# 00 ROS Quick Review Notes

September 22, 2021

**Information:** Some simple notes about using ROS for a quick reference.

**Written by:** Zihao Xu

**Last update date:** September.22.2021

## 1 ROS Workspace

### 1.1 Catkin Workspaces

- A catkin workspace is a folder where you modify, build and install catkin packages.
- Usually a catkin workspace (uniformly named *catkin_ws*) consists of three spaces: *build*, *devel*, *src*.
  - *src*: The source space contains the source code of catkin packages. This is where the source code for the packages can be edited.
  - *build*: The build space is where CMake is invoked to build the catkin packages in the source space. Some cache information and other immediate files of CMake and catkin are kept here.
  - *devel*: The development space is where built targets are placed prior to being installed, which provides a useful testing and development environment which does not require invoking the installation step.
  - *install*: THe install space is where the built targets are installed and is usually not used in development.
- Detailed information can be found here.

### 1.2 Commonly used commands

*source /opt/ros/<distro>/setup.zsh*

- Set up the environment variables for ROS.
- Necessary on every new shell.

*printenv | prep ROS*

- Check the currently activated ROS environments.
- Especially useful when multiple versions of ROS are installed.

*catkin_make*

- A convenience tool for building code in a catkin workspaces.

- Need to be called in the root of the desired catkin workspace.
- Running it the first time in the workspace, it will create a *CMakeLists.txt* in the *src* folder. Previously this is a separate step completed by `catkin_init_workspace`.

`source devel/setup.bash`

- Overlay the workspace on top of the environment.
- Can be checked by viewing the *ROS_PACKAGE_PATH* environment: `echo $ROS_PACKAGE_PATH`.

# 2 ROS Filesystem

## 2.1 Packages

- *Packages* are the **software organization unit** of ROS code.
- Each *package* can contain libraries, executables, scripts, or other artifacts.

## 2.2 Manifests

- A *manifest* is a **description** of a package.
- It serves to define dependencies between *packages* and to capture meta information about the *package* like version, maintainer, licenses, etc…

## 2.3 Commonly used commands

- *`rospack find [package_name]`* returns the path to package. Other useful commands for *rospack* can be viewed here.
- *`rospack depends1 [package_name]`* finds the **first-order** dependencies of a given package.
- *`rospack depends [package_name]`* finds all the nested dependencies of a given package.
- *`roscd <package>[/subdir]`* changes directory directly to a package or a subdirectory of a package.
- *`roscd log`* changes directory to the folder where ROS stores log files.
- *`rosls <package>[/subdir]`* will *ls* directly in a package or a subdirectory of a package.

# 3 ROS Graph Concepts

## 3.1 Nodes

- A node is an executable file within a ROS package.
- A node uses ROS to communicate with other nodes.
- ROS client libraries allow nodes written in different programming languages to communicate:
  - rospy = python client library
  - roscpp = python c++ client library

## 3.2 Messages

- ROS data type used when subscribing or publishing to a topic.

## 3.3 Topics

- Nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages.

## 3.4 Services

- Another way that nodes can communicate with each other.
- Service allow nodes to send a *request* and receive a *response*.

## 3.5 Master

- Name service for ROS which helps nodes find each other.

## 3.6 rosout

- ROS equivalent of stdout/stderr.

## 3.7 Parameter server

- Nodes use parameter to store and retrieve parameters at runtime.
- Best used for static, non-binary data such as configuration parameters.

## 3.8 roscore

- Master + rosout + parameter server.

# 4 ROS Graph Commands

For all ros commands, a *-h* argument can always be used to see the help documentation about advanced usages.

## 4.1 roscore

- *roscore* is the **first** thing one should run when using ROS.
- Only one roscore can be running.

## 4.2 rosnode

- *rosnode list* lists the ROS nodes that are currently running.
- *rosnode info [/node_name]* returns information (publications, subscriptions, services, etc…) about a specific node.

## 4.3 rosrun

- *rosrun [package_name] [node_name]* run a node from a given package.

