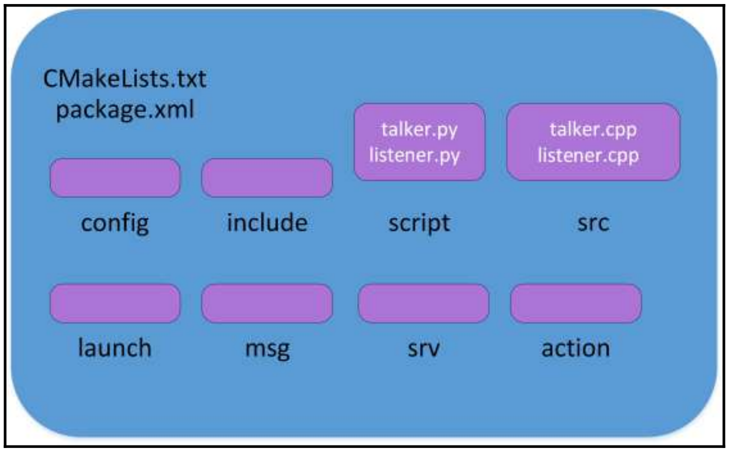


Figure 3: Structure of a typical C++ ROS package

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 it 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.
- `script`: This folder keeps executable Python scripts. In the block diagram, we can see two example scripts.
- `src`: This folder stores the C++ source codes.
- `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 services definitions.
- `action`: This folder contains the action files. We will see more about these kind of files in the next chapter.
- `package.xml`: This is the package manifest file of this package.
- `CMakeLists.txt`: This files contains the directives to compile the package.

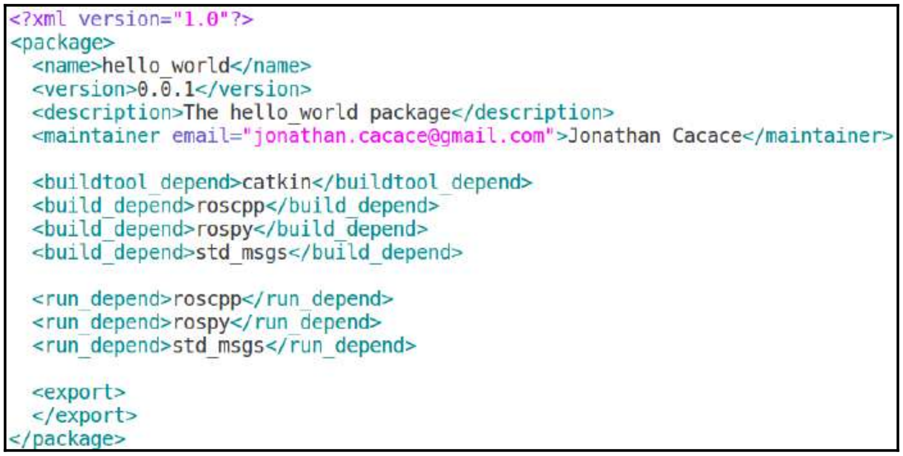
We need to know some commands to create, modify, and work with the ROS packages. Here are some of the commands used to work with ROS packages:

- `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 filesystem
- `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 current directory using a package name, stack name, or a special location. 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 using the *vim* editor.
- `rosrun`: This command is used to run an executable inside a package.

The definition of `package.xml` of a typical package is shown in the following screenshot:



```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.1</version>
  <description>The hello_world package</description>
  <maintainer email="jonathan.cacace@gmail.com">Jonathan Cacace</maintainer>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </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></build_depend>` tag includes the packages that are necessary to build the source code of the package. The packages inside the `<run_depend></run_depend>` tags are necessary during runtime of the package node.

ROS metapackages

Metapackages are specialized packages in ROS that only contain one file, that is, a `package.xml` file. They don't contain folders and files like a normal package.

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

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

Also, in metapackages, there are no `<buildtool_depend>` dependencies for `catkin`; there are only `<run_depend>` dependencies, which are the packages grouped in the metapackage.

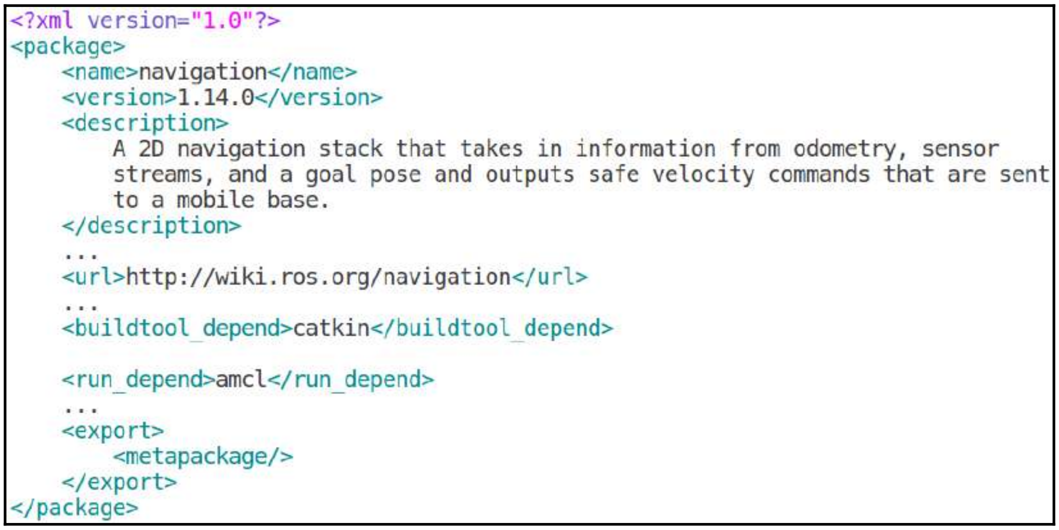
The ROS navigation stack is a good example of metapackages. If ROS and its navigation package are installed, we can try the following command, by switching to the navigation metapackage folder:

```
$ roscd navigation
```

Open `package.xml` using your favorite text editor (`gedit` in the following case):

```
$ gedit package.xml
```

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



```
<?xml version="1.0"?>
<package>
    <name>navigation</name>
    <version>1.14.0</version>
    <description>
        A 2D navigation stack that takes in information from odometry, sensor
        streams, and a goal pose and outputs safe velocity commands that are sent
        to a mobile base.
    </description>
    ...
    <url>http://wiki.ros.org/navigation</url>
    ...
    <buildtool_depend>catkin</buildtool_depend>

    <run_depend>amcl</run_depend>
    ...
    <export>
        <metapackage/>
    </export>
</package>
```

Figure 5: Structure of meta-package `package.xml`

ROS messages

The ROS nodes can write or read data that has a different 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 is stored in `.msg` files in the `msg` subdirectory of a ROS package. Even though the ROS framework provides a large set of robotic-specific messages already implemented, developers can define their own message type inside their nodes.

The message definition can consist of two types: `fields` and `constants`. The field is split into field types and field names. The field type 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:

```
int32 number
string name
float32 speed
```

Here, the first part is the field type and the 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 showing some of the built-in field types that we can use in our message:

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

Other kinds of messages are designed to cover a specific application necessity, such as exchanging common geometrical (`geometry_msgs`) or sensor (`sensor_msgs`) information. A special type of ROS message is called a message header. 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:

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

The `rosmsg` 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:

