Chapter 4

Important Concepts of ROS

In order to develop a robot related to ROS[1], it is necessary to understand the essential components and concepts of ROS. This chapter will introduce the terminology used in ROS and the important concepts of ROS such as message communication, message file, name, coordinate transformation (TF), client library, communication between heterogeneous devices, file system, and build system.

4.1. ROS Terminology

This section explains the most frequently used ROS terms. Use this section as a ROS glossary. Many terms may be new to the reader and even if there are unfamiliar terms, look over the definition and move on. You will become more familiar with the concepts as you engage with examples and exercises in each of the following chapters.

ROS

ROS provides standard operating system services such as hardware abstraction, device drivers, implementation of commonly used features including sensing, recognizing, mapping, motion planning, message passing between processes, package management, visualizers and libraries for development as well as debugging tools.

Master

The master[2] acts as a name server for node-to-node connections and message communication. The command roscore is used to run the master, and if you run the master, you can register the name of each node and get information when needed. The connection between nodes and message communication such as topics and services are impossible without the master.

The master communicates with slaves using XMLRPC (XML-Remote Procedure Call)[3], which is an HTTP-based protocol that does not maintain connectivity. In other words, the slave nodes can access only when they need to register their own information or request information of other nodes. The connection status of each other is not checked regularly. Due to this feature, ROS can be used in very large and complex environments. XMLRPC is very lightweight and supports a variety of programming languages, making it well suited for ROS, which supports variety of hardware and programming languages.

When you execute ROS, the master will be configured with the URI address and port configured in the ROS_MASTER_URI. By default, the URI address uses the IP address of local PC, and port number 11311, unless otherwise modified.

1 http://wiki.ros.org/ROS/Concepts
2 http://wiki.ros.org/Master
3 https://en.wikipedia.org/wiki/XML–RPC

Node

A node[4] refers to the smallest unit of processor running in ROS. Think of it as one executable program. ROS recommends creating one single node for each purpose, and it is recommended to develop for easy reusability. For example, in case of mobile robots, the program to operate the robot is broken down into specialized functions. Specialized node is used for each function such as sensor drive, sensor data conversion, obstacle recognition, motor drive, encoder input, and navigation.

Upon startup, a node registers information such as name, message type, URI address and port number of the node. The registered node can act as a publisher, subscriber, service server or service client based on the registered information, and nodes can exchange messages using topics and services.

The node uses XMLRPC for communicating with the master and uses XMLRPC or TCPROS[5] of the TCP/IP protocols when communicating between nodes. Connection request and response between nodes use XMLRPC, and message communication uses TCPROS because it is a direct communication between nodes independent from the master. As for the URI address and port number, a variable called ROS_HOSTNAME, which is stored on the computer where the node is running, is used as the URI address, and the port is set to an arbitrary unique value.

Package

A package[6] is the basic unit of ROS. The ROS application is developed on a package basis, and the package contains either a configuration file to launch other packages or nodes. The package also contains all the files necessary for running the package, including ROS dependency libraries for running various processes, datasets, and configuration file. The number of official packages is about 2,500 for ROS Indigo as of July 2017 (http://repositories.ros.org/status_page/ ros_indigo_default.html) and about 1,600 packages for ROS Kinetic (http://repositories.ros.org/status_page/ros_kinetic_default.html). In addition, although there could be some redundancies, there are about 4,600 packages developed and released by users (http://rosindex.github.io/stats/).

Metapackage

A metapackage[7] is a set of packages that have a common purpose. For example, the Navigation metapackage consists of 10 packages including AMCL, DWA, EKF, and map_server.

4 http://wiki.ros.org/Nodes
5 http://wiki.ros.org/ROS/TCPROS
6 http://wiki.ros.org/Packages
7 http://wiki.ros.org/Metapackages

Message

A node[8] sends or receives data between nodes via a message. Messages are variables such as integer, floating point, and boolean. Nested message structure that contains another messages or an array of messages can be used in the message.

TCPROS and UDPROS communication protocol is used for message delivery. Topic is used in unidirectional message delivery while service is used in bidirectional message delivery that request and response are involved.

Topic

The topic[9] is literally like a topic in a conversation. The publisher node first registers its topic with the master and then starts publishing messages on a topic. Subscriber nodes that want to receive the topic request information of the publisher node corresponding to the name of the topic registered in the master. Based on this information, the subscriber node directly connects to the publisher node to exchange messages as a topic.

Publish and Publisher

The term 'publish' stands for the action of transmitting relative messages corresponding to the topic. The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic. The publisher is declared in the node and can be declared multiple times in one node.

Subscribe and Subscriber

The term 'subscribe' stands for the action of receiving relative messages corresponding to the topic. The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topic from the master. Based on received publisher information, the subscriber node directly requests connection to the publisher node and receives messages from the connected publisher node. A subscriber is declared in the node and can be declared multiple times in one node.

The topic communication is an asynchronous communication which is based on publisher and subscriber, and it is useful to transfer certain data. Since the topic continuously transmits and receives stream of messages once connected, it is often used for sensors that must periodically transmit data. On the other hands, there is a need for synchronous communication with which request and response are used. Therefore, ROS provides a message synchronization method called 'service'. A service consists of the service server that responds to requests and the service client that requests to respond. Unlike the topic, the service is a one-time message

8 http://wiki.ros.org/Messages
9 http://wiki.ros.org/Topics

communica tion. When the request and response of the service is completed, the connection between two nodes is disconnected.

Service

The service[10] is synchronous bidirectional communication between the service client that requests a service regarding a particular task and the service server that is responsible for responding to requests.

Service Server

The 'service server' is a server in the service message communication that receives a request as an input and transmits a response as an output. Both request and response are in the form of messages. Upon the service request, the server performs the designated service and delivers the result to the service client as a response. The service server is implemented in the node that receives and executes a given request.

Service Client

The 'service client' is a client in the service message communication that requests service to the server and receives a response as an input. Both request and response are in the form of message. The client sends a request to the service server and receives the response. The service client is implemented in the node which requests specified command and receives results.

Action

The action[11] is another message communication method used for an asynchronous bidirectional communication. Action is used where it takes longer time to respond after receiving a request and intermediate responses are required until the result is returned. The structure of action file is also similar to that of service. However, feedback data section for intermediate response is added along with goal and result data section which are represented as request and response in service respectively. There are action client that sets the goal of the action and action server that performs the action specified by the goal and returns feedback and result to the action client.

Action Server

The 'action server' is in charge of receiving goal from the client and responding with feedback and result. Once the server receives goal from the client, it performs predefined process.

10 http://wiki.ros.org/Services
11 http://wiki.ros.org/actionlib

Action Client

The 'action client' is in charge of transmitting the goal to the server and receives result or feedback data as inputs from the action server. The client delivers the goal to the action server, then receives corresponding result or feedback, and transmits follow up instructions or cancel instruction.

Parameter

The parameter[12] in ROS refers to parameters used in the node. Think of it as *.ini configuration files in Windows program. Default values are set in the parameter and can be read or written if necessary. In particular, it is very useful when configured values can be modified in real-time. For example, you can specify settings such as USB port number, camera calibration parameters, maximum and minimum values of the motor speed.

Parameter Server

When parameters are called in the package, they are registered with the parameter server[13] which is loaded in the master.

Catkin

The catkin[14] refers to the build system of ROS. The build system basically uses CMake (Cross Platform Make), and the build environment is described in the 'CMakeLists.txt' file in the package folder. CMake was modified in ROS to create a ROS-specific build system. Catkin started the alpha test from ROS Fuerte and the core packages began to switch to Catkin in the ROS Groovy version. Catkin has been applied to most packages in the ROS Hydro version. The Catkin build system makes it easy to use ROS-related builds, package management, and dependencies among packages. If you are going to use ROS at this point, you should use Catkin instead of ROS build (rosbuild).

ROS Build

The ROS build (rosbuild)[15] is the build system that was used before the Catkin build system. Although there are some users who still use it, this is reserved for compatibility of ROS, therefore, it is officially not recommended to use. If an old package that only supports the rosbuild must be used, we recommend using it after converting rosbuild to catkin.

12 http://wiki.ros.org/Parameter%20Server#Parameters
13 http://wiki.ros.org/Parameter%20Server
14 http://wiki.ros.org/catkin
15 http://wiki.ros.org/rosbuild

roscore

roscore[16] is the command that runs the ROS master. If multiple computers are within the same network, it can be run from another computer in the network. However, except for special case that supports multiple roscore, only one roscore should be running in the network. When ROS master is running, the URI address and port number assigned for ROS_MASTER_URI environment variables are used. If the user has not set the environment variable, the current local IP address is used as the URI address and port number 11311 is used which is a default port number for the master.

rosrun

rosrun[17] is the basic execution command of ROS. It is used to run a single node in the package. The node uses the ROS_HOSTNAME environment variable stored in the computer on which the node is running as the URI address, and the port is set to an arbitrary unique value.

roslaunch

While rosrun is a command to execute a single node, roslaunch[18] in contrast executes multiple nodes. It is a ROS command specialized in node execution with additional functions such as changing package parameters or node names, configuring namespace of nodes, setting ROS_ROOT and ROS_PACKAGE_PATH, and changing environment variables[19] when executing nodes.

roslaunch uses the '*.launch' file to define which nodes to be executed. The file is based on XML (Extensible Markup Language) and offers a variety of options in the form of XML tags.

bag

The data from the ROS messages can be recorded. The file format used is called bag[20], and '*.bag' is used as the file extension. In ROS, bag can be used to record messages and play them back when necessary to reproduce the environment when messages are recorded. For example, when performing a robot experiment using a sensor, sensor values are stored in the message form using the bag. This recorded message can be repeatedly loaded without performing the same test by playing the saved bag file. Record and play functions of rosbag are especially useful when developing an algorithm with frequent program modifications.

16 http://wiki.ros.org/roscore
17 http://wiki.ros.org/rosbash#rosrun
18 http://wiki.ros.org/roslaunch
19 http://wiki.ros.org/ROS/EnvironmentVariables
20 http://wiki.ros.org/Bags

ROS Wiki

ROS Wiki is a basic description of ROS based on Wiki (http://wiki.ros.org/) that explains each package and the features provided by ROS. This Wiki page describes the basic usage of ROS, a brief description of each package, parameters used, author, license, homepage, repository, and tutorial. The ROS Wiki currently has more than 18,800 pages of content.

Repository

An open package specifies repository in the Wiki page. The repository is a URL address on the web where the package is saved. The repository manages issues, development, downloads, and other features using version control systems such as svn, hg, and git. Many of currently available ROS packages are using GitHub[21] as repositories for source code. In order to view the contents of the source code for each package, check the corresponding repository.

Graph

The relationship between nodes, topics, publishers, and subscribers introduced above can be visualized as a graph. The graphical representation of message communication does not include the service as it only happens one time. The graph can be displayed by running the 'rqt_graph' node in the 'rqt_graph' package. There are two execution commands, 'rqt_graph' and 'rosrun rqt_graph rqt_graph'.