## 4.4 rostopic

- *rostopic list -v* list full details about each topic.
- *rostopic echo [topic]* shows the data published on a given topic.
- *rostopic type [topic]* returns the message type of a given topic.
- *rostopic pub [topic] [msg_type] [args]* publishes data on to a topic currently advertised.
- *rostopic hz [topic]* reports the rate at which data is published.

## 4.5 rosmsg

- *rosmsg show [msg_type]* shows the details of a given message.

## 4.6 rossrv

- *rossrc show [service type]* shows the details of a given service.

## 4.7 rosservice

- *rosservice list* prints information about active services.
- *rosservice type [service]* prints the type of a given service.
- *rosservice call [service] [args]* calls the service with the provided args.

### 4.8 rosparam

- *rosparam list* list the parameters on the Parameter Server.
- *rosparam set [param_name] [args]* set the given parameter with provided args.
- *rosparam get [param_name]* get the values of a given parameter.
- *rosparam get /* will show the contents of the entire Parameter Server.
- *rosparam dump [file_name] [namespace]* writes all the parameters to the given file.
- *rosparam load [file_name] [namespace]* load the parameters to a given namespace from the provided file.

### 4.9 rqt

- *rqt* will open rqt's the main window.
- *rosrun rqt_graph rqt_graph* creates a dynamic graph of what's going on in the system.
- *rosrun rqt_plot rqt_plot* displays a scrolling time plot of the data published on topics.

# 5 Customize a ROS package

## 5.1 Basic Structure

- The package must contain a catkin compliant **package.xml** file, which provides meta information about the package.
- The package must contain a **CMakeLists.txt** which uses catkin.
- Each package must have its own folder.
- Usually use a catkin workspace to work with catkin packages.

## 5.2 Creating a catkin Package

- In the source space of a catkin workspace, use the command *catkin_create_pkg <package_name> [depend1] [depend2] [depend3] ...* to initialize a catkin package. Advanced functionalities of *catkin_create_pkg* can be found here.

## 5.3 Customize package.xml

- **Format**
  - *<package format="2">.....</package>*
- **Name**
  - *<name>package name</name>*
- **Version**
  - *<version>package version</version>*
- **Description**
  - *<description>Write the descriptions of the package here</description>*
- **Maintainer**
  - *<maintainer email="example@example.com">name</maintainer>*
  - Multiple maintainers are allowed.
- **License**
  - *<license>license type<license>*
- **Build Tool Dependencies**
  - *<buildtool_depend>catkin</buildtool_depend>*
  - Specify build system tools which this package needs to build itself. Typically only *catkin* is needed.
- **Build Dependencies:**
  - *<build_depend>package</build_depend>*
  - Specify the packages needed at compile time, such as including headers from the package, linking against libraries from the package or requiring any other resource at build time.
- **Build Export Dependencies**
  - *<build_export_depend>package</build_export_depend>*
  - Specify the packages needed to build libraries against this package. Usually used when transitively including the headers in public headers in this package.
- **Execution Dependencies**
  - *<exec_depend>package</exec_depend>*
  - Specify which packages are needed to run code in this package. Usually used when depending on shared libraries in this package.

- **Test Dependencies**
  - *<test_depend>package</test_depend>*
  - Specify only additional dependencies for unit tests.
- **Documentation Toll Dependencies**
  - *<doc_depend>package</doc_depend>*
  - Specify documentation tools which this package needs to generate documentation.
- **Dependencies**
  - *<depend>package</depend>*
  - Specify that a dependency is a build, export, and execution dependency.
  - The most commonly used dependency tag.
- **Reference**
  - For some other advanced tags can be used in **package.xml**, refer to this website