```
$ rosmsg show std_msgs/Header
```

This will give you an output like the preceding example message header. We will look at the `rosmsg` command and how to work with custom message definitions further 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 the request that is separated by `---` and in the next section is the message type of the response. In these examples, both `Request` and `Response` are strings.

In the upcoming sections, we will look at how to work with ROS services.

Understanding the ROS computation graph level

The computation in ROS is done using a network of a 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`, `rosparam`, `rosservice`, and `rosnode` to introspect the preceding concepts.

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

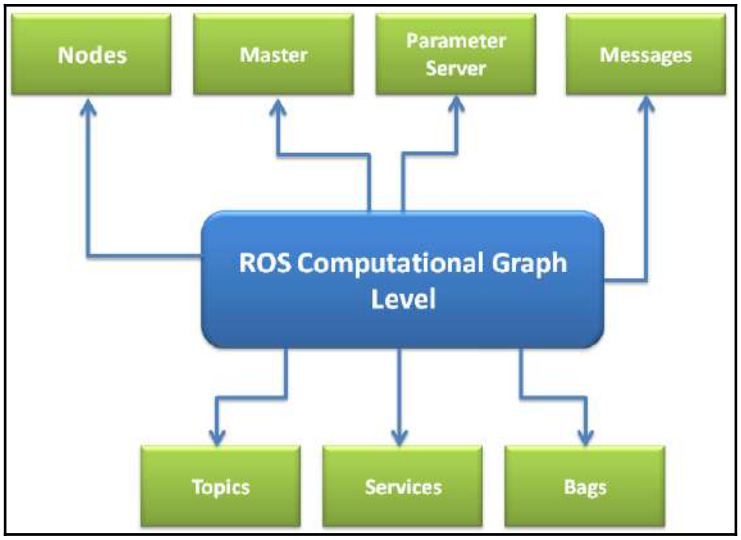


Figure 6: Structure of the ROS Graph layer

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. Using client library APIs, we can implement different ROS functionalities, such as the communication methods between nodes, which is particularly useful when different nodes of our robot must exchange information between them. Using the ROS communication methods, they 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 the functionality. Being a simple structure, ROS nodes are easy to debug.
- **Master**: The ROS Master provides the 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. The parameter server is a part of the 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, the publish/subscribe communication model may not be suitable. For example, in some cases, we need a kind of request/response interaction, in which one node can ask for the execution of a fast procedure to another node; for example, asking for some quick calculation. The ROS service interaction is like a remote procedure call.

- **Logging**: ROS provides a logging system for storing data, such as sensor data, which can be difficult to collect but is necessary for developing and testing robot algorithms: the bagfiles. Bagfiles 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 the 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`):

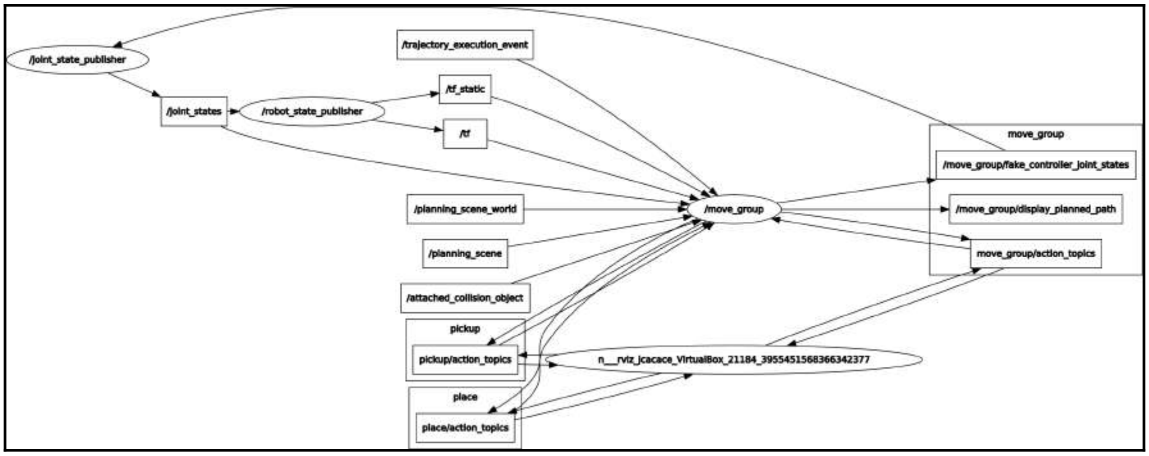


Figure 7: Graph of communication between nodes using topics

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 code 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 `rosnode` command can be used to get information about a ROS node. Here are the usages of `rosnode`:

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

We will look at 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 in knowing which node is publishing the topic or subscribing topics; they only look for the topic name and whether the message types of the publisher and subscriber are matching.

The communication using topics are unidirectional. If we want to implement a 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 teleoperations.

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 in a human readable format. Users can use the "-p" option to print data in a csv format.
- `$ 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 implement natively such kind of communication because it is unidirectional. The ROS services are mainly used in a distributed system.

The ROS services are 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. For example, consider a node able to provide the sum of two numbers received in input, implementing this functionality through a ROS service. The other nodes of our system might request the sum of two numbers via this service. Differently, topics are used to stream continuous data flow.

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 `rosmsg`, 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.

Further details about this command can be found at:
`http://wiki.ros.org/rosbag/Commandline`

There is a GUI tool to handle the record and playback of bag files called `rqt_bag`. To learn more about `rqt_bag`, go to: `http://wiki.ros.org/rqt_bag`.

The ROS Master

The ROS Master is much like a DNS server, associating unique names and IDs to ROS elements active in our system. When any node starts in the ROS system, it will start looking for the ROS Master and register the name of the node in it. So, the ROS Master has the details of all the 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 the ROS Master. The ROS Master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, the 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, the 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 requirement. If we stop any nodes, it will check with the ROS Master once again. This same method is used for the ROS services.

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

The `ROS_MASTER_URI` environment variable contains the IP and port of the ROS Master. Using this variable, ROS nodes can locate the 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 a 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 will the remote nodes be able 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 the ROS Master interacts with a publishing and subscribing node, with the publisher node publishing a string type topic with a `Hello World` message and the subscriber node subscribing to this topic:

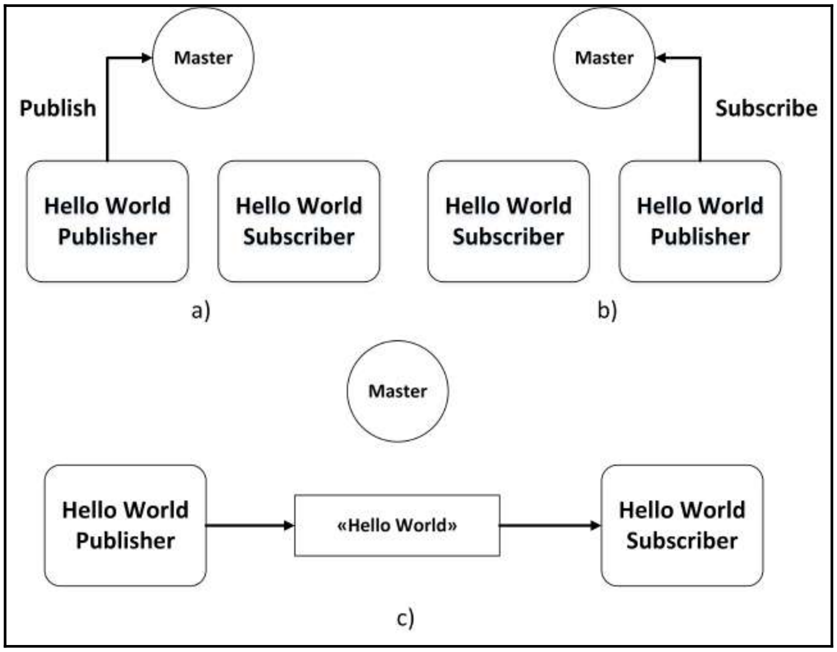


Figure 8: Communication between the ROS Master and Hello World publisher and subscriber

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

When programming a robot, we might have to define robot parameters, such as robot controller gains P, I, and D. When the number of parameters increases, we might need to store them as files. In some situations, 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:

- 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 them. 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 is 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 `dyamic_reconfigure` package (`http://wiki.ros.org/dynamic_reconfigure`).

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 metapackages 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 for starting with ROS?

Before getting started with ROS and trying the code in this book, the following prerequisites should be met:

- **Ubuntu 16.04 LTS / Ubuntu 15.10 / Debian 8**: ROS is officially supported by Ubuntu and Debian operating systems. We prefer to stick with the LTS version of Ubuntu, that is, Ubuntu 16.04.
- **ROS kinetic desktop full installation**: Install the full desktop installation of ROS. The version we prefer is ROS kinetic, the latest stable version. The following link gives you the installation instruction of the latest ROS distribution: `http://wiki.ros.org/kinetic/Installation/Ubuntu`. Choose the `ros-kinetic-desktop-full` package from the repository list.

Running the ROS Master and the ROS parameter server

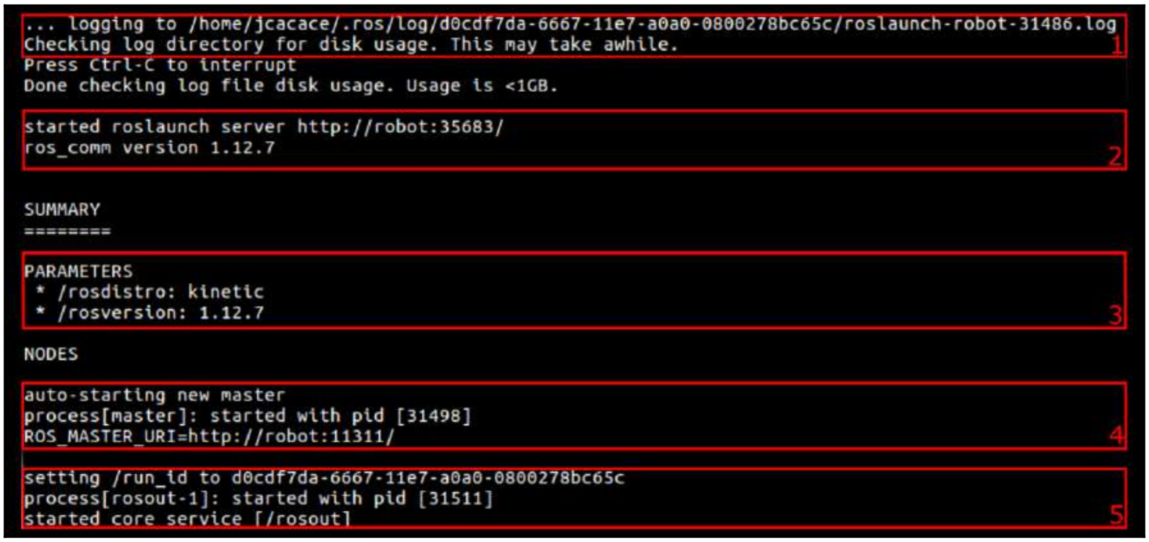
Before running any ROS nodes, we should start the ROS Master and the ROS parameter server. We can start the ROS Master and the ROS parameter server by using a single command called `roscore`, which will start the following programs:

- 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 re-broadcast the collected log message to another topic. The `/rosout` topic is published by ROS nodes by 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 `roscore` command 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
```



```
... logging to /home/jcacace/.ros/log/d0cdf7da-6667-11e7-a0a0-0800278bc65c/roslaunch-robot-31486.log
Checking log directory for disk usage. This may take awhile.                                          1
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://robot:35683/
ros_comm version 1.12.7                                                                               2


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7                                                                                3

NODES

auto-starting new master
process[master]: started with pid [31498]
ROS_MASTER_URI=http://robot:11311/                                                                    4

setting /run_id to d0cdf7da-6667-11e7-a0a0-0800278bc65c
process[rosout-1]: started with pid [31511]
started core service [/rosout]                                                                        5
```