Name

Nodes, parameters, topics, and services all have names[22]. These names are registered on the master and searched by the name to transfer messages when using the parameters, topics, and services of each node. Names are flexible because they can be changed when being executed, and different names can be assigned when executing identical nodes, parameters, topics, and services multiple times. Use of names makes ROS suitable for large-scale projects and complex systems.

Client Library

ROS provides development environments for various languages by using client library[23] in order to reduce the dependency on the language used. The main client libraries are C++, Python, Lisp, and other languages such as Java, Lua, .NET, EusLisp, and R are also supported. For this purpose, client libraries such as roscpp, rospy, roslisp, rosjava, roslua, roscs, roseus, PhaROS, and rosR have been developed.

21 http://www.github.com/
22 http://wiki.ros.org/Names
23 http://wiki.ros.org/Client%20Libraries

URI

A URI (Uniform Resource Identifier) is a unique address that represents a resource on the Internet. The URI is one of basic components that enables interaction with Internet and is used as an identifier in the Internet protocol.

MD5

MD5 (Message-Digest algorithm 5)[24] is a 128-bit cryptographic hash function. It is used primarily to verify data integrity, such as checking whether programs or files are in its unmodified original form. The integrity of the message transmission/reception in ROS is verified with MD5.

RPC

RPC (Remote Procedure Call)[25] stands for the function that calls a sub procedure on a remote computer from another computer in the network. RPC uses protocols such as TCP/IP and IPX, and allows execution of functions or procedures without having the developer to write a program for remote control.

XML

XML (Extensible Markup Language) is a broad and versatile markup language that W3C recommends for creating other special purpose markup languages. XML utilizes tags in order to describe the structure of data. In ROS, it is used in various components such as *.launch, *.urdf, and package.xml.

XMLRPC

XMLRPC (XML-Remote Procedure Call) is a type of RPC protocol that uses XML as the encoding format and uses the request and response method of the HTTP protocol which does not maintain nor check the connection. XMLRPC is a very simple protocol, used only to define small data types or commands. As a result, XMLRPC is very lightweight and supports a variety of programming languages, making it well suited for ROS, which supports a variety of hardware and languages.

TCP/IP

TCP stands for Transmission Control Protocol. It is often called TCP/IP. The Internet protocol layer guarantees data transmission using TCP, which is based on the IP (Internet Protocol) layer in the Internet Protocol Layers. It guarantees the sequential transmission and reception of data.

24 https://en.wikipedia.org/wiki/Md5sum
25 http://wiki.ros.org/ROS/Technical%20Overview

TCPROS is a message format based on TCP/IP and UDPROS is a message format based on UDP. TCPROS is more frequently used in ROS.

CMakeLists.txt

Catkin, which is the build system of ROS, uses CMake by default. The build environment is specified in the 'CMakeLists.txt'[26] file in each package folder.

package.xml

An XML file[27] contains package information that describes the package name, author, license, and dependent packages.

4.2. Message Communication

So far, we have only given an introduction to ROS but not explained in detail how ROS works. In this section, we will take a look at the core functions and concepts of ROS. For the detailed description of each term used in ROS concept description, refer to the glossary of terms discussed earlier. This section will only deal with ROS concepts. The actual programming method is covered in Chapter 7, ROS Basic Programming.

As described in Chapter 2, ROS is developed in unit of nodes, which is the minimum unit of executable program that has broken down for the maximum reusability. The node exchanges data with other nodes through messages forming a large program as a whole. The key concept here is the message communication methods among nodes. There are three different methods of exchanging messages: a topic which provides a unidirectional message transmission/reception, a service which provides a bidirectional message request/response and an action which provides a bidirectional message goal/result/feedback. In addition, the parameters used in the node can be modified from the outside of node. This can also be considered as a type of message communication in the larger context. Message communication is illustrated in Figure 4-1 and the differences are summarized in Table 4-1. It is important to use each topic, service, action, and parameter according to its correct purpose when programming on ROS.

26 http://wiki.ros.org/catkin/CMakeLists.txt
27 http://wiki.ros.org/catkin/package.xml

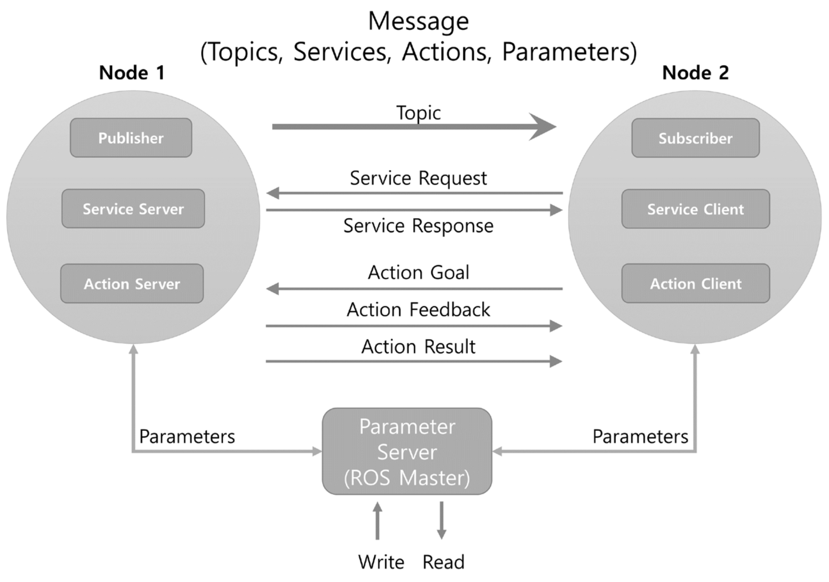


FIGURE 4-1 Message Communication between Nodes

Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difficult to use the service due to long response times after the request or when an intermediate feedback value is needed

TABLE 4-1 Comparison of the Topic, Server, and Action

4.2.1. Topic

Communication on topic uses the same type of message for both publisher and subscriber as shown in Figure 4-2. The subscriber node receives the information of publisher node corresponding to the identical topic name registered in the master. Based on this information, the subscriber node directly connects to the publisher node to receive messages. For example, if the current position of the robot is generated in the form of odometry[28] information by calculating the encoder values of both wheels of the mobile robot, the asynchronous odometry information can be continuously transmitted in unidirectional flow using a topic message(x, y,

28 http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom

i). As topics are unidirectional and remain connected to continuously send or receive messages, it is suitable for sensor data that requires publishing messages periodically. In addition, multiple subscribers can receive message from a publisher and vice versa. Multiple publishers and subscribers connections are available as well.

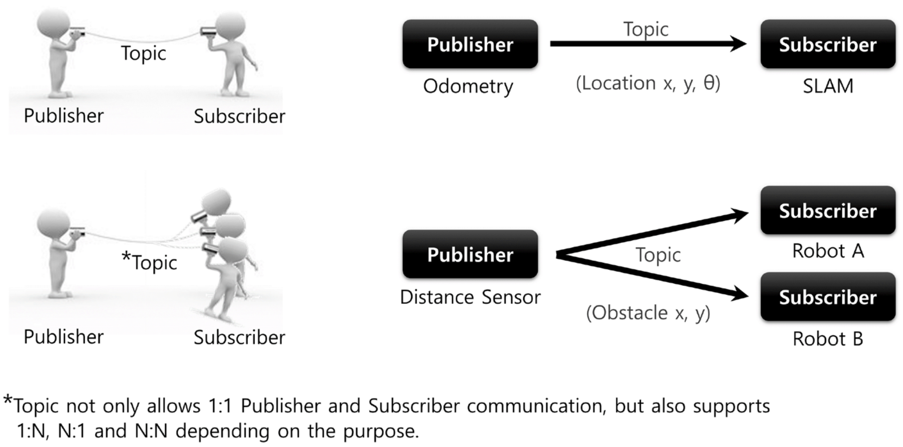


FIGURE 4-2 Topic Message Communication

4.2.2. Service

Communication on service is a bidirectional synchronous communication between the service client requesting a service and the service server responding to the request as shown in Figure 4-3. The aforementioned 'publish' and 'subscribe' of the topic is an asynchronous method which is advantageous on periodical data transmission. On the other hands, there is a need for synchronous communication which uses request and response. Accordingly, ROS provides a synchronized message communication method called 'service'.

A service consists of a service server that responds only when there is a request and a service client that can send requests as well as receiving responses. Unlike the topic, the service is one-time message communication. Therefore, when the request and response of the service are completed, the connection between two nodes will be disconnected. A service is often used to command a robot to perform a specific action or nodes to perform certain events with a specific condition. Service does not maintain the connection, so it is useful to reduce the load of the network by replacing topic. For example, if the client requests the server for the current time as shown in Figure 4-3, the server will check the time and respond to the client, and the connection is terminated.

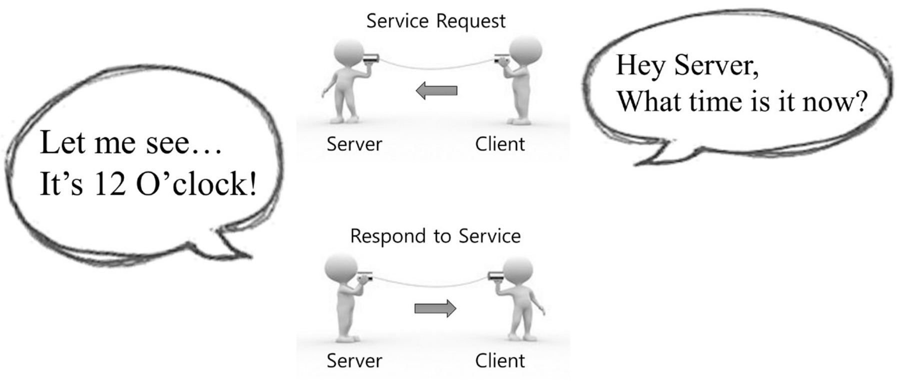


FIGURE 4-3 Service Message Communication

4.2.3. Action

Communication on action[29] is used when a requested goal takes a long time to be completed, therefore progress feedback is necessary. This is very similar to the service where 'goals' and 'results' correspond to 'requests' and 'responses' respectively. In addition, the 'feedback' is added to report feedbacks to the client periodically when intermediate values are needed. The message transmission method is the same as the asynchronous topic. The feedback transmits an asynchronous bidirectional message between the action client which sets the goal of the action and an action server that performs the action and sends the feedback to the action client. For example, as shown in Figure 4-4, if the client sets home-cleaning tasks as a goal to the server, the server informs the user of the progress of the dishwashing, laundry, cleaning, etc. in the form of feedback, and finally sends the final message to the client as a result. Unlike the service, the action is often used to command complex robot tasks such as canceling transmitted goal while the operation is in progress.

