Robotic Operating System

Version 1.0

Ruotolo Vincenzo

January 16, 2022

Contents

List of Code Listings

Part I
Introduction

1 Definition

ROS is an open-source, meta-operating system for robotic systems, providing:

- hardware abstraction
- low-level device control
- implementation of commonly used functionality
- message-passing between processes
- package management
- tools and libraries for obtaining, building, writing, and running code across multiple computers

The purpose of ROS is:

- Support code reusability in robotics research and development
- Thin: code written for ROS can be used with other robot software frameworks
- ROS-agnostic libraries: the preferred development model is to write ROS-agnostic libraries with clean functional interfaces
- Language independence: implementation in different programming language (most used C++ and Phyton)
- Easy testing: built-in unit/integration test framework called rostest
- Scaling: ROS is appropriate for large runtime systems and for large development processes.

2 Description

ROS is a distributed framework of processes called nodes which enable executables to be individually designed and loosely coupled at runtime. Communication between nodes is implemented by means of topics, services, actions and parameters, for each of which is defined a data structure. Messages can be recorded through bag functionality which allows to reproduce offline them. The launch functionality allows to run multiple nodes simultaneously and settings their parameters. This represents the ***computational graph level***.

Nodes and in general anything that is useful to organize together, are organized into packages, which are the smallest unit that is possible to build in ROS, fulfilling a defined function and defined by their own configurational files. This is the base of the ***filesystem level***. Packages can be grouped into stacks, which can be built as packages in a nested framework. Furthermore, within packages can be defined message structures.

ROS features are accessible via code through ***client libraries***, allowing to implement nodes in different coding languages. Actually, the most supported are C++ (roscpp) and Python (rospy), alongside LISP (roslisp) and Matlab/Simulink.

The ***community level*** is the essence of ROS. Packages and stacks can be shared across the ROS community through a federated system of online repositories.

Furthermore, a ***higher-level architecture*** is necessary to build larger systems on top of ROS.

Part II
Computational Graph Level

3 ROS Master

The ROS Master provides name registration and lookup to the rest of the Computation Graph. It acts as a nameservice, storing topics and services registration information for nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. Nodes connect to other nodes directly; the Master only provides lookup information. Nodes communication is established over the TCPROS protocol, which uses standard TCP/IP sockets. There is always only one master in the network, nodes must know network address of which.

4 Nodes

Nodes are basic processes that perform computation, written through ROS client library (roscpp, rospy). Each node has its own name (graph resource name) that identify it and a node type (package resource name). They are combined into a graph and communicate with each other using:

- topics
- services
- actions

Furthermore, they have variables which can be managed using:

- parameter server
- dynamic reconfigures

4.1 Python node

1. Package structure

```
/pkg_name
    /nodes
        node.py
    CMakeLists.txt
    package.xml
```

2. Make the node executable

```
chmod +x node.py
```

4.2 C++ node

1. Package structure

```
/pkg_name
    /src
        node.cpp
    /include
        node.h
    CMakeLists.txt
    package.xml
```