Figure 9: Terminal messages while running the `roscore` command

The following are explanations of each section when executing `roscore` on the Terminal:

- In **section 1**, we can see a log file is created inside the `~/.ros/log` folder for collecting logs from ROS nodes. This file can be used for debugging purposes.
- In **section 2**, the command starts a ROS launch file called `roscore.xml`. When a launch file starts, it automatically starts the `rosmaster` and the 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 **section 3**, we can see the parameters such as `rosdistro` and `rosversion` displayed on the Terminal. These parameters are displayed when it executes `roscore.xml`. We look at `roscore.xml` and its details further in the next section.
- In **section 4**, we can see the `rosmaster` node is started using `ROS_MASTER_URI`, which we defined earlier as an environment variable.
- In **section 5**, 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`:

```
<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 `rosversionroslaunch` 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 is as follows, as per our discussion on the `rosout` node subscribe `/rosout` topic. This has all the log messages from the ROS nodes and `/rosout_agg` will rebroadcast the log messages:

```
/rosout
/rosout_agg
```

The following command lists 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 the `roslaunch` 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.

Questions

After going through the chapter, you should now be able to answer the following questions:

- Why should we learn ROS?
- How does ROS differ from other robotic software platforms?
- What are the basic elements of ROS framework?
- What is the internal working of `roscore`?

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 the ROS Master, the parameter server, and `roscore` , and looked at the explanation of the working of `roscore`. In the next chapter, we will introduce the ROS package management, discussing some practical examples of the ROS communication system.

2
Getting Started with ROS Programming

After discussing the basics of the ROS Master, the parameter server, and `roscore` , we can now start to create and build a ROS package. In this chapter, we will create different ROS nodes implementing the ROS communication system. Working with ROS packages, we will also refresh the concepts of ROS nodes, topics, messages, services, and actionlib.

We will cover the following list of topics:

- Creating, compiling and running ROS packages.
- Working with standard and custom ROS messages.
- Working with ROS services and actionlib.
- Maintaining and releasing your ROS packages.
- Creating a wiki page for ROS packages.

Creating a ROS package

The ROS packages are the basic unit of the ROS system. We can create a ROS package, build it, and release it to the public. The current distribution of ROS we are using is kinetic. 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, `rosbuild` was the build system. Because of the various flaws of `rosbuild`, `catkin` came into existence, which is basically based on **Cross Platform Make** (**CMake**). This has a lot of advantages, such as porting the package into another operating system, such as Windows. If an OS supports CMake and Python, `catkin` -based packages can be easily ported into it.

The first requirement work with ROS packages is to create a ROS `catkin` workspace. After installed ROS, we can create and build a `catkin_workspace` called `catkin_ws`:

```
$ mkdir -p ~/catkin_ws/src
```

To compile the workspace, we should source the ROS environment, in order to get access to ROS functions:

```
$ source /opt/ros/kinetic/setup.bash
```

Switch to the source, `src` folder previously created.

```
$ 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
```

This last command will create a `devel` and a `build` directory in the catkin workspace. Inside the `devel` folder different setup files are located. To add the created ROS workspace to the ROS environment, we should source one of this file. In addition, we can source the setup file of this workspace every time that a new `bash` session starts with the following command:

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

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. 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.

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

```
$ catkin_create_pkg package_name [dependency1] [dependency2]
```

Source code folder: All ROS packages, either created from scratch or downloaded from other code repositories, must be placed in the `src` folder of the ROS workspace, otherwise they can not be recognized by the ROS system and compiled.

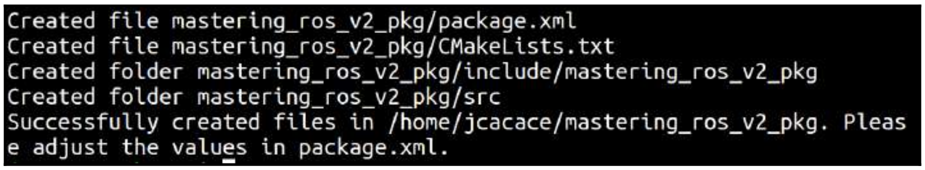
Here is the command to create the sample ROS package:

```
$ 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` metapackage provides interfaces to create preemptible 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.