29 http://wiki.ros.org/actionlib

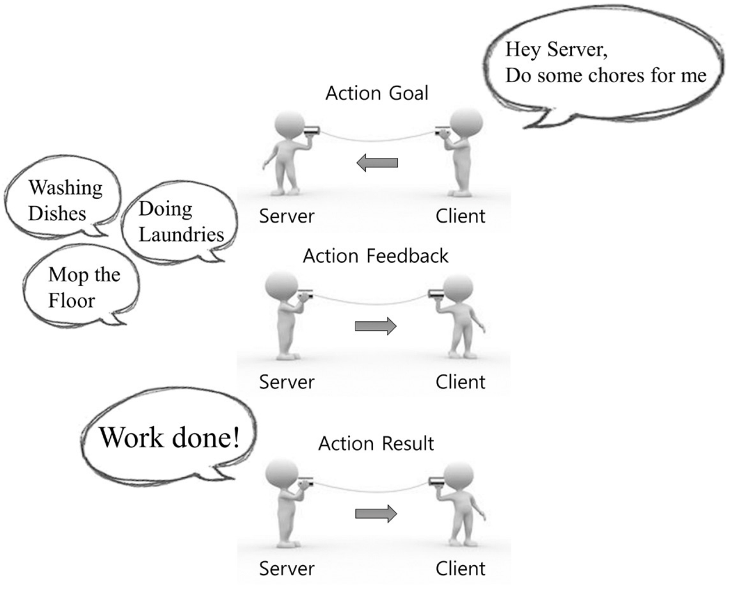


FIGURE 4-4 Action Message Communication

The publisher, the subscriber, the service server, the service client, the action server, and the action client can be implemented in separate nodes. In order to exchange messages among these nodes, the connection has to be established first with the help of a master. A master acts like a name server as it keeps names of nodes, topics, services and action as well as the URI address, port number and parameters. In other words, nodes register their own information with the master upon launch, and acquire relative information of other nodes from the master. Then, each node directly connects to each other to perform message communication. This is shown in Figure 4-5.

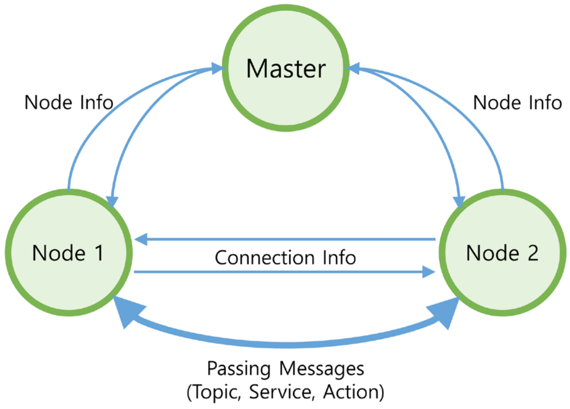


FIGURE 4-5 Message Communication

4.2.4. Parameter

Message communication is largely divided into topics, services, and actions. Parameters are global variables used in nodes and in the larger context, they can also be considered as a message communication. In Windows programs *.ini file is used to save configurations just as parameters in ROS. The configuration are set with default values and can be read or written externally if necessary. Especially, the configured values can be modified in real-time from the outside by using the write function. It is very useful to flexibly cope with changing environment.

Although parameters are not strictly a message communication method, I think that they belong to the scope of message communication in that they use messages. For example, you can change parameters to set the USB port to connect to, get the camera color correction value, and configure the maximum and minimum values of the speed and commands.

4.2.5. Message Communication Flow

The master manages the information of the nodes, and each node connects and communicates with other nodes as needed. Let's learn about the most important communication sequence of the master, nodes, topics, services, and action messages.

Running the Master

A master that manages connection information in a message communication between nodes is an essential element that must be run first in order to use ROS. The ROS master is run by using the 'roscore' command and runs the server with XMLRPC. The master registers the name of nodes, topics, services, action, message types, URI addresses and ports for node-to-node connections, and relays the information to other nodes upon request.

```
$ roscore
```

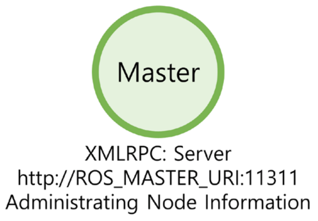


FIGURE 4-6 Running the Master

Running the Subscriber Node

Subscriber nodes are launched with either a 'rosrun' or 'roslaunch' commands. The subscriber node registers its node name, topic name, message type, URI address, and port with the master as it runs. The master and node communicate using XMLRPC.

```
$ rosrun PACKAGE_NAME NODE_NAME
$ roslaunch PACKAGE_NAME LAUNCH_NAME
```

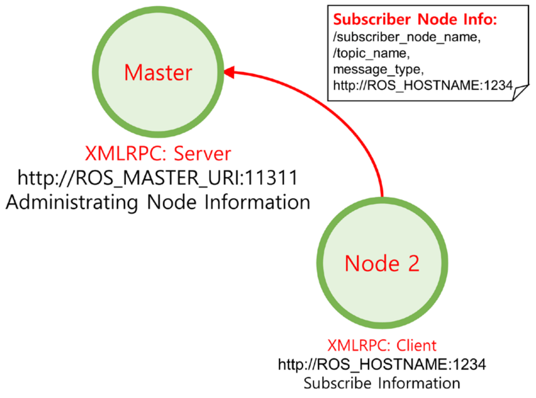


FIGURE 4-7 Running the Subscriber Node

Running the Publisher Node

Publisher nodes, like subscriber nodes, are executed by 'rosrun' or 'roslaunch' commands. The publisher node registers its node name, topic name, message type, URI address and port with the master. The master and node communicate using XMLRPC.

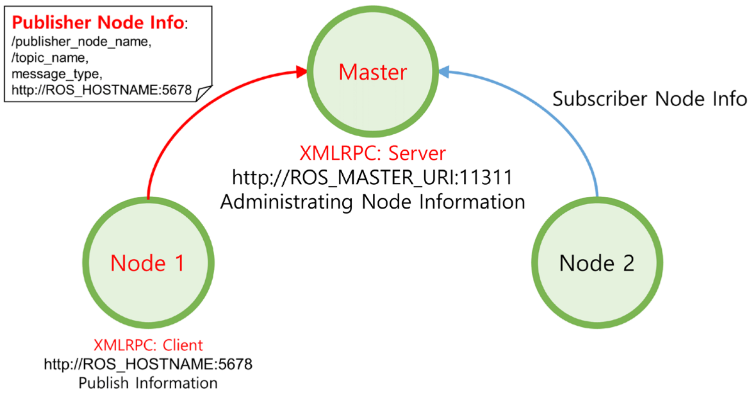
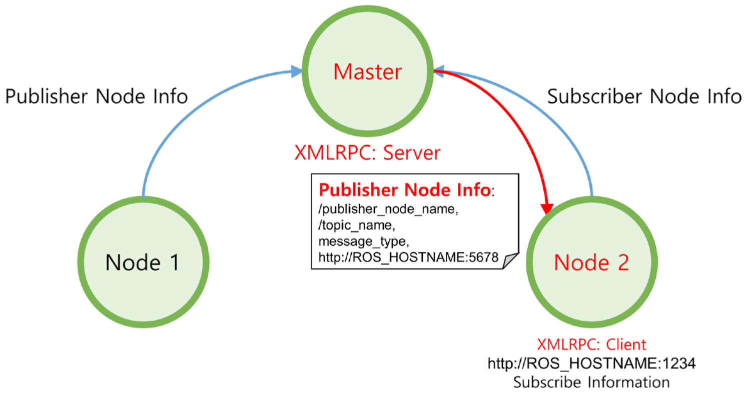


FIGURE 4-8 Running the Publisher Node

Providing Publisher Information

The master distributes information such as the publisher's name, topic name, message type, URI address and port number of the publisher to subscribers that want to connect to the publisher node. The master and node communicate using XMLRPC.



Publisher Node Info:
/publisher_node_name,
/topic_name,
message_type,
http://ROS_HOSTNAME:5678

FIGURE 4-9 Provide Publisher Node Information to Subscriber Node

Connection Request from the Subscriber Node

The subscriber node requests a direct connection to the publisher node based on the publisher information received from the master. During the request procedure, the subscriber node transmits information to the publisher node such as the subscriber node's name, the topic name, and the message type. The publisher node and the subscriber node communicate using XMLRPC.

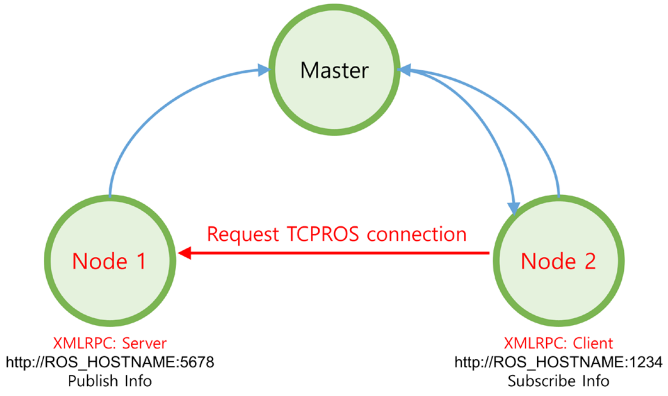


FIGURE 4-10 Connection Request from the Subscriber Node

Connection Response from the Publisher Node

The publisher node sends the URI address and port number of its TCP server in response to the connection request from the subscriber node. The publisher node and the subscriber node communicate using XMLRPC.

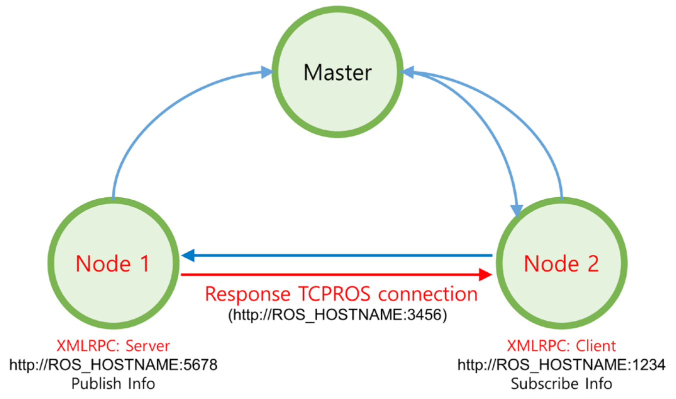


FIGURE 4-11 Connection Response from the Publisher Node

TCPROS Connection

The subscriber node creates a client for the publisher node using TCPROS, and connects to the publisher node. At this point, the communication between nodes uses TCP/IP based protocol called TCPROS.

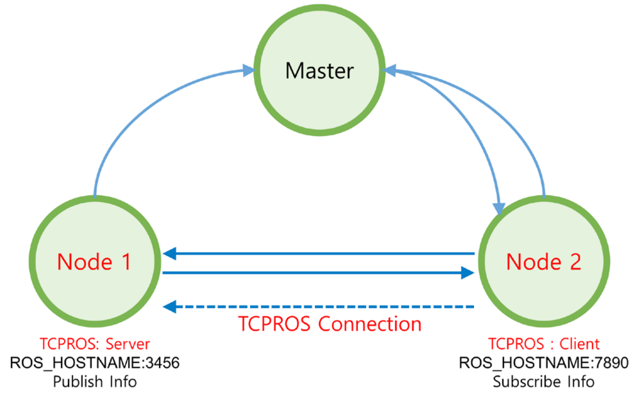


FIGURE 4-12 TCP Connection

Message Transmission

The publisher node transmits a predefined message to the subscriber node. The communication between nodes uses TCPROS.