### 5.4 Customize CMakeLists.txt

- **Required CMake Version**
  - *cmake_minumum_required(VERSION 2.8.3)*
  - Catkin requires version 2.8.3 or higher.
- **Package Name**
  - *project(proj_name)*
  - Use the variable *${PROJECT_NAME}* to reference this project name in the CMake script
- **Finding Dependent CMake Packages**
  - *find_package(catkin REQUIRED COMPONENTS [depend1] [depend2] ...)*
  - If a package is found by CMake through find_package, it results in the creation of several CMake environment variables that give information about the found package.
  - For catkin packages, if one *find_package* them as components of catkin, this is advantageous as a single set of environment variables is created with the *catkin_* prefix.
- **Enable Python module support**
  - *catkin_python_setup()*
  - Required when the package provides some Python modules. In this case, a **setup.py** is also required.
- **Message/Service/Action Generators**
  - Messages (.msg), services (.srv), and actions (.action) files in ROS require a special preprocessor build step before being built and used by ROS packages.
  - *add_message_files(FILES [msg1] [msg2] ...)*
  - *add_service_files(FILES [srv1] [srv2] ...)*
  - *add_action_files(FILES [action1] [action2] ...)*
- **Invoke message/service/action generation**
  - *generate_messages(...)*
  - Required if any message/service/action files are to be built and used by ROS packages.
- **Specify package build info export**
  - *catkin_package(...)*
  - Argument *INCLUDE_DIRS*: The exported include paths for the package.
  - Argument *LIBRARIES*: The exported libraries from the project.
  - Argument *CATKIN_DEPENDS*: Other catkin projects that this project depends on.
  - Argument *DEPENDS*: Non-catkin CMake projects that this project depends on.
  - Argument *CFG_EXTRAS*: Additional configuration options.
- **Reference**

– Here are only some basic notes for reading a simple **CMakeLists.txt**.
– For detailed information and guides, refer to this website.

# 6 Customize a ROS msg and srv

## 6.1 msg file

- msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages.
- msg files are stored in the *msg* directory of a package.
- msgs are text files with a field type and field name per line. Available filed types are:
  - Header
  - int8, int16, int32, int64
  - float32, float64
  - string
  - time, duration
  - other msg files
  - variable-length array[] and fixed-length array[C]
- Header is a special type in ROS, which contains a timestamp and coordinate frame information that are commonly used in ROS.

## 6.2 srv file

- An srv file describes a service. It is composed of two parts: a request and a response.
- srv files are stored in the *srv* directory of a package.
- srv files are similar to msg files, except they contain two parts: a request and a response. The two parts are separated by a '—' line.
  - The request part is above the '—' line.
  - The response part is below the '—' line.

## 6.3 Create a msg

### 6.3.1 Write the definition

- The first step is to create a file in the form 'Msg_Name.msg' in the *msg* directory and write the field types and field names.

### 6.3.2 Edit the package.xml to set up dependencies

- To make sure the msg files are turned into source code for C++, Python, and other languages, edit **package.xml** to add a build dependency on **message_generation** and a execution dependency on **message_runtime**.

### 6.3.3 Edit CMakeLists.txt correspondingly

- Similarly, add the **message_generation** dependency to the *find_package()* call in the **CMakeLists.txt** to generate messages. Usually, simply adding **message_generation** to the list of *COMPONENTS* works.
- Also make sure to export the **message_runtime** dependency by adding it to the *catkin_package()* call in the argument *CATKIN_DEPENDS*
- Uncomment the block of codes *add_message_files(FILES [msg1] [msg2] ...)* and fill in the custom msg filenames so that CMake knows when it has to reconfigure reconfigure the project after you add other .msg files.

- Uncomment the block of codes *generate_messages(DEPENDENCIES [depend_msg1] [depend_msg2] ...)* and fill in any msgs the custom msg file depends on.

#### 6.3.4 Make the package again

- Change directory to the source space and execute *catkin_make* command. Any .msg file in the *msg* directory will generate code for use in all supported languages.

### 6.4 Create a srv

#### 6.4.1 Write the definition

- The first step is to create a file in the form 'Service_Name.srv' in the *srv* directory and write the field types and field names.
- Remember a srv file has a request part and a response part divided by a '—' line.
- One cannot embed another .srv inside of a .srv like writing msg files.