After package creation, additional dependencies can be added manually by editing the `CMakeLists.txt` and `package.xml` files. We will get the following message if the package has been successfully created:



```
Created file mastering_ros_v2_pkg/package.xml
Created file mastering_ros_v2_pkg/CMakeLists.txt
Created folder mastering_ros_v2_pkg/include/mastering_ros_v2_pkg
Created folder mastering_ros_v2_pkg/src
Successfully created files in /home/jcacace/mastering_ros_v2_pkg. Pleas
e adjust the values in package.xml.
```

Figure 1: Terminal messages while creating a ROS package

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:

```
~/catkin_ws $ catkin_make
```

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 `mastering_ros_demo_pkg` folder, joining the `/src` subdirectory for the source code. `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 the 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 the standard message definition of primitive datatypes. `std_msgs/Int32.h` contains the 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;
```

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

```
 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 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:

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

This is used to set the frequency of sending data:

```
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:

```
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:

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

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

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

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

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

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

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

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

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:

```
#include "ros/ros.h"
#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:

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

This is the header needed for the subscribers:

```
void number_callback(const std_msgs::Int32::ConstPtr& msg) {
   ROS_INFO("Recieved [%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:

```
 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, the buffer size, and the `callback` function. We are subscribing the `/numbers` topic and we have already seen the `callback` function in the preceding section:

```
 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 `mastering_ros_demo_pkg` to view the existing `CMakeLists.txt` file. The following code snippet in this file is responsible for building these two nodes:

```
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 a workspace:

```
$ cd ~/catkin_ws
```

Build `mastering_ros_demo_package` as follows:

```
$ catkin_make
```

We can either use the preceding command to build the entire workspace, or use the the `–DCATKIN_WHITELIST_PACKAGES` option. With this option, it is possible to set one or more packages to compile:

```
$ catkin_make –DCATKIN_WHITELIST_PACKAGES="pkg1,pkg2,..."
```

Note that is necessary to revert this configuration to compile other packages or the entire workspace. This can be done using the following command:

```
$ catkin_make –DCATKIN_WHITELIST_PACKAGES=""
```

If the building is done, we can execute the nodes. First, start `roscore`:

```
$ roscore
```

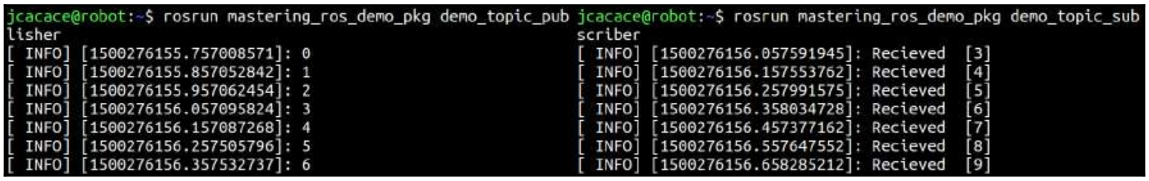
Now run both commands in two shells. In the running publisher:

```
$ rosrun mastering_ros_demo_package demo_topic_publisher
```

In the running subscriber:

```
$ rosrun mastering_ros_demo_package demo_topic_subscriber
```

We can see the output as shown here:



```
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_topic_pub jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_topic_sub
lisher                                                        scriber
[ INFO] [1500276155.757008571]: 0                             [ INFO] [1500276156.057591945]: Recieved  [3]
[ INFO] [1500276155.857052842]: 1                             [ INFO] [1500276156.157553762]: Recieved  [4]
[ INFO] [1500276155.957062454]: 2                             [ INFO] [1500276156.257991575]: Recieved  [5]
[ INFO] [1500276156.057095824]: 3                             [ INFO] [1500276156.358034728]: Recieved  [6]
[ INFO] [1500276156.157087268]: 4                             [ INFO] [1500276156.457377162]: Recieved  [7]
[ INFO] [1500276156.257505796]: 5                             [ INFO] [1500276156.557647552]: Recieved  [8]
[ INFO] [1500276156.357532737]: 6                             [ INFO] [1500276156.658285212]: Recieved  [9]
```

Figure 2: Running topic publisher and subscriber

The following diagram shows how the nodes communicate with each other. We can see that the `demo_topic_publisher` node publishes the `/numbers` topic and then subscribes to the `demo_topic_subscriber` node:

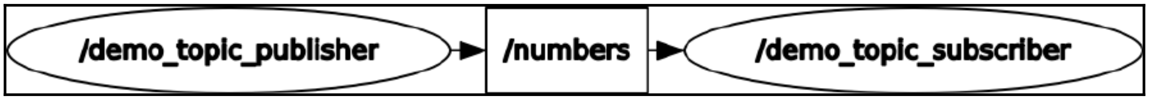


Figure 3: Graph of the communication between publisher and subscriber nodes

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 will look at how to create custom messages and services definitions in the current package. The message definitions are stored in a `.msg` file and the service definitions 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:

```
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 `<exec_depend>message_runtime</exec_depend>`.

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

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

Uncomment the following line and add the custom message file:

```
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 `mastering_ros_demo_pkg/1` folder for these code.

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

```
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
```

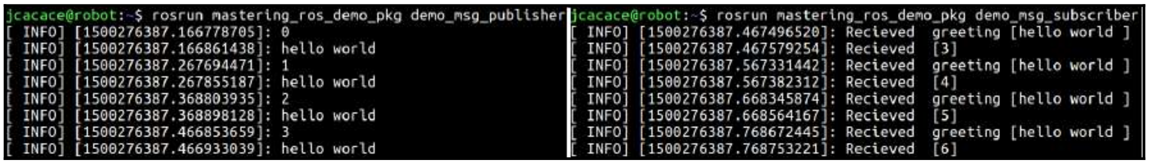
- Start the custom message publisher node:

```
$ rosrun mastering_ros_demo_pkg demo_msg_publisher
```

- Start the custom message subscriber node:

```
$ rosrun 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:



```
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_msg_publisher
[ INFO] [1500276387.166778705]: 0
[ INFO] [1500276387.166861438]: hello world
[ INFO] [1500276387.267694471]: 1
[ INFO] [1500276387.267855187]: hello world
[ INFO] [1500276387.368803935]: 2
[ INFO] [1500276387.368898128]: hello world
[ INFO] [1500276387.466853659]: 3
[ INFO] [1500276387.466933039]: hello world
```

```
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_msg_subscriber
[ INFO] [1500276387.467496520]: Recieved  greeting [hello world ]
[ INFO] [1500276387.467579254]: Recieved  [3]
[ INFO] [1500276387.567331442]: Recieved  greeting [hello world ]
[ INFO] [1500276387.567382312]: Recieved  [4]
[ INFO] [1500276387.668345874]: Recieved  greeting [hello world ]
[ INFO] [1500276387.668564167]: Recieved  [5]
[ INFO] [1500276387.768672445]: Recieved  greeting [hello world ]
[ INFO] [1500276387.768753221]: Recieved  [6]
```

Figure 4: Running publisher and subscriber using custom message definitions.

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

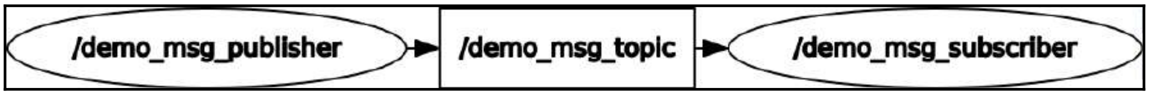


Figure 5: Graph of the communication between message publisher and subscriber

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:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

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

```
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:

```
## 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 `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:

```
#include "ros/ros.h"
#include "mastering_ros_demo_pkg/demo_srv.h"
#include <iostream>
#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 an explanation of the code:

```
#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 we can use this in our code. The `sstream.h` is for getting string streaming classes:

```
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 with 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:

```
 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 `demo_srv.srv`. This response will go to the service client node:

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

This creates a service called `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 `demo_service_client.cpp` is working.