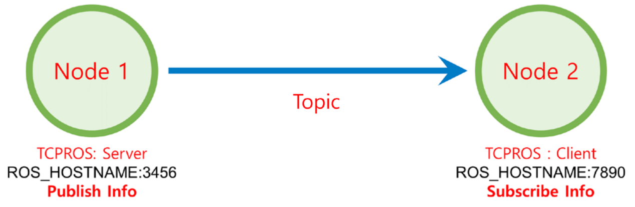


FIGURE 4-13 Topic Message Transmission

Service Request and Response

The procedures discussed above correspond to the communication on 'topic'. Topic communication publishes and subscribes messages continuously, unless the publisher or subscriber is terminated. There are two types of services.

- Service Client: Request service and receive response
- Service Server: Receive a service, execute the specified task, and return a response

The connection between the service server and the client is the same as the TCPROS connection for the publisher and subscriber described above. Unlike the topic, the service terminates connection after successful request and response. If additional request is necessary, the connection procedure must be carried out again.

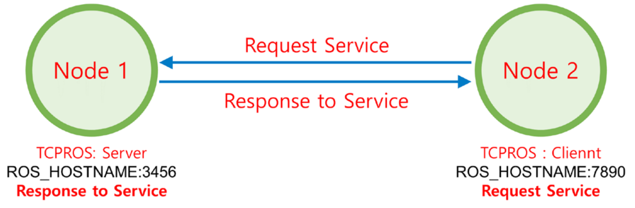


FIGURE 4-14 Service Request and Response

Action Goal, Result, Feedback

Action may look similar to the request and the response of the service with an additional feedback message in order to provide intermediate result between the request (goal) and the response(result), but in practice it is rather more like a topic. In fact, if you use the 'rostopic' command to list up topics, there are five topics such as goal, status, cancel, result, and feedback that are used in the action. The connection between the action server and the client is similar to the TCPROS connection of the publisher and subscriber, but the usage is slightly different. For example, when an action client sends a cancel command or the server sends a result value, the connection will be terminated.

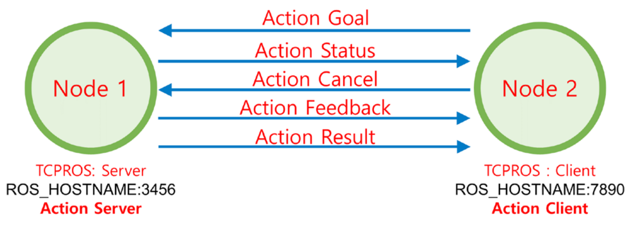


FIGURE 4-15 Action Message Communication

We previously tested ROS with 'turtlesim'. In this test, the master and two nodes were used, and the '/turtle1/cmd_vel' topic was used between the two nodes to pass the translational and rotational messages to the virtual TurtleBot. Putting this in perspective with the ROS concept described above, it can be represented as in Figure 4-16.

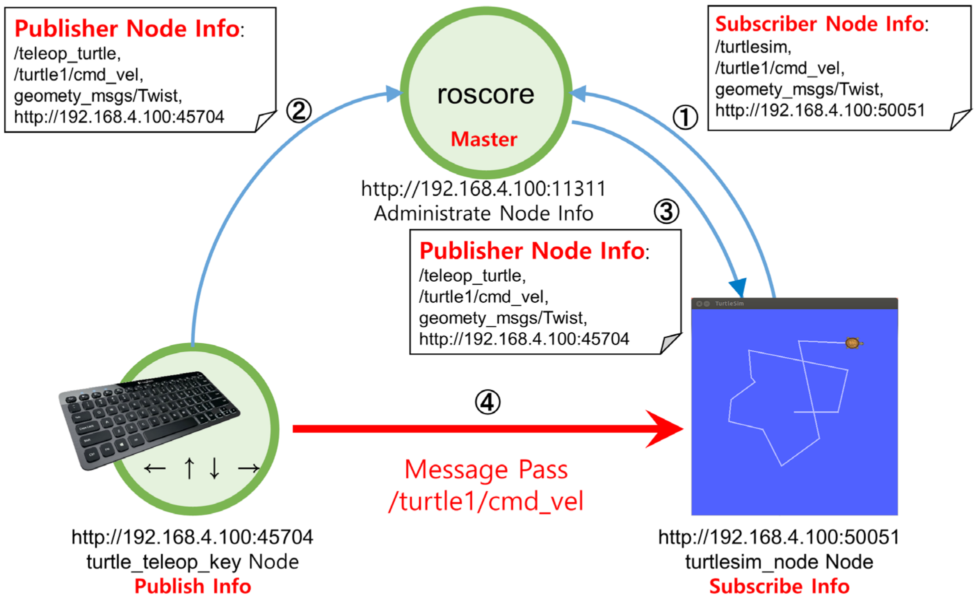


FIGURE 4-16 Example of Message Communication

4.3. Message

A message[30] is a bundle of data used to exchange data between nodes. The topics, services, and actions are using messages to communicate. A message can include basic data types such as integer, floating point, Boolean as well as message arrays such as 'float32[] ranges', 'Point32[10] points'. Moreover, a message can contain other messages such as 'geometry_msgs/PoseStamped'. Also, the header 'std_msgs/Header' which is commonly used in ROS can be included in the message. These messages can be described as field types and field names as shown below.

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

The ROS data type shown in Table 4-2 can be used in the field type. The following example is the simplest form of message, and you can use an array for the field type as shown in Table 4-3. Embedded messages in the message are also commonly used.

```
int32 x
int32 y
```

ROS Data Type	Serialization	C++ Data Type	Python Data Type
bool	unsigned 8-bit int	uint8_t	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int
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	std::string	str
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time

30 http://wiki.ros.org/msg

ROS Data Type	Serialization	C++ Data Type	Python Data Type
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Table 4-2 Basic data types for messages in ROS, serialization methods, corresponding C ++ and Python data types

ROS Data Type	Serialization	C++ Data Type	Python Data Type
fixed-length	no extra serialization	boost::array, std::vector	tuple
variable-length	uint32 length prefix	std::vector	tuple
uint8[]	uint32 length prefix	std::vector	bytes
bool[]	uint32 length prefix	std::vector<uint8_t>	list of bool

Table 4-3 How to use ROS message data types as an array, corresponding C ++ and Python data types

The header (std_msgs/Header), which is commonly used in ROS, can also be used as a message. The Header.msg file in std_msgs[31] contains the sequence ID, time stamp, and frame ID, and use them to probe the message or measure the time.

std_msgs/Header.msg

```
# Sequence ID: Messages are sequentially incremented by 1.
uint32 seq
# Timestamp: Has two child attributes, the stamp.sec for second and the stamp.nsec for
nanosecond.
time stamp
# Stores the Frame ID
string frame_id
```

The following shows how actually to use a message in the ROS program. For example, in the case of the 'teleop_turtle_key' node of the turtlesim package, which we tested in Chapter 3, the translational speed (meter/sec) and rotational speed (radian/sec) is sent as a message to the turtlesim node according to the directional keys (←, →, ↑, ↓) entered from the keyboard. The TurtleBot moves on the screen using the received speed values. The message used at this time is the 'twist'[32] message in 'geometry_msgs'.

31 http://wiki.ros.org/std_msgs
32 http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html

```
Vector3  linear
Vector3  angular
```

In the message structure above, ‘linear’ and ‘angular’ values are declared as a Vector3 type. This is the similar form to the nested message as the Vector3 is a message type in the ‘geometry_msgs’[33]. The Vector3[34] contains the following data.

```
float64 x
float64 y
float64 z
```

In other words, six topics published from the ‘teleop_turtle_key’ node are linear.x, linear.y, linear.z, angular.x, angular.y, and angular.z. All of these are float64 type which is one of the basic data types described in ROS. With these data, arrow keys of the keyboard can be converted to the translational speed (meter/sec) and the rotational speed (radian/sec) message, so that the TurtleBot could be controlled.

The topic, service, and action described in the previous section use messages. Although they are similar in the form and the concept, they are divided into three types according to their usage. This will be discussed in more detail in the following section.

4.3.1. msg File

The ‘msg’ file is the message file used by topics, with has the file extension of ‘*.msg’. The ‘Twist’[35] message in the ‘geometry_msgs’ described above is an example of message. Such msg file consists of field types and field names.

geometry_msgs/Twist.msg

```
Vector3  linear
Vector3  angular
```

4.3.2. srv File

The ‘srv’ file is the message file used by services, with the file extension of ‘*.srv’. For example, the SetCameraInfo[36] message in the ‘sensor_msgs’ described above is a typical srv file. The major

33 http://docs.ros.org/api/geometry_msgs/html/index-msg.html
34 http://docs.ros.org/api/geometry_msgs/html/msg/Vector3.html
35 http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html
36 http://docs.ros.org/api/sensor_msgs/html/srv/SetCameraInfo.html

difference from the msg file is that the series of three hyphens (---) serve as a delimiter; the upper message being the service request message and the lower message being the service response message.

sensor_msgs/SetCameraInfo.srv

```
sensor_msgs/CameraInfo camera_info
---
bool success
string status_message
```

4.3.3. action File

The action message file[37] is the message file used by actions[38], with the file extension of '*.action'. Unlike msg and srv, it is relatively uncommon message file, so there is no typical example of the message file, but can be used as shown in following example. The major difference from the msg and srv files is that the series of three hyphens (---) are used in two places as delimiters, the first being the goal message, the second being the result message, and the third being the feedback message. The biggest difference of the action file is the feedback message feature. The goal message and the result message of the action file can be compared to the request and the response message of the srv file mentioned above, but the additional feedback message of the action file is used to send feedback while the designated process is being performed. As describe in the following example, when the starting position of 'start_pose' and the goal position of 'goal_pose' of the robot are transmitted as request values, the robot moves to the received goal position and returns the 'result_pose'. While the robot is moving to the goal position, the 'percent_complete' message periodically transmits feedback values showing the progress in the form of the percentage of the goal point reached.

```
geometry_msgs/PoseStamped start_pose
geometry_msgs/PoseStamped goal_pose
---
geometry_msgs/PoseStamped result_pose
---
float32 percent_complete
```

37 http://wiki.ros.org/actionlib_msgs
38 http://wiki.ros.org/actionlib

4.4. Name

ROS has an abstract data type called 'graph' as its basic concept[39]. This graph shows the connection relationship between each node and the relationship of messages (data) sent and received with arrows. To do this, messages and parameters used in nodes, topics, and services in ROS all have unique names[40]. Let's take a closer look at the names of topics. The name of the topic is divided into the relative method, the global method and the private method as shown in Table 4-4.