#### 6.4.2 Edit the package.xml to set up dependencies

- To make sure the msg files are turned into source code for C++, Python, and other languages, edit **package.xml** to add a build dependency on **message_generation** and a execution dependency on **message_runtime**.
- This step is exactly the same as what is needed for creating a msg file and do not need to be repeated if has been done.

#### 6.4.3 Edit CMakeLists.txt correspondingly

- Similarly, add the **message_generation** dependency to the *find_package()* call in the **CMakeLists.txt** to generate messages. Usually, simply adding **message_generation** to the list of *COMPONENTS* works. Despite its name, **message_generation** works for both msg and srv.
- Also make sure to export the **message_runtime** dependency by adding it to the *catkin_package()* call in the argument *CATKIN_DEPENDS*
- Uncomment the block of codes *add_service_files(FILES [srv1] [srv2] ...)* and fill in the custom srv filenames.
- Uncomment the block of codes *generate_messages(DEPENDENCIES [depend_msg1] [depend_msg2] ...)* and fill in any msgs the custom srv file depends on.

#### 6.4.4 Make the package again

- Change directory to the source space and execute *catkin_make* command. Any .srv file in the *srv* directory will generate code for use in all supported languages.

# 7 rospy

## 7.1 Initialization

### 7.1.1 Python Script Declaration

```
[ ]: #!/usr/bin/env python
```

- This first line makes sure the script is executed as a Python script.

### 7.1.2 Initialize a ROS Node

```
[ ]: rospy.init_node('node_name')
```

- When no arguments are provided, the node name must be **unique**.
- Argument *anonymous = True* can be used if one does not care about the unique name of one node.
- Argument *log_level=rospy.INFO* can be used to edit the log level for publishing log messages to *rosout*.
- Argument *disable_signals = True* can be used if one does not want the node end on *Ctrl-C*.

### 7.1.3 Accessing command-line arguments

```
[ ]: rospy.myargv(argv=sys.argv)
```

- Returns a copy of *sys.argv* with remapping arguments removed.

### 7.1.4 Shutting down

- The most common usage is:

```
[ ]: while not rospy.is_shutdown():
         do some work
```

- Another way is to use *rospy.spin()* make the node sleep until the *is_shutdown()* flag is *True*.

```
[ ]: ... setup callbacks
rospy.spin()
```

### 7.1.5 ROS Rate

- rospy.Rate() is a class which helps conveniently maintaining a particular rate for a loop.

```
[ ]: rate = rospy.Rate(10) #10Hz
while not rospy.is_shutdown():
    do some work
    rate.sleep()
```

## 7.2 Message

### 7.2.1 Message generation

- The .msg files are coded into python classes and need to be imported.

```
[ ]: # There are two ways of importing
     import std_msgs.msg
     from std_msgs.msg import String
```

### 7.2.2 Message initialization

- **No arguments**

```
[ ]: msg = std_msgs.msg.String()
     msg.data = "hello world"
```

- **In-order arguments(*args)**
  - In this case, a value for all of the fields must be provided, in order.

```
[ ]: msg = std_msgs.msg.ColorRGBA(255.0, 255.0, 255.0, 128.0)
```

- **Keyword arguments(**kwds)**
  - In this case, only the fields that values of which are provided will be initialized while the rest receive default values.

```
[ ]: msg = std_msgs.msg.ColorRGBA(b=255.0)
```

## 7.3 Publishers and Subscribers

### 7.3.1 Publisher Initialization

- The only requirements to create a rospy.Publisher are the topic name, the Message class and the queue size.

```
[ ]: pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
```

### 7.3.2 Use a Publisher to publish

- **Explicit Style**

```
[ ]: pub.publish(std_msgs.msg.String("hello world"))
```

- **Implicit style with in-order arguments**
  - In this case, a value for all of the fields must be provided, in order.

```
[ ]: pub.publish(255.0, 255.0, 255.0, 128.0)
```

- **Implicit style with keyword arguments**
  - In this case, only the fields that values of which are provided will be initialized while the rest receive default values.