Here is the definition of this code:

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

  }
  else
  {
   ROS_ERROR("Failed to call service");
   return 1;
  }

 ros::spinOnce();
 loop_rate.sleep();

 }
 return 0;
}
```

Let's explain the code:

```
  ros::ServiceClient client =
 n.serviceClient<mastering_ros_demo_pkg::demo_srv>("demo_service");
```

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

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

This line will create a new service object instance:

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

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

```
   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 will do nothing:

```
    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:

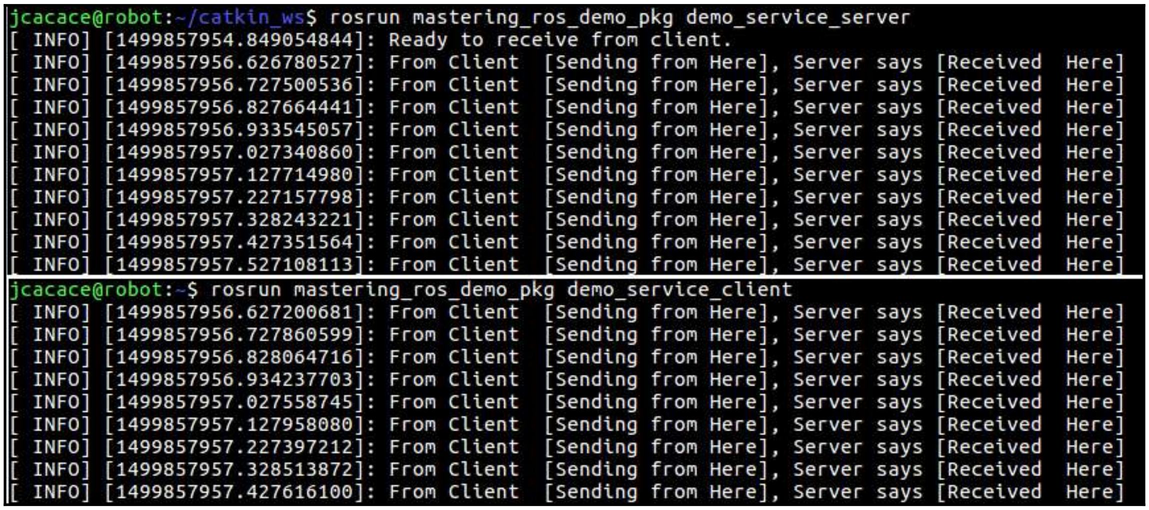
```
add_executable(demo_service_server src/demo_service_server.cpp)
add_executable(demo_service_client src/demo_service_client.cpp)

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 the 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
```



```
jcacace@robot:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_service_server
[ INFO] [1499857954.849054844]: Ready to receive from client.
[ INFO] [1499857956.626780527]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.727500536]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.827664441]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.933545057]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.027340860]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.127714980]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.227157798]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.328243221]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.427351564]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.527108113]: From Client  [Sending from Here], Server says [Received  Here]
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_service_client
[ INFO] [1499857956.627200681]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.727860599]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.828064716]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857956.934237703]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.027558745]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.127958080]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.227397212]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.328513872]: From Client  [Sending from Here], Server says [Received  Here]
[ INFO] [1499857957.427616100]: From Client  [Sending from Here], Server says [Received  Here]
```

Figure 6: Running ROS service client and server nodes.

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 if the reply takes too much time or the server is not finished with the given work, we have to wait until it completes, blocking the main application while waiting for the termination of the requested action. In addition, the calling client could be implemented to monitor the execution of the remote process. In these cases, we should implement our application using `actionlib`. This 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.

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 callback 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 acknowledgment. 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 1 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 `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 with the header files:

```
#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 accessing our action definition:

```
class Demo_actionAction
{
```

This class contains the action server definition:

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

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

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

Create a feedback instance for sending feedback during the operation:

```
mastering_ros_demo_pkg::Demo_actionResult result;
```

Create a result instance for sending the final result:

```
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:

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

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

```
 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:

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

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

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

Inside this loop, it will check whether the action server is active or it is preempted. If it occurs, the function will return:

```
 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:

```
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 workings 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:

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

In the 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:

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

This will create an action client instance:

```
 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:

```
 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:

```
 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:

```
 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:

```
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():

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

We should add Boost as a system dependency:

```
## Generate actions in the 'action' folder
 add_action_files(
  FILES
  Demo_action.action
 )
```

We need to add our action file in add_action_files():

```
## 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()`:

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

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

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

```
##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
```

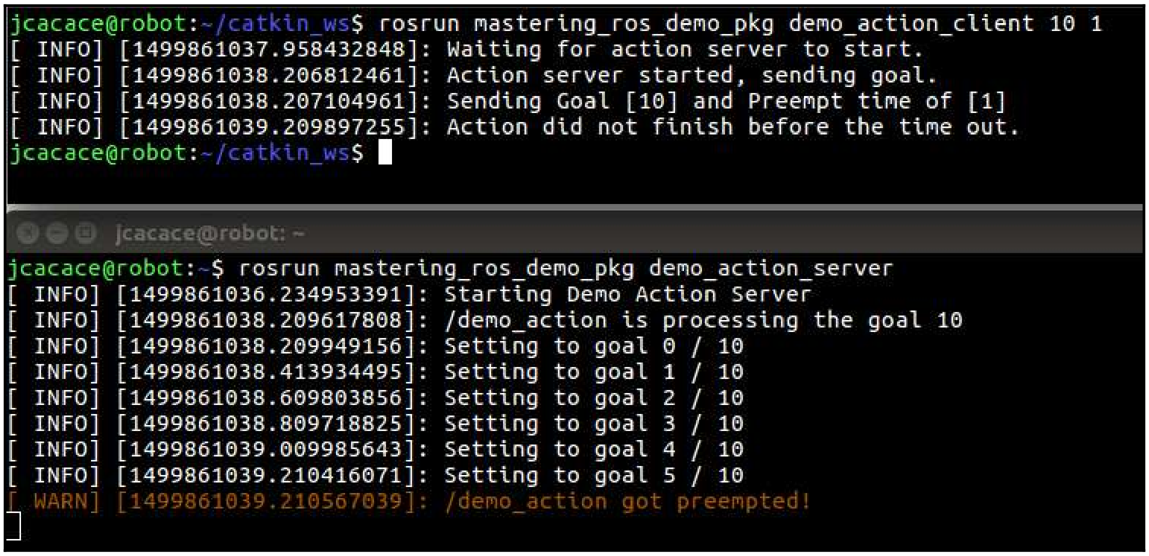
- Launch the action server node:

```
$rosrun mastering_ros_demo_pkg demo_action_server
```

- Launch the action client node:

```
$rosrun mastering_ros_demo_pkg demo_action_client 10 1
```

The output of these process is shown as follows:



```
jcacace@robot:~/catkin_ws$ rosrun mastering_ros_demo_pkg demo_action_client 10 1
[ INFO] [1499861037.958432848]: Waiting for action server to start.
[ INFO] [1499861038.206812461]: Action server started, sending goal.
[ INFO] [1499861038.207104961]: Sending Goal [10] and Preempt time of [1]
[ INFO] [1499861039.209897255]: Action did not finish before the time out.
jcacace@robot:~/catkin_ws$
```

```
jcacace@robot:~$ rosrun mastering_ros_demo_pkg demo_action_server
[ INFO] [1499861036.234953391]: Starting Demo Action Server
[ INFO] [1499861038.209617808]: /demo_action is processing the goal 10
[ INFO] [1499861038.209949156]: Setting to goal 0 / 10
[ INFO] [1499861038.413934495]: Setting to goal 1 / 10
[ INFO] [1499861038.609803856]: Setting to goal 2 / 10
[ INFO] [1499861038.809718825]: Setting to goal 3 / 10
[ INFO] [1499861039.009985643]: Setting to goal 4 / 10
[ INFO] [1499861039.210416071]: Setting to goal 5 / 10
[ WARN] [1499861039.210567039]: /demo_action got preempted!
```

Figure 7: Running ROS actionlib server and client.