The topic is usually declared as shown in the following code. This will be covered in more detail in Chapter 7. Here, let's modify the topic's name in order to understand how to use names.

```
int main(int argc, char **argv)            // Node Main Function
{
  ros::init(argc, argv, "node1");          // Node Name Initialization
  ros::NodeHandle nh;                      // Node Handle Declaration
  // Publisher Declaration, Topic Name = bar
  ros::Publisher node1_pub = nh.advertise<std_msg::Int32>("bar", 10);
```

In the above example, the name of the node is '/node1'. If the publisher is declared as a 'bar' without any symbols, the topic will have the relative name '/bar'. Even if the slash(/) character is used to declare in global, the topic name will still be '/bar'.

```
ros::Publisher node1_pub =  nh.advertise<std_msg::Int32>("/bar", 10);
```

However, if you declare the name as private using the tilde(~) character, the topic name becomes '/node1/bar'.

```
ros::Publisher node1_pub =  nh.advertise<std_msg::Int32>("~bar", 10);
```

The declaration of the name can vary as shown in Table 4-4. The '/wg' means a change of the namespace. This is discussed in more detail in the next section.

39 http://wiki.ros.org/ROS/Concepts
40 http://wiki.ros.org/Names

Node	Relative (Default)	Global	Private
/node1	bar → /bar	/bar → /bar	~bar → /node1/bar
/wg/node2	bar → /wg/bar	/bar → /bar	~bar → /wg/node2/bar
/wg/node3	foo/bar → /wg/foo/bar	/foo/bar → /foo/bar	~foo/bar → /wg/node3/foo/bar

TABLE 4-4 Naming Rule

How can two cameras be run? Simply executing the related node twice will terminate the previously executed node, due to the fact that there must be a unique name in ROS. However, achieving two cameras does not require you to run a separate program or change the source code. Simply change the name of the node when running it by using either namespaces or remapping.

To help you understand, suppose you have a virtual 'camera_package'. Suppose that the camera node is executed when the 'camera_node' of 'camera_package' is executed, the way to run this is as follows.

```
$rosrun camera_package camera_node
```

If 'camera_node' transmits the image data of the camera via the image topic, this image topic can be received with 'rqt_image_view' as follows

```
$rosrun rqt_image_view rqt_image_view
```

Now let's modify the topic values of these nodes by remapping. The following command will change the topic name as '/front/image'. In the below command, the 'image' is the topic name of 'camera_node' and the below example shows how to change the topic name by setting options in execution commands.

```
$ rosrun camera_package camera_node image:=front/image
$ rosrun rqt_image_view rqt_image_view image:=front/image
```

For example, if there are three cameras, such as front, left, and right, when the multiple nodes are executed under the same name, there will be conflicted names and therefore the previously executed node gets terminated. Therefore, nodes with the same name can be executed in the following way. Below, the name option is followed by consecutive underscores(__). Options such as '__ns', '__name', '__log', '__ip', '__hostname', and '__master' are special options used when running the node. Also, single underscore(_) is placed in front of the topic name if it is used as a private.

```
$ rosrun camera_package camera_node __name:=front _device:=/dev/video0
$ rosrun camera_package camera_node __name:=left _device:=/dev/video1
$ rosrun camera_package camera_node __name:=right _device:=/dev/video2
$ rosrun rqt_image_view rqt_image_view
```

The following example will bind the nodes and topics into a single namespace. This ensures that all nodes and topics are grouped into a single namespace and all names are changed accordingly.

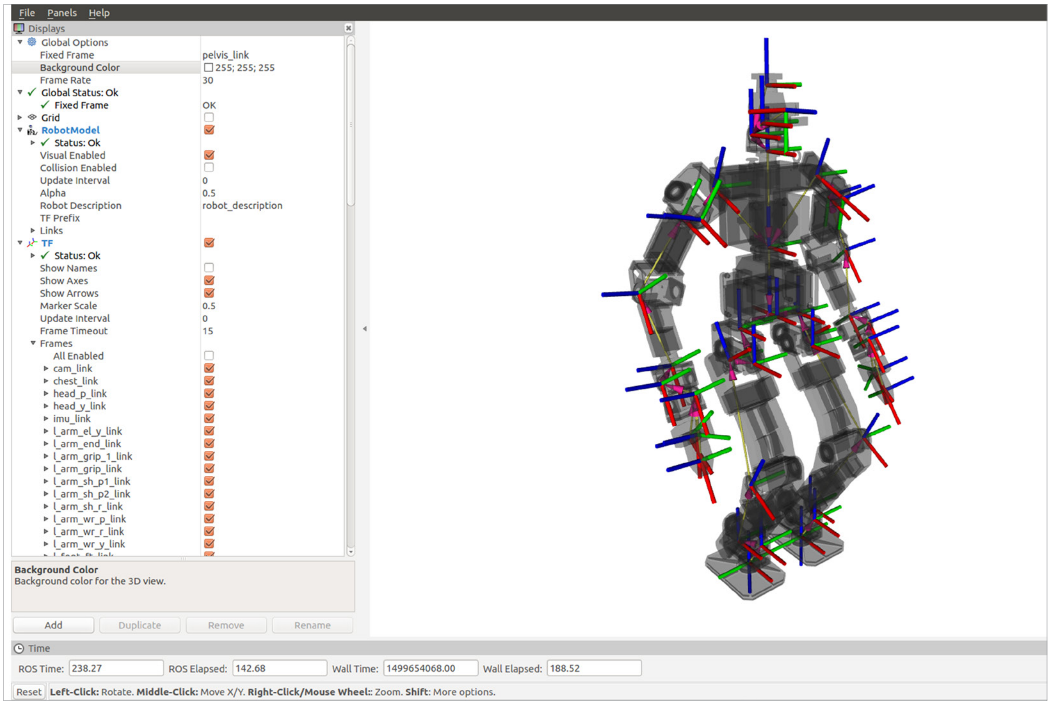
```
$ rosrun camera_package camera_node __ns:=back
$ rosrun rqt_imgae_view rqt_imgae_view __ns:=back
```

We have seen various uses of names. Names support the ability to seamlessly connect the ROS system as a whole. In this section, we learned how to change the name value using the node command 'rosrun'. Similarly, by using 'roslaunch', it is possible to execute these options at once. This will be discussed in more detail in Chapter 7 with examples.

4.5. Coordinate Transformation (TF)

When describing the robot's arm pose as shown in Figure 4-17, it can be described as the relative coordinate transform[41] of each joint. For example, the hand of a humanoid robot is connected to the wrist, the wrist is connected to the elbow, and the elbow is connected to the shoulder. In addition, the elbow can be displayed in relation to the leg joint poses while the robot is walking and moving. Finally, the coordinate of elbow correlates with the center of robot feet. Conversely, when the robot is walking, coordinates of the robot's hands move according to the relative coordinate transformation of the respective correlated joints. In addition, if the robot tries to catch an object, the origin of the robot will be relatively positioned on a specific map, and the object will also be positioned on the map. The robot can calculate the position of the object relative to its position on the map to catch the object. In robotics programming, the robot's joints (or wheels with rotating axes) and the position of each robot through coordinate transformation are very important, and in ROS, this is represented by TF (transform)[42].

41 http://wiki.ros.org/geometry/CoordinateFrameConventions
42 http://wiki.ros.org/tf



```
Displays
Global Options
  Fixed Frame            pelvis_link
  Background Color       ☐ 255; 255; 255
  Frame Rate             30
Global Status: Ok
  Fixed Frame            OK
Grid                     ☐
RobotModel               ✓
  Status: Ok
  Visual Enabled         ✓
  Collision Enabled      ☐
  Update Interval        0
  Alpha                  0.5
  Robot Description      robot_description
  TF Prefix
  Links
TF                       ✓
  Status: Ok
  Show Names             ☐
  Show Axes              ✓
  Show Arrows            ✓
  Marker Scale           0.5
  Update Interval        0
  Frame Timeout          15
  Frames
    All Enabled          ☐
    cam_link             ✓
    chest_link           ✓
    head_p_link          ✓
    head_y_link          ✓
    imu_link             ✓
    l_arm_el_y_link      ✓
    l_arm_end_link       ✓
    l_arm_grip_1_link    ✓
    l_arm_grip_link      ✓
    l_arm_sh_p1_link     ✓
    l_arm_sh_p2_link     ✓
    l_arm_sh_r_link      ✓
    l_arm_wr_p_link      ✓
    l_arm_wr_r_link      ✓
    l_arm_wr_y_link      ✓

Background Color
Background color for the 3D view.

Time
ROS Time: 238.27   ROS Elapsed: 142.68   Wall Time: 1499654068.00   Wall Elapsed: 188.52

Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
```

FIGURE 4-17 Coordinate of the Robot Parts (THORMANG3, http://robots.ros.org/thormang/)

In ROS, the coordinate transformation TF is one of the most useful concepts when describing the robot parts as well as obstacles and objects. The pose can be described as a combination of positions and orientations. Here, the position is expressed by three vectors x, y, and z, and the orientation by four vectors x, y, z, and w, called a quaternion. The quaternion is not intuitive because they do not describe the rotation of three axes (x, y, z), such as the roll, pitch, and yaw angles that are often used. However, the quaternion form is free from the gimbal lock or speed issues that present in the Euler method of roll, pitch and yaw vectors. Therefore, the quaternion type is preferred in robotics, and ROS also uses quaternion for this reason. Of course, functions to convert Euler values to quaternions are provided for convenience.

TF uses the message structure[43] shown below. The Header is used to record the converted time, and a message named 'child_frame_id' is used to specify the child coordinates. The relative position and orientation are described in the following data form: transform.translation.x / transform.translation.y / transform.translation.z / transform.rotation.x / transform.rotation.y / transform.rotation.z / transform.rotation.w

43 http://docs.ros.org/api/geometry_msgs/html/msg/TransformStamped.html

geometry_msgs/TransformStamped.msg

```
Header header
string child_frame_id
Transform transform
```

A brief description of TF was given. More detailed TF examples will be explained in the modeling part of mobile robots (Chapter 10, 11) and manipulators (Chapter 13).

4.6. Client Library

Programming languages are very diverse. C++ is often used for performance and hardware-centric control, and Python or Ruby is used for productivity. LISP is widely used in the artificial intelligence field, MATLAB for scientific software such as and numerical analysis, Java for Android, as well as many more such as C#, Go, Haskell, Node.js, Lua, R, EusLisp, Julia, etc. It cannot be said which of these languages is more important than others. The developer will select the most suitable language depends on the nature of the work. Therefore, ROS, which supports robots with various purposes, allows developer to select the most appropriate language according to the purpose. Nodes can be written in various languages, and the information is exchanged through message communication between nodes. The software module that makes it possible to write nodes in various languages is the client library. Some of the most commonly used libraries are 'roscpp' for the C++ language, 'rospy' for the Python language, 'roslisp' for LISP, and 'rosjava' for Java. In addition, there are 'roscs, roseus, rosgo, roshask, rosnodejs, RobotOS.jl, roslua, PhaROS, rosR, rosruby' and 'Unreal-Ros-Plugin'. Each client library is still under development for language diversity of ROS.

This book will focus on 'roscpp' for the C++ language. Even if different languages are used, the concepts are the same with different programming syntax. Therefore, developers can use the language most suitable for the purpose by referring to the wiki of each client library.