[ ]:
```
pub.publish(b=255)
```

#### 7.3.3 queue_size behavior and queuing

- *publish()* in rospy is **synchronous** by default
- To use **asynchronous** publishing behavior, the keyword argument *queue_size* must be passed to *subscribe* which defines the maximum queue size before messages are being dropped.
- This asynchronous behavior ensures that only the subscribers having connectivity problems will not receive new messages.

#### 7.3.4 Selecting a good queue_size

- It is recommended to pick a value which is bigger than it needs to be rather than a too small value.
- Selecting the queue size should take the loop rate into consideration.
- Setting the *queue_size* to be 1 is a good choice when only the latest published information is wanted.
- Setting the *queue_size* to be 10 or greater is a good choice when any changes in value are wanted and need to be recorded.

#### 7.3.5 Use a subscriber

- Use subscriber to subscribe to a given topic with desired message type. The received messages would be the first argument of the callback function.

[ ]:
```
def callback(data):
    do some work to received data

rospy.Subscriber("topic_name", msg_type, callback)
```

### 7.4 Service and Client

For detailed information, refer to the ROS wiki.

#### 7.4.1 Service definitions, request messages, and response messages

- rospy converts .srv files into Python source code and creates three classes. If the .srv file is *package/srv/Service1.srv*, then the classes would be:
  - Service Definitions *package.srv.Service1*
  - Request Messages *package.srv.Service1Request*
  - Response Messages *package.srv.Service1Response*
- Service Definition
  - A container for the request and response type.
  - Must be used whenever one create or call a service.
- Service Request Messages
  - The request message is used to call the appropriate service.
  - Typically one does not need to use the request message directly.
- Service Response Messages

  - The response message is used to contain the return value from the appropriate service.
  - Service handlers must return response messages instances of the correct type.

### 7.4.2 Client Node: Calling services

**Service proxies**

- In ROS, call a service by creating a *rospy.ServiceProxy* instance with the name of the service to be called.
  - The instance is callable and can be invoked like methods.
- Before creating a proxy, one often will want to block until the service is available.

**Exceptions**

- If a service returns an error for the request, a *rospy.ServiceException* will be raised. The exception would contain any error messages that the service sent.
  - *TypeError*: Request is not of the valid type.
  - *ServiceException*: Communication with remote service failed.
  - *ROSSerializationException*: This usually indicates a type error with one of the fields.

[ ]:
```
rospy.wait_for_service("service_name", timeout=None)
proxy = rospy.ServiceProxy("service_name", service_class, persistent=False,␣
↪headers=None)
try:
    response = proxy(request_message_data)
except rospy.ServiceException as exc:
    print("Service did not process request: " + str(exc))
```

**Explicit style**

- Create the **Request* instance and pass it to proxy

[ ]:
```
request = package.srv.Service1Request(request_message_data)
response = proxy(request)
```

**Implicit style with in-order arguments** - In the in-order style, a new Message instance will be created with the arguments provided, in order. - In this case, a value for all of the fields must be provided, in order.

[ ]:
```
response = proxy(request_message_data)
```

**Implicit style with keyword arguments**

- In this case, only the fields that values of which are provided will be initialized while the rest receive default values.

[ ]:
```
response = proxy(arg1 = ...)
```

**Persistent connections**

- With a persistent connection, a client stays connected to a service. Otherwise, a client normally does a lookup and reconnects to a service each time.

- A persistent connection greatly improve performance for repeated requests while also makes the client more fragile to service failures.
- Clients using persistent connections should implement their own reconnection logic in the event that the persistent connection fails.

### 7.4.3 Service Node: Providing services

***rospy.Service* instance**

- In ROS, provide a Service by creating a ***rospy.Service*** instance with a callback to invoke when new requests are received.
- Each inbound request is handled in its own thread, so services must be thread-safe.
- Remember to return some data according to the definition of *package.srv.Service1Response*.

[ ]:
```
def handler(request_messsage_data):
    response to income requests

def Service1_server():
    rospy.init_node("Service1_server")
    server = rospy.Service("service_name", service_class, handler,␣
 ↪buff_size=65536)
    rospy.spin()
```