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, we can write all nodes inside an 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 the 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:

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

Paste the following content into the file:

```
<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></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:

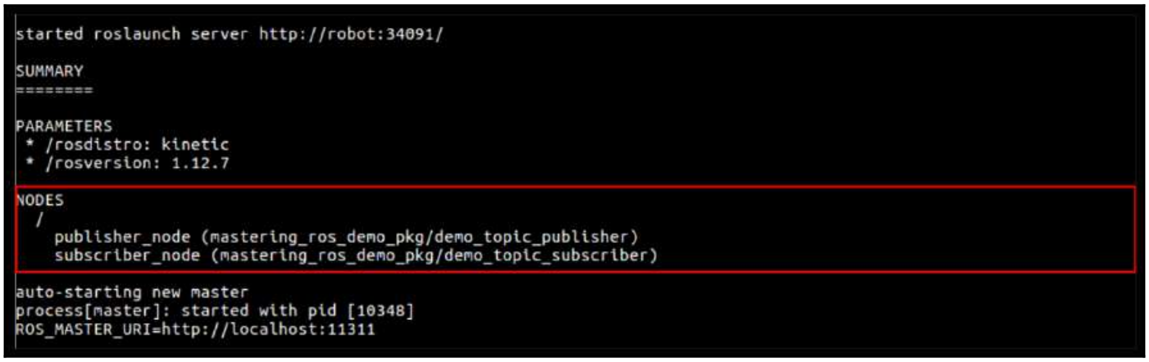
```
 <node name="publisher_node" pkg="mastering_ros_demo_pkg"
type="demo_topic_publisher" output="screen"/>
```

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:

```
$ roslaunch mastering_ros_demo_pkg demo_topic.launch
```

Here is the output we get if the launch is successful:



```
started roslaunch server http://robot:34091/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES
  /
    publisher_node (mastering_ros_demo_pkg/demo_topic_publisher)
    subscriber_node (mastering_ros_demo_pkg/demo_topic_subscriber)

auto-starting new master
process[master]: started with pid [10348]
ROS_MASTER_URI=http://localhost:11311
```

Figure 8: Terminal messages while launching the `demo_topic`.launch file.

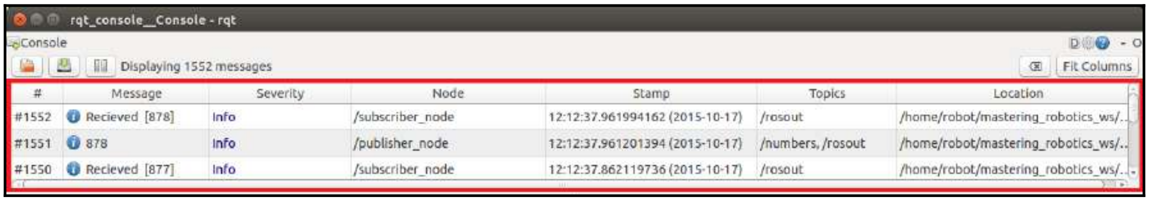
We can check the list of nodes using:

```
$ rosnode list
```

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

```
$ rqt_console
```

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



#	Message	Severity	Node	Stamp	Topics	Location
#1552	Recieved [878]	Info	/subscriber_node	12:12:37.961994162 (2015-10-17)	/rosout	/home/robot/mastering_robotics_ws/...
#1551	878	Info	/publisher_node	12:12:37.961201394 (2015-10-17)	/numbers, /rosout	/home/robot/mastering_robotics_ws/...
#1550	Recieved [877]	Info	/subscriber_node	12:12:37.862119736 (2015-10-17)	/rosout	/home/robot/mastering_robotics_ws/...

Figure 9: 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**: Streaming continuous data flow, such as sensor data. For example, stream joypad data to teleoperate a robot, publish robot odometry, publish video stream from a camera.
- **Services**: Executing procedures that terminate quickly. For example, save calibration parameter of sensors, save a map generated by the robot during its navigation, or load a parameter file.
- **Actionlib**: Execute long and complex actions managing their feedback. For example, navigate towards a target or plan a motion path.

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

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

Maintaining the ROS package

Most ROS packages are maintained using a **Version Control System** (**VCS**) such as Git, Subversion (svn), Mercurial (hg), and so on. A collection of packages that share a common VCS can be called a repository. The package in the repository can be released using a catkin release automation tool called bloom. Most 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 is important for all software, especially open source applications. 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/jocacace/mastering_ros_demo_pkg`.

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/`). 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 for 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/jocacace/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`.

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 to 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 will check whether the package contains change logs, versions, and so on. The following command makes our package consistent and recommended for 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.

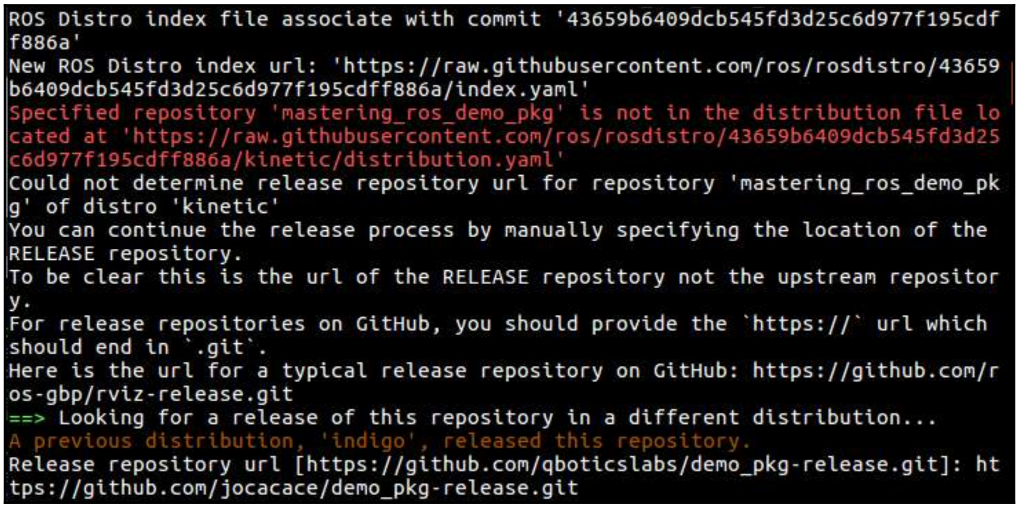
Releasing our package

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 kinetic --track kinetic 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`.

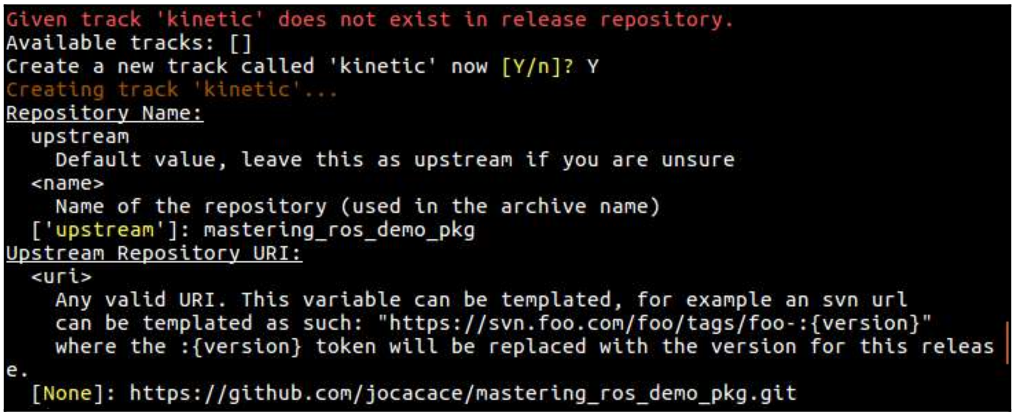
The following message will be displayed when there is no reference of the package in `rosdistro`:



```
ROS Distro index file associate with commit '43659b6409dcb545fd3d25c6d977f195cdf
f886a'
New ROS Distro index url: 'https://raw.githubusercontent.com/ros/rosdistro/43659
b6409dcb545fd3d25c6d977f195cdff886a/index.yaml'
Specified repository 'mastering_ros_demo_pkg' is not in the distribution file lo
cated at 'https://raw.githubusercontent.com/ros/rosdistro/43659b6409dcb545fd3d25
c6d977f195cdff886a/kinetic/distribution.yaml'
Could not determine release repository url for repository 'mastering_ros_demo_pk
g' of distro 'kinetic'
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 repositor
y.
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/r
os-gbp/rviz-release.git
==> Looking for a release of this repository in a different distribution...
A previous distribution, 'indigo', released this repository.
Release repository url [https://github.com/qboticslabs/demo_pkg-release.git]: ht
tps://github.com/jocacace/demo_pkg-release.git
```

Figure 10: Terminal messages when there is no reference of the package in `rosdistro`

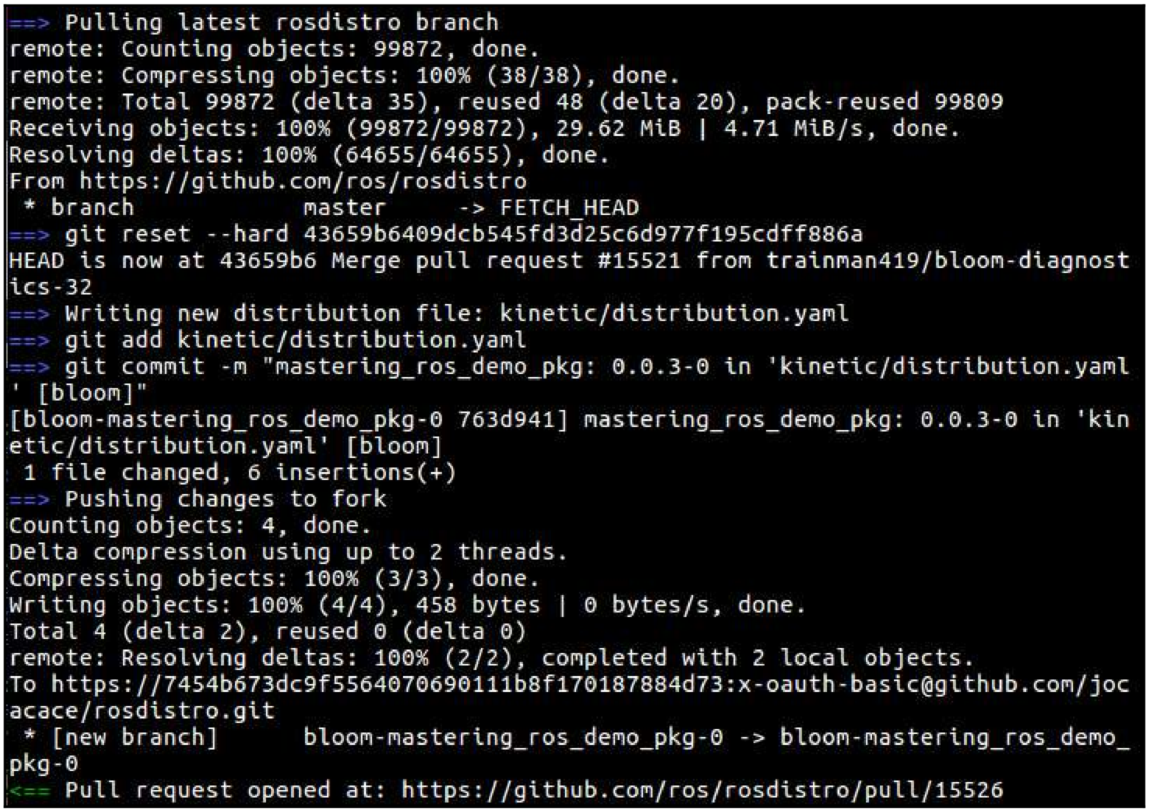
We should give the release repository in the terminal that is marked in red in the preceding screenshot. In this case, the URL was `https://github.com/jocacace/demo_pkg-release`:



```
Given track 'kinetic' does not exist in release repository.
Available tracks: []
Create a new track called 'kinetic' now [Y/n]? Y
Creating track 'kinetic'...
Repository Name:
  upstream
    Default value, leave this as upstream if you are unsure
  <name>
    Name of the repository (used in the archive name)
  ['upstream']: mastering_ros_demo_pkg
Upstream Repository URI:
  <uri>
    Any valid URI. This variable can be templated, for example an svn url
    can be templated as such: "https://svn.foo.com/foo/tags/foo-:{version}"
    where the :{version} token will be replaced with the version for this releas
e.
  [None]: https://github.com/jocacace/mastering_ros_demo_pkg.git
```

Figure 11: Inputting the release repository URL

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:



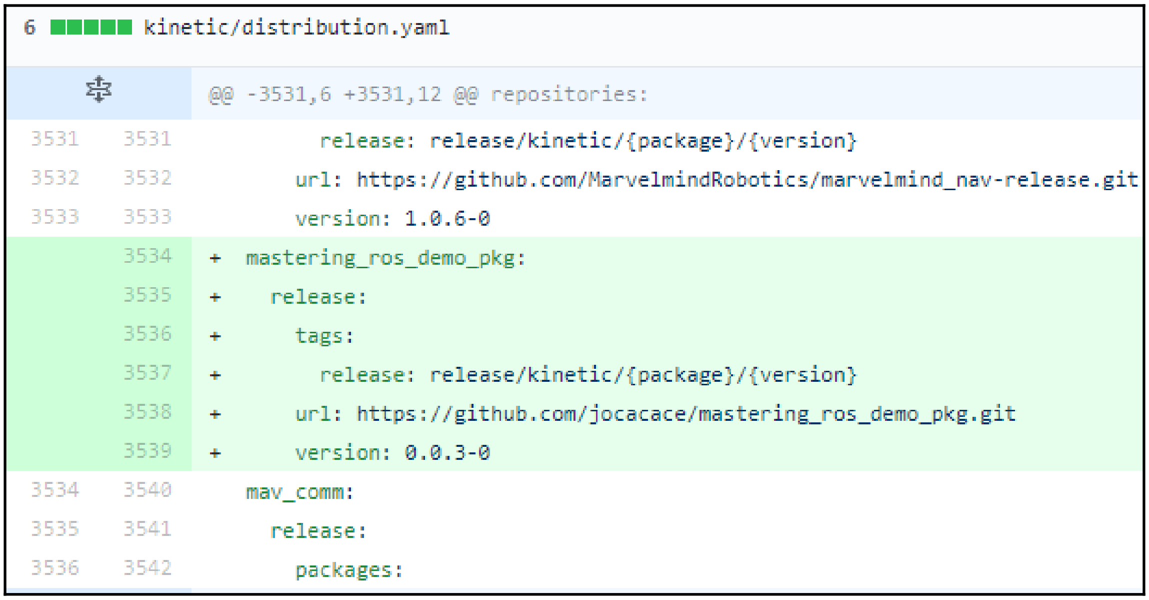
```
==> Pulling latest rosdistro branch
remote: Counting objects: 99872, done.
remote: Compressing objects: 100% (38/38), done.
remote: Total 99872 (delta 35), reused 48 (delta 20), pack-reused 99809
Receiving objects: 100% (99872/99872), 29.62 MiB | 4.71 MiB/s, done.
Resolving deltas: 100% (64655/64655), done.
From https://github.com/ros/rosdistro
 * branch            master     -> FETCH_HEAD
==> git reset --hard 43659b6409dcb545fd3d25c6d977f195cdff886a
HEAD is now at 43659b6 Merge pull request #15521 from trainman419/bloom-diagnost
ics-32
==> Writing new distribution file: kinetic/distribution.yaml
==> git add kinetic/distribution.yaml
==> git commit -m "mastering_ros_demo_pkg: 0.0.3-0 in 'kinetic/distribution.yaml
' [bloom]"
[bloom-mastering_ros_demo_pkg-0 763d941] mastering_ros_demo_pkg: 0.0.3-0 in 'kin
etic/distribution.yaml' [bloom]
 1 file changed, 6 insertions(+)
==> Pushing changes to fork
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 458 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://7454b673dc9f5564070690111b8f170187884d73:x-oauth-basic@github.com/joc
acace/rosdistro.git
 * [new branch]      bloom-mastering_ros_demo_pkg-0 -> bloom-mastering_ros_demo_
pkg-0
<== Pull request opened at: https://github.com/ros/rosdistro/pull/15526
```

Figure 12: Sending a pull request to rosdistro

The `pull` request for this package can be viewed at `https://github.com/ros/rosdistro/pull/15526`.

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

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