4.7. Communication between Heterogenous Devices

As described in the meta-operating system part of Chapter 2, ROS supports communication between different devices through the use of message communication, message, name, transform function, and client library (see Figure 4-18). ROS can be run regardless of the type of operating system it is installed upon, and regardless of the programming language used. As long as ROS is installed and each node is properly designed, communication between nodes is very easy. For example, the status of a robot can be monitored on MacOS, even if Ubuntu, a distribution of Linux, is installed on the robot. At the same time, the user can command the robot from an Android-based app. An example of this is covered in the USB camera description in Chapter 8, by taking an example of video stream transmission between two different PCs. And even for a microcontroller embedded system that cannot install ROS, message communication is possible if it is designed to be able to send and receive ROS messages. This is discussed in more detail in the Embedded Systems section of Chapter 9.

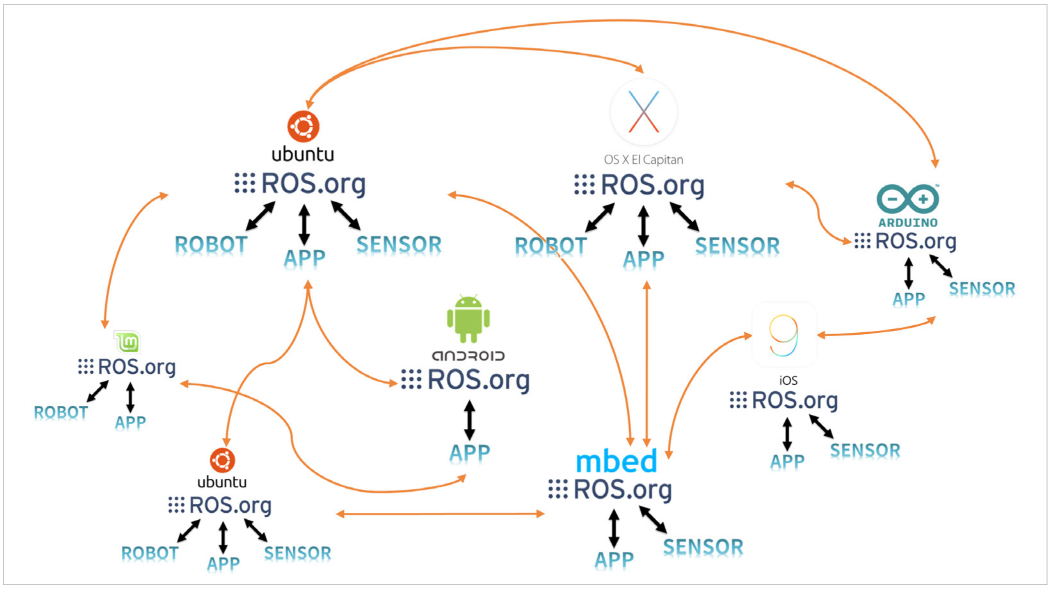


FIGURE 4-18 Communication between heterogenous Devices

4.8. File System

4.8.1. File Configuration

Let's learn about the file configuration of ROS. In ROS, the package is the basic unit for software configuration, and ROS applications are developed as a package. The package contains one or more nodes which are the smallest execution processors in ROS, or include configuration files for running other nodes. As of July 2017, ROS Indigo has around 2,500 packages and ROS Kinetic has around 1,600 official packages. There are about 5,000 packages developed and released by users as well, although there may be some redundancies. These packages are also managed as a set of packages, which is a collection of packages with a common purpose called a metapackage. For example, the Navigation metapackage consists of 10 packages including AMCL, DWA, EKF, and map_server and more. Each package contains 'package.xml', which is an XML file containing information about the package, including its name, author, license, and dependent packages. In addition, Catkin, which is the ROS build system, uses CMake and 'CMakeLists.txt' in the package folder describes the build environment. In addition, the package consists of the source code for node and message files for message communication between nodes.

ROS's file system is divided into installation folders and workspace folders. If the desktop version of ROS is installed, the installation folder is created in the /opt folder, and core utilities including roscore, rqt, RViz, robot related library, simulation and navigation are installed within the folder. The user rarely has a need to modify the files in this area. However, in order to modify the package that is officially distributed as a binary file, check the repository that contains the original source and copy the source to the workspace by using 'git clone [REPOSITORY_ADDRESS]' in '~/catkin_ws/src' rather than using the package installation command of 'sudo apt-get install ros-kinetic-xxx',.

The user's workspace can be create wherever the user wants, but let's create it in the Linux user folder of '~/catkin_ws/ ('~/' is the '/home/user/' folder in the Linux). Next, let's learn about the ROS installation folder and workspace.

Binary Installation and Source Code Installation

There are two methods to install ROS packages. The first is to install the packages provided in the binary form which can be executed immediately without a build process. The second is for the user to download the source code of the package and build it before installation. These methods are used in different purposes. If you would like to modify the package or check the contents of the source code, you can use the latter installation method. The following is an example of a TurtleBot3 package and describes the differences between the two installation methods.

1. Binary Installation

```
$ sudo apt-get install ros-kinetic-turtlebot3
```

2. Source Code Installation

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ cd ~/catkin_ws/
$ catkin_make
```

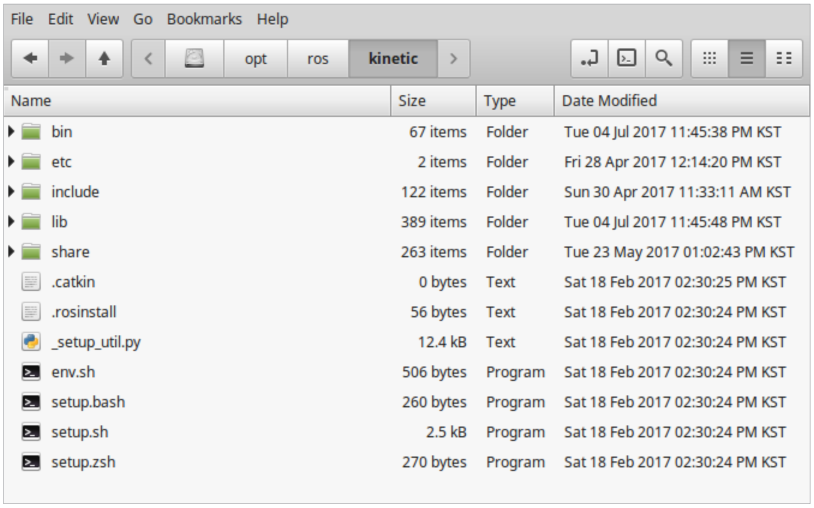
4.8.2. Installation Folder

ROS is installed in the '/opt/ros/[VERSION_NAME]' folder. For example, if you have installed the ROS Kinetic Kame version, the ROS installation path is:

- ROS Installation Folder Path '/opt/ros/kinetic'

File Configuration

When ROS is installed, '/opt/ros/kinetic' folder consists of bin, etc, include, lib, the share folder and some configuration files as shown in Figure 4-19.



Name	Size	Type	Date Modified
bin	67 items	Folder	Tue 04 Jul 2017 11:45:38 PM KST
etc	2 items	Folder	Fri 28 Apr 2017 12:14:20 PM KST
include	122 items	Folder	Sun 30 Apr 2017 11:33:11 AM KST
lib	389 items	Folder	Tue 04 Jul 2017 11:45:48 PM KST
share	263 items	Folder	Tue 23 May 2017 01:02:43 PM KST
.catkin	0 bytes	Text	Sat 18 Feb 2017 02:30:25 PM KST
.rosinstall	56 bytes	Text	Sat 18 Feb 2017 02:30:24 PM KST
_setup_util.py	12.4 kB	Text	Sat 18 Feb 2017 02:30:24 PM KST
env.sh	506 bytes	Program	Sat 18 Feb 2017 02:30:24 PM KST
setup.bash	260 bytes	Program	Sat 18 Feb 2017 02:30:24 PM KST
setup.sh	2.5 kB	Program	Sat 18 Feb 2017 02:30:24 PM KST
setup.zsh	270 bytes	Program	Sat 18 Feb 2017 02:30:24 PM KST

FIGURE 4-19 ROS File Configuration

File and folder descriptions

The ROS folder contains the packages and ROS programs selected upon the installation of ROS. The details are as follows.

/bin	Executable Binary Files
/etc	ROS and Catkin related Configuration Files
/include	Header Files
/lib	Library Files
/share	ROS Packages
env.*	Environment Configuration Files
setup.*	Environment Configuration Files

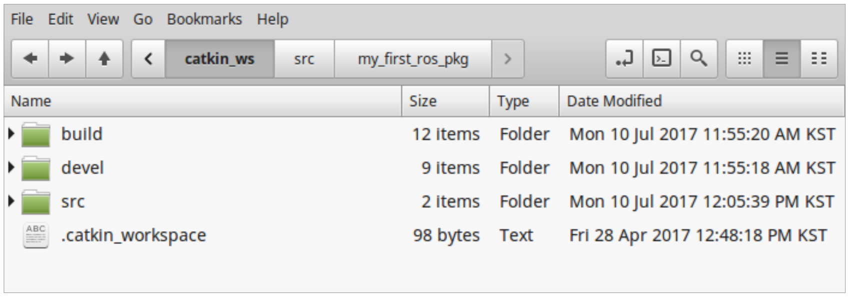
4.8.3. Workspace Folder

You can create a workspace wherever you want, but in this book, let's use '~/catkin_ws/' which is under the Linux user folder for convenience. The full path for the selected workspace folder will be '/home/username/catkin_ws'. For example, if the username is 'oroca' and the name of the catkin folder is 'catkin_ws', the path is:

- **Workspace Path**: /home/oroca/catkin_ws/

File Configuration

As shown in Figure 4-20, there is a folder called 'catkin_ws' under the '/home/username/' folder, and it consists of build, devel, and src folders. Note that the build and devel folders are created after catkin_make.



Name	Size	Type	Date Modified
build	12 items	Folder	Mon 10 Jul 2017 11:55:20 AM KST
devel	9 items	Folder	Mon 10 Jul 2017 11:55:18 AM KST
src	2 items	Folder	Mon 10 Jul 2017 12:05:39 PM KST
.catkin_workspace	98 bytes	Text	Fri 28 Apr 2017 12:48:18 PM KST

FIGURE 4-20 File Configuration of catkin workspace

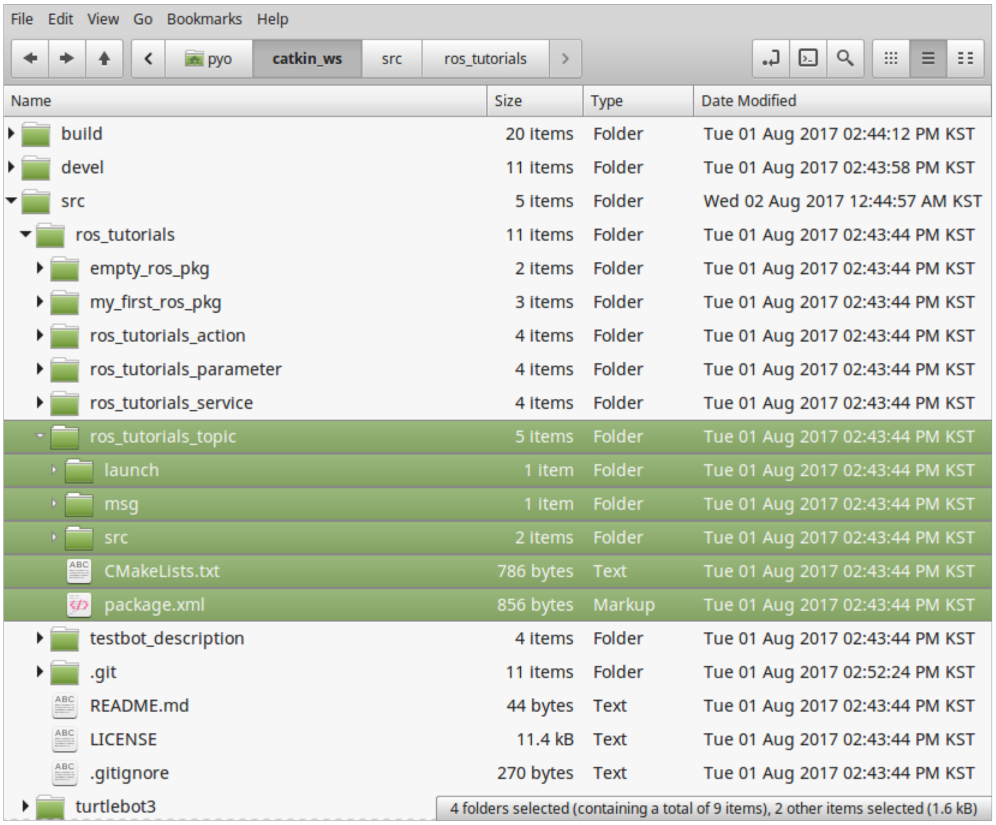
Detailed File Configuration

The workspace is a space that stores and builds user-created packages and packages published by other developers. Users perform most operations related to ROS in this folder. The details are as follows.

/build	Build Related Files
/devel	msg, srv Header Files and User Package Library, Execution Files
/src	User Packages

User Package

The '~/catkin_ws/src' folder is the space for the user source code. In this folder, you can save and build your own ROS packages or packages developed by other developers. The ROS build system will be described in detail in the next section. Figure 4-21 below shows the state after completing the 'ros_tutorials_topic' package. It describes folders and files that are commonly used, although the configuration can vary depending on the purpose of the package.



Name	Size	Type	Date Modified
build	20 items	Folder	Tue 01 Aug 2017 02:44:12 PM KST
devel	11 items	Folder	Tue 01 Aug 2017 02:43:58 PM KST
src	5 items	Folder	Wed 02 Aug 2017 12:44:57 AM KST
ros_tutorials	11 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
empty_ros_pkg	2 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
my_first_ros_pkg	3 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
ros_tutorials_action	4 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
ros_tutorials_parameter	4 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
ros_tutorials_service	4 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
ros_tutorials_topic	5 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
launch	1 item	Folder	Tue 01 Aug 2017 02:43:44 PM KST
msg	1 item	Folder	Tue 01 Aug 2017 02:43:44 PM KST
src	2 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
CMakeLists.txt	786 bytes	Text	Tue 01 Aug 2017 02:43:44 PM KST
package.xml	856 bytes	Markup	Tue 01 Aug 2017 02:43:44 PM KST
testbot_description	4 items	Folder	Tue 01 Aug 2017 02:43:44 PM KST
.git	11 items	Folder	Tue 01 Aug 2017 02:52:24 PM KST
README.md	44 bytes	Text	Tue 01 Aug 2017 02:43:44 PM KST
LICENSE	11.4 kB	Text	Tue 01 Aug 2017 02:43:44 PM KST
.gitignore	270 bytes	Text	Tue 01 Aug 2017 02:43:44 PM KST
turtlebot3			

4 folders selected (containing a total of 9 items), 2 other items selected (1.6 kB)

FIGURE 4-21 File Configuration of the User Package

/include	Header Files
/launch	Launch Files Used with roslaunch
/node	Script for rospy
/msg	Message Files
/src	Source Code Files
/srv	Service Files
CMakeLists.txt	Build Configuration File
package.xml	Package Configuration File

4.9. Build System

The ROS build system uses CMake (Cross Platform Make) by default and the build environment is described in the 'CMakeLists.txt' file in the package folder. ROS provides ROS-specific catkin build system with modified CMake.

The reason for using CMake on ROS is to allow the ROS package to be built on multiple platforms. Unlike Make, which relies only on Unix-based systems, CMake supports Windows, as well as Unix-based systems of Linux, BSD, and OS X. It also supports Microsoft Visual Studio and can be easily applied to Qt development. Furthermore, the Catkin build system makes it easy to use ROS-related builds, package management, and dependencies between packages.

4.9.1. Creating a Package

The command to create a ROS package is as follows.

```
$ catkin_create_pkg [PACKAGE_NAME] [DEPENDENT_PACKAGE_1] [DEPENDENT_PACKAGE_N]
```

'catkin_create_pkg' command creates a package folder that contains the 'CMakeLists.txt' and 'package.xml' files necessary for the Cake build system. Let's create a simple package to help you understand. First, open a new terminal window (Ctrl + Alt + t) and run the following command to move to the workspace folder.

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

The package name to be created is 'my_first_ros_pkg'. Package names in ROS should all be lowercase and must not contain spaces. The naming guideline also uses an underscore(_) between each word instead of a dash(-) or a space. See the relevant pages for coding style guide[44] [45] and naming conventions in ROS. Now, let's create a package named 'my_first_ros_pkg' with the following command:

```
$ catkin_create_pkg my_first_ros_pkg std_msgs roscpp
```

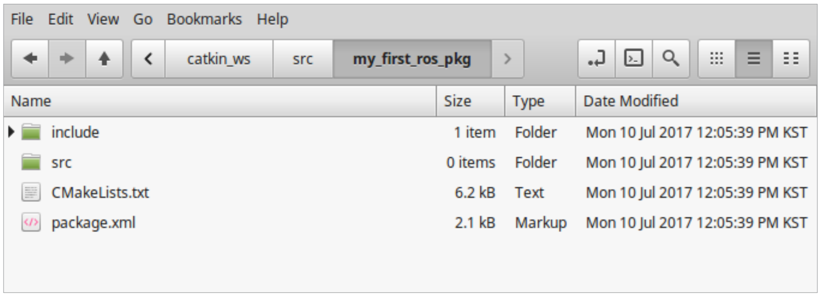
'std_msgs' and 'roscpp' were added as optional dependent packages in the previous command. This means that the 'std_msgs', which is a standard message package of ROS, and the 'roscpp', which is a client library necessary to use C/C++ in ROS, must be installed prior to the

44 http://wiki.ros.org/CppStyleGuide
45 http://wiki.ros.org/PyStyleGuide

creation of the package. These dependent package settings can be specified when creating the package, but can also be created directly in 'package.xml'.

Once the package is created, 'my_first_ros_pkg' package folder will be created in the '~/catkin_ws/src' folder, along with the default internal folder that the ROS package should have, and the 'CMakeLists.txt' and 'package.xml' files. The contents can be checked with the 'ls' command as below, and the inside of the package can be checked using the GUI-based tool Nautilus which acts like Window Explorer.

```
$ cd my_first_ros_pkg
$ ls
include                 → Include Folder
src                     → Source Code Folder
CMakeLists.txt          → Build Configuration File
package.xml             → Package Configuration File
```



Name	Size	Type	Date Modified
include	1 item	Folder	Mon 10 Jul 2017 12:05:39 PM KST
src	0 items	Folder	Mon 10 Jul 2017 12:05:39 PM KST
CMakeLists.txt	6.2 kB	Text	Mon 10 Jul 2017 12:05:39 PM KST
package.xml	2.1 kB	Markup	Mon 10 Jul 2017 12:05:39 PM KST

FIGURE 4-22 Automatically Created Files and Folders when Creating a New Package

4.9.2. Modifying the Package Configuration File (package.xml)

'Package.xml', which is one of the essential ROS configuration files, is an XML file containing information about the package, including the package name, author, license, and dependent packages. The original file without any modifications is shown below.

package.xml

```
<?xml version="1.0"?>
<package>
  <name>my_first_ros_pkg</name>
  <version>0.0.0</version>
  <description>The my_first_ros_pkg package</description>
```

```
<!-- One maintainer tag required, multiple allowed, one person per tag -->
<!-- Example:  -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
<maintainer email="oroca@todo.todo">pyo</maintainer>

<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
<license>TODO</license>

<!-- Url tags are optional, but mutiple are allowed, one per tag -->
<!-- Optional attribute type can be: website, bugtracker, or repository -->
<!-- Example: -->
<!-- <url type="website">http://wiki.ros.org/my_first_ros_pkg</url> -->

<!-- Author tags are optional, mutiple are allowed, one per tag -->
<!-- Authors do not have to be maintianers, but could be -->
<!-- Example: -->
<!-- <author email="jane.doe@example.com">Jane Doe</author> -->

<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!--   <run_depend>message_runtime</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>

<!-- The export tag contains other, unspecified, tags -->
<export>
```

```
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

Below are descriptions of each statement.

<?xml>	This tag indicates that the contents in the document abide by the XML Version 1.0.
<package>	This tag is paired with </package> tag to indicate the configuration part of the ROS package configuration part.
<name>	This tag indicates the package name. The package name entered when creating the package is used. The name of the package can be changed by the developer.
<version>	This tag indicates the package version. The developer can assign the version of the package.
<description>	A short description of the package. Usually 2-3 sentences.
<maintainer>	The name and e-mail address of the package administrator.
<license>	This tag indicates the license, such as BSD, MIT, Apache, GPLv3, LGPLv3.
<url>	This tag indicates address of the webpage describing the package, or bug management, repository, etc. Depending on the type, you can assign it as a website, bugtracker, or repository.
<author>	The name and email address of the developer who participated in the package development. If multiple developers were involved, append multiple <author> tags to the following lines.
<buildtool_depend>	Describes the dependencies of the build system. As we are using the Catkin build system, write 'catkin'.
<build_depend>	Dependent package name when building the package.
<run_depend>	Dependent package name when running the package.
<test_depend>	Dependent package name when testing the package.
<export>	It is used when using a tag name that is not specified in ROS. The most widely used case is for metapackages. In this case, use <export><metapackage/></export> to notify that the package is a metapackage.
<metapackage>	The official tag used within the export tag that declares the current package as a metapackage.

I modified the package configuration file (package.xml) as follows. Let's modify it in your own environment as well. If you are unfamiliar with it, you can use the below file as is:

package.xml

```xml
<?xml version="1.0"?>
<package>
  <name>my_first_ros_pkg</name>
  <version>0.0.1</version>
  <description>The my_first_ros_pkg package</description>
  <license>Apache License 2.0</license>
  <author email="pyo@robotis.com">Yoonseok Pyo</author>
  <maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
  <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
  <url type="website">http://www.robotis.com</url>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>roscpp</run_depend>
  <export></export>
</package>
```

4.9.3. Modifying the Build Configuration File (CMakeLists.txt)

Catkin, the build system for ROS, uses CMake and describes the build environment in the 'CMakeLists.txt' in the package folder. It configures the executable file creation, dependency package priority build, link creation, and so on. The original file without any modifications is shown below.

CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 2.8.3)
project(my_first_ros_pkg)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)

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


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
# catkin_python_setup()

################################################
## Declare ROS messages, services and actions ##
################################################

## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a run_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
##     * add a run_depend tag for "message_runtime"
## * In this file (CMakeLists.txt):
##   * add "message_generation" and every package in MSG_DEP_SET to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

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

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
##   * add a build_depend and a run_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
##   * add "dynamic_reconfigure" to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * uncomment the "generate_dynamic_reconfigure_options" section below
##     and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
```

```
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
# )

###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES my_first_ros_pkg
#  CATKIN_DEPENDS roscpp std_msgs
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
# add_library(my_first_ros_pkg
#   src/${PROJECT_NAME}/my_first_ros_pkg.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
## either from message generation or dynamic reconfigure
# add_dependencies(my_first_ros_pkg ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

## Declare a C++ executable
# add_executable(my_first_ros_pkg_node src/my_first_ros_pkg_node.cpp)

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(my_first_ros_pkg_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
# target_link_libraries(my_first_ros_pkg_node
#   ${catkin_LIBRARIES}
# )

#############
## Install ##
#############

# all install targets should use catkin DESTINATION variables
# See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html

## Mark executable scripts (Python etc.) for installation
## in contrast to setup.py, you can choose the destination
# install(PROGRAMS
#   scripts/my_python_script
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark executables and/or libraries for installation
# install(TARGETS my_first_ros_pkg my_first_ros_pkg_node
#   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark cpp header files for installation
```

```
# install(DIRECTORY include/${PROJECT_NAME}/
#   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
#   FILES_MATCHING PATTERN "*.h"
#   PATTERN ".svn" EXCLUDE
# )

## Mark other files for installation (e.g. launch and bag files, etc.)
# install(FILES
#   # myfile1
#   # myfile2
#   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# )

#############
## Testing ##
#############

## Add gtest based cpp test target and link libraries
# catkin_add_gtest(${PROJECT_NAME}-test test/test_my_first_ros_pkg.cpp)
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
```

Options in the build configuration file (CMakeLists.txt) are as follows. The below describes the minimum required version of ‘cmake’ installed on the operating system. Since it is currently specified as version 2.8.3, if you use a lower version of Cmake than this, you need to update the ‘cmake’ to meet the minimum requirement.

```
cmake_minimum_required(VERSION 2.8.3)
```

The project describes the name of the package. Use the package name entered in ‘package.xml’. Note that if the package name is different from the package name described in the <name> tag in ‘package.xml’, an error will occur when building the package.

```
project(my_first_ros_pkg)
```

The ‘find_package’ entry is the component package required to perform a build on Catkin. In this example, ‘roscpp’ and ‘std_msgs’ are set as dependent packages. If the package entered here is not found in the system, an error will occur when building the package. In other words, this is an option to require the installation of dependent packages for the custom package.

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)
```

The following is a method used when using packages other than ROS. For example, when using Boost, the ‘system’ package must be installed beforehand. This feature is an option that allows you to install dependent packages.

```
find_package(Boost REQUIRED COMPONENTS system)
```

The ‘catkin_python_setup()’ is an option when using Python with ‘rospy’. It invokes the Python installation process ‘setup.py’.

```
catkin_python_setup()
```

‘add_message_files’ is an option to add a message file. The ‘FILES’ option will automatically generate a header file (*.h) by referring to the ‘.msg’ files in the ‘msg’ folder of the current package. In this example, message files Message1.msg and Message2.msg are used.

```
add_message_files(
  FILES
  Message1.msg
  Message2.msg
)
```

‘add_service_files’ is an option to add a service file to use. The ‘FILES’ option will refer to ‘.srv’ files in the ‘srv’ folder in the package. In this example, you have the option to use the service files Service1.srv and Service2.srv.

```
add_service_files(
  FILES
  Service1.srv
  Service2.srv
)
```

'generate_messages' is an option to set dependent messages. This example sets the DEPENDENCIES option to use the 'std_msgs' message package.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

'generate_dynamic_reconfigure_options' loads configuration files that are referred when using 'dynamic_reconfigure'.

```
generate_dynamic_reconfigure_options(
  cfg/DynReconf1.cfg
  cfg/DynReconf2.cfg
)
```

The following are the options when performing a build on Catkin. 'INCLUDE_DIRS' is a setting that specifies to use the header file in the 'include' folder, which is the internal folder of the package. 'LIBRARIES' is a setting used to specify the package library in the following configuration. 'CATKIN_DEPENDS' specifies dependent packages and in this example, the dependent packages are set to 'roscpp' and 'std_msgs'. 'DEPENDS' is a setting that describes system-dependent packages.

```
catkin_package(
 INCLUDE_DIRS include
 LIBRARIES my_first_ros_pkg
 CATKIN_DEPENDS roscpp std_msgs
 DEPENDS system_lib
)
```

'include_directories' is an option to specify folders to include. In the example, '${catkin_INCLUDE_DIRS}' is configured, which refers to the header file the 'include' folder in the package. To specify an additional include folder, append it to the next line of '${catkin_INCLUDE_DIRS}'.

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

'add_library' declares the library to be created after the build. The following option will create 'my_first_ros_pkg' library from 'my_first_ros_pkg.cpp' file in the 'src' folder.

```
add_library(my_first_ros_pkg
  src/${PROJECT_NAME}/my_first_ros_pkg.cpp
)
```

'add_dependencies' is a command to perform certain tasks prior to the build process such as creating dependent messages or dynamic reconfigurations. The following options describe the creation of dependent messages and dynamic reconfiguration, which are the dependencies of the 'my_first_ros_pkg' library.

```
add_dependencies(my_first_ros_pkg ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

'add_executable' specifies the executable to be created after the build. The option specifies the system to refer to the 'src/my_first_ros_pkg_node.cpp' file to generate the 'my_first_ros_pkg_node' executable file. If there are multiple '*.cpp' files to be referenced, append them after 'my_first_ros_pkg_node.cpp'. If there are two or more executable files to be created, add an additional 'add_executable' entry.

```
add_executable(my_first_ros_pkg_node src/my_first_ros_pkg_node.cpp)
```

'add_dependencies' option is like the 'add_dependencies' previously described, which is required to perform certain tasks such as creating dependent messages or dynamic reconfigurations prior to building libraries or executable files. The following describes the dependency of the executable file named 'my_first_ros_pkg_node', not the library mentioned above. It is most often used when creating message files prior to building executable files.

```
add_dependencies(my_first_ros_pkg_node ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})
```

'target_link_libraries' is an option that links libraries and executables that need to be linked before creating an executable file.

```
target_link_libraries(my_first_ros_pkg_node
 ${catkin_LIBRARIES}
)
```

In addition, the Install option used when creating the official distribution ROS package and the testing option used for the package test is provided.

The following is the modified build configuration file (CMakeLists.txt). Modify the file for your package. For more information on how to use the configuration file, please refer to the packages of TurtleBot3 and ROBOTIS OP3 published at 'https://github.com/ROBOTIS-GIT'.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(my_first_ros_pkg)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
catkin_package(CATKIN_DEPENDS roscpp std_msgs)
include_directories(${catkin_INCLUDE_DIRS})
add_executable(hello_world_node src/hello_world_node.cpp)
target_link_libraries(hello_world_node ${catkin_LIBRARIES})
```

4.9.4. Writing Source Code

The following setting is configured in the executable file creation section (add_executable) of the 'CMakeLists.txt' file mentioned above.

```
add_executable(hello_world_node src/hello_world_node.cpp)
```

This is the setting to create the executable 'hello_world_node' by referring to the 'hello_world_node' source code in the 'src' folder of the package. As 'hello_world_node.cpp' source code has to be manually created and written by developer, let's write a simple example.

First, move to the source code folder (src) in your package folder by using 'cd' command and create the 'hello_world_node.cpp' file as shown below. This example uses the gedit editor, but you can use your preferred editor, such as vi, gedit, qtcreator, vim, or emacs.

```
$ cd ~/catkin_ws/src/my_first_ros_pkg/src/
$ gedit hello_world_node.cpp
```

Then, write the following source code in the created file.

hello_world_node.cpp

```
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <sstream>

int main(int argc, char **argv)
```

```
{
  ros::init(argc, argv, "hello_world_node");
  ros::NodeHandle nh;
  ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("say_hello_world", 1000);
  ros::Rate loop_rate(10);
  int count = 0;

  while (ros::ok())
  {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world!" << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```

4.9.5. Building the Package

Once above code is saved in the file, all the necessary work for building a package is completed. Before building the package, update the profile of the ROS package with the below command. It is a command to apply the previously created package to the ROS package list. Although this is not mandatory, it is convenient to update after creating a new package as it will allows to find the package using auto-completion feature with the Tab key.

```
$ rospack profile
```

The following is a Catkin build. Go to the Catkin workspace and build the package.

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

Alias Command

As mentioned in Section 3.2 Setting Up the ROS Development Environment, if you set alias cm='cd ~/catkin_ws && catkin_make' in the '.bashrc' file, you can replace the above command with 'cm' command in the terminal window. As it is very useful, make sure to set it by referring to the ROS Development Environment setup section.

4.9.6. Running the Node

If the build was completed without any errors, a 'hello_world_node' file should have been created in '~/catkin_ws/devel/lib/my_first_ros_pkg'.

The next step is to run the node. Open a terminal window (Ctrl + Alt + t) and run roscore before running the node. Note that roscore must be running in order to execute ROS nodes, and roscore only needs to be run once unless it stops.

```
$ roscore
```

Finally, open a new terminal window (Ctrl + Alt + t) and run the node with the command below. This is a command to run a node called 'hello_world_node' in a package named 'my_first_ros_pkg'.

```
$ rosrun my_first_ros_pkg hello_world_node
[INFO] [1499662568.416826810]: hello world!0
[INFO] [1499662568.516845339]: hello world!1
[INFO] [1499662568.616839553]: hello world!2
[INFO] [1499662568.716806374]: hello world!3
[INFO] [1499662568.816807707]: hello world!4
[INFO] [1499662568.916833281]: hello world!5
[INFO] [1499662569.016831357]: hello world!6
[INFO] [1499662569.116832712]: hello world!7
[INFO] [1499662569.216827362]: hello world!8
[INFO] [1499662569.316806268]: hello world!9
[INFO] [1499662569.416805945]: hello world!10
```

When the node is running, messages such as 'hello world!0,1,2,3 ...' can been viewed in the terminal window in strings. This is not an actual message transfer in ROS, but it can be seen as a result of the build system example discussed in this section. Since this section is intended to describe the build system of ROS, the source code for messages and nodes will be discussed in more detail in the following chapters.