```
6 kinetic/distribution.yaml
@@ -3531,6 +3531,12 @@ repositories:
          release: release/kinetic/{package}/{version}
        url: https://github.com/MarvelmindRobotics/marvelmind_nav-release.git
        version: 1.0.6-0
+   mastering_ros_demo_pkg:
+     release:
+       tags:
+         release: release/kinetic/{package}/{version}
+       url: https://github.com/jocacace/mastering_ros_demo_pkg.git
+       version: 0.0.3-0
    mav_comm:
      release:
        packages:
```

Figure 13: The distribution.yaml file of ROS kinetic.

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

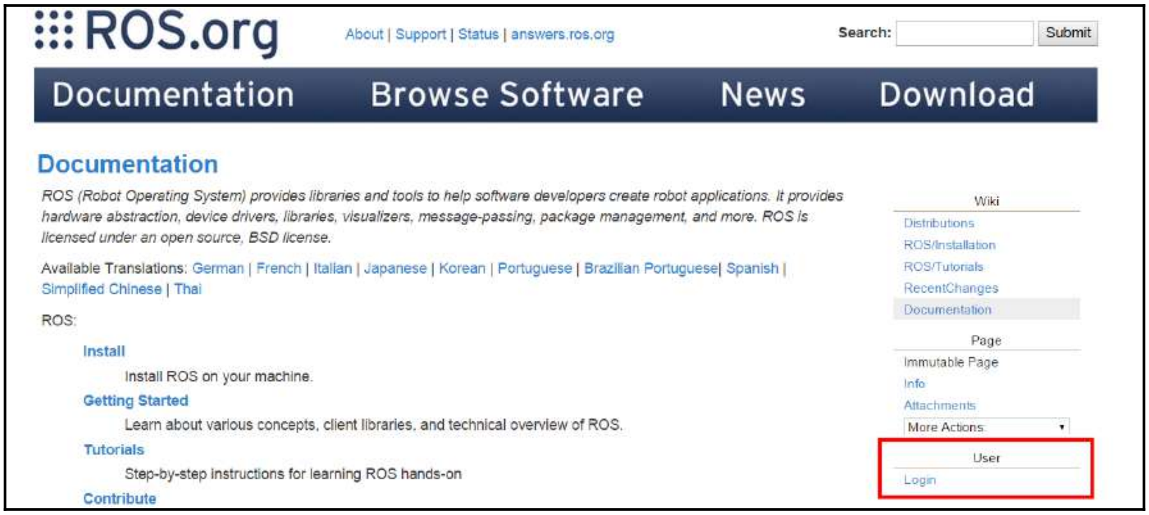
Creating a Wiki page for your ROS package

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` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit: `http://www.packtpub.com/support` and register to have the files emailed directly to you. You can also download chapter codes from: `https://github.com/jocacace/mastering_ros_2nd_ed.git`; this code repository contains the link to all other code repositories used in this book.

The first step is to register in wiki using your e-mail address. Go to `wiki.ros.org`, and click on the **Login** button, as shown in the screenshot:

>


ROS.org

About | Support | Status | answers.ros.org

Search: Submit

Documentation Browse Software News Download

Documentation

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

Available Translations: German | French | Italian | Japanese | Korean | Portuguese | Brazilian Portuguese| Spanish | Simplified Chinese | Thai

ROS:

Install
Install ROS on your machine.
Getting Started
Learn about various concepts, client libraries, and technical overview of ROS.
Tutorials
Step-by-step instructions for learning ROS hands-on
Contribute

Wiki
Distributions
ROS/Installation
ROS/Tutorials
RecentChanges
Documentation

Page
Immutable Page
Info
Attachments
More Actions

User
Login

Figure 14: Locating the login option from ROS wiki

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

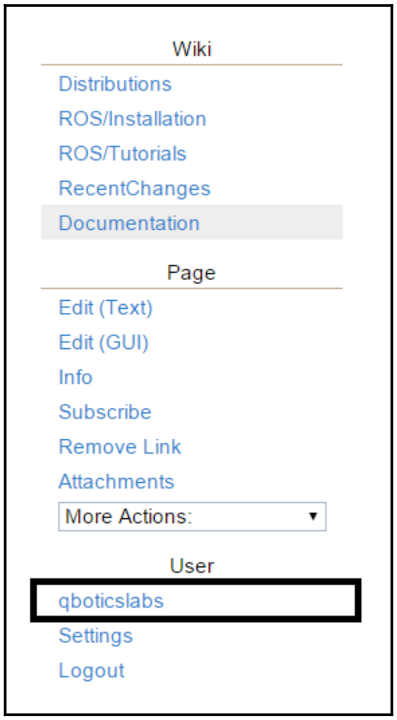
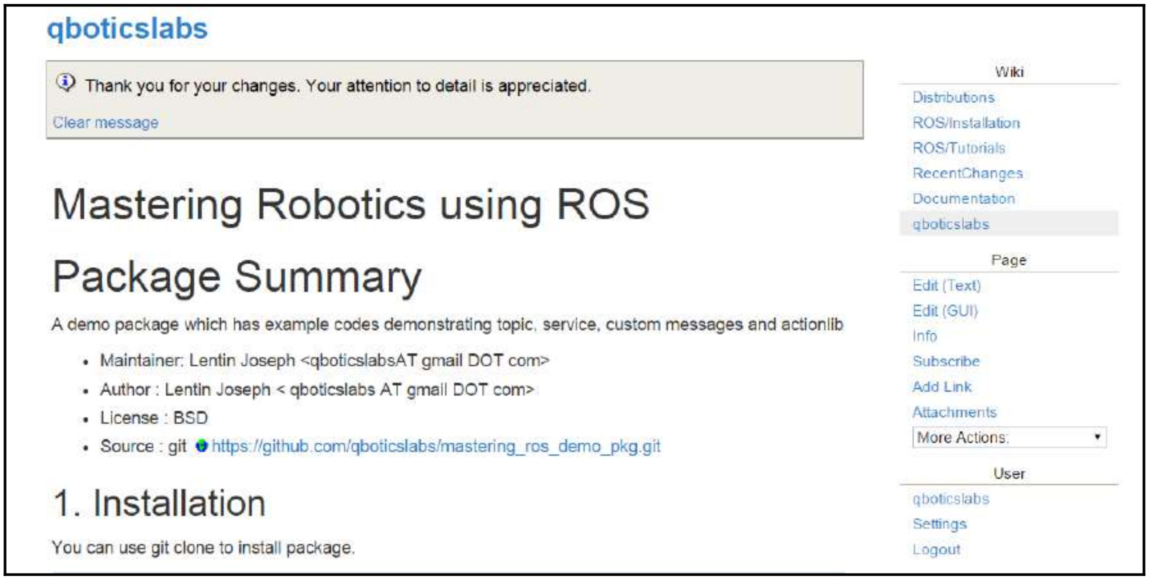


Figure 15: Locating the user account button from ROS wiki

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:



Thank you for your changes. Your attention to detail is appreciated.

Mastering Robotics using ROS

Package Summary

A demo package which has example codes demonstrating topic, service, custom messages and actionlib

- Maintainer: Lentin Joseph <qboticslabsAT gmail DOT com>
- Author : Lentin Joseph < qboticslabs AT gmail DOT com>
- License : BSD
- Source : git https://github.com/qboticslabs/mastering_ros_demo_pkg.git

1. Installation

You can use git clone to install package.

Figure 16: Creating a new wiki page.

Questions

After going through the chapter, you should now be able to answer the following questions:

- Which kind of communication protocols between nodes are supported by ROS?
- What is the difference between `rosrun` and `roslaunch` commands?
- How do ROS topics and services differ in their operations?
- How do ROS services and `actionlib` differ in their operations?

Summary

In this chapter, we provided different examples of ROS nodes in which ROS features such as ROS topics, services, and actions were implemented. We discussed how to create and compile ROS packages using custom and standard messages. After demonstrating the workings 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.