2. Make the node executable

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
add_excutable(node src/node.cpp)
```

5 Communication

Messages	Structure	Application	Example	Tools
Topics	.msg	one-way continuous data stream	sensors data	rostopic
Services	.srv	blocking task for processing a request	trigger, request state, computations	rosservice
Actions	.action	non-blocking preemptable goal-oriented tasks	navigation, grasping, motion	actionlib
Parameter Server	.param	global constant parameters	names, settings, calibration data, robot setup	rosparam
Dynamic reconfigure	.cfg	local, tunable parameters	controller parameters	??

5.1 Topics

Topics are named buses for unidirectional, streaming communication between nodes, in which there are:

- Publisher – nodes that send data
- Subscriber – nodes interested in data

The master initializes the topic communication and tracks all the information. Topics' default protocol is TCPROS (TCP/IP-based). Another protocol is UDPROS (UDP-based) only for roscpp, that is a low-latency, lossy transport, so is best suited for tasks like teleoperation.
Rules:

- Any node can publish a message to any topic
- Any node can subscribe to any topic
- Any node can both publish and subscribe to any topics at the same time
- Multiple nodes can publish and subscribe to the same topic

5.2 Services

Services are a pair of messages exchanged between nodes:

- Client (Response) – node providing a service
- Server (Request) – node requesting a service which gives back the response

Rules:

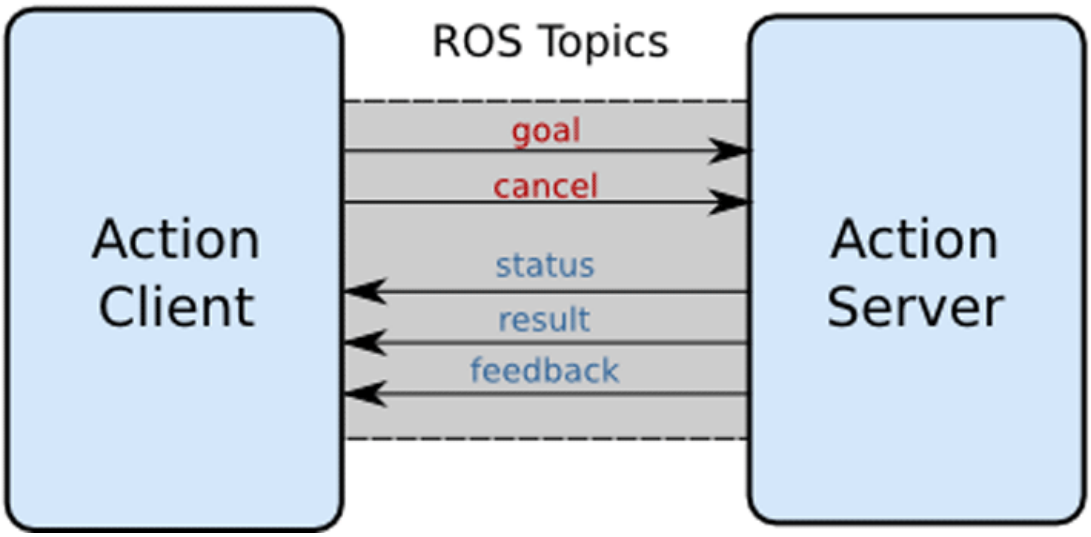
- Any node can implement services (server)
- Any node can call services (client)
- Servers call are synchronous/blocking

5.3 Actions

Actions, like services, are messages exchanged between two nodes, providing additional features:

- ActionClient – node providing a task and the possibility to cancel that task
- ActionServer – node receiving a task which give back the status, the result and the feedback

The used protocol is the “ROS Action Protocol” which is built on top of ROS messages (High-level).



Rules:

- Goal - can be messages, services or parameters data structures used to send goals
- Cancel – used to send cancel requests
- Status – used to notify clients on the current state of every goal in the system
- Feedback - provides information about the incremental progress of a goal
- Result - send upon completion of the goal

5.4 Parameter Server

A parameter server is a shared, multi-variate dictionary implemented using XMLRPC. Nodes can use this server to store and retrieve parameters runtime, for static, non-binary data. Supported types:

- strings
- 32-bit integers
- doubles
- float
- booleans
- lists of elements
- dictionary
- base64-encoded binary data
- iso8601 dates

5.5 Dynamic reconfigure

The dynamic reconfigure is a package providing a means to update parameters at runtime without having to restart the node.

CMakeLists.txt

```
generate_dynamic_reconfigure_options(
    cfg/dynamic.cfg
)
```

6 Bags

It's a format for saving, storing and playback ROS message data in .bag files. Bag files can be played back in ROS to the same topic they were recorded from, or even remapped to new topics.

7 Launch files

Part III
Filesystem

8 Package

The package is the smallest unit that is possible to build in ROS and the main unit for organizing software, which can contain:

- ROS runtime processes (nodes)
  - scripts: contains executables python scripts
  - src: contains C++ cource code
  - include: contains headers and libraries needed
  - launch: contains launch files
- ROS-dependent libraries (dependencies)
  - msg: custom messages structure definition
  - src: custom services structure definition
  - action: custom actions structure definition
  - config: configuration files
- Configuration files
  - package.xml: manifest file
  - CMakeLists.txt: CMake build file
- Anything else that is usefully organized together

```
catkin_ws
  /src
    /pkg_name
      /config
        file.config
      /include
        /pkg_name
          header.h
      /scripts
        node.py
```

```
      /src
        node.cpp
      /launch
        node.launch
      /msg
        message.msg
      /srv
        message.srv
      /param
        message.param
      /action
        message.action
      package.xml
      CMakeLists.txt
```

8.1 package.xml

The package manifest provides metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.

```
<package format="2">

  <name>pkg_name</name>
  <version>1.2.4</version>
  <description>pkg_description</description>
  <maintainer email="maintainer_email">maintainer_name</maintainer>
  <license>[sw license]</license>
  <url>http://ros.org/wiki/pkg_name</url>
  <author>author_name</author>

  <buildtool_depend>catkin</buildtool_depend>

  <! dependency only for building >
  <build_depend>...</build_depend>

  <! dependency only for building which exports a header >
  <build_export_depend>...</build_export_depend>

  <! dependency required only when running the package >
  <exec_depend>...</exec_depend>

  <! dependency including build_depend, build_export_depend, exec_depend >
  <depend>...</depend>
```

```
</package>
```

8.2 CMakeLists.txt

The CmakeLists file describes the build instructions for the Cmake.

```
########## Informations ##########
cmake_minimum_required()
project(pkg_name)

# Compiler
set(CMAKE_BUILD_TYPE "Release")
include(CheckCXXCompilerFlag)
CHECK_CXX_COMPILER_FLAG("-std=c++11" COMPILER_SUPPORTS_CXX11)
CHECK_CXX_COMPILER_FLAG("-std=c++14" COMPILER_SUPPORTS_CXX14)

if(COMPILER_SUPPORTS_CXX14)
    set(CMAKE_CXX_FLAGS "-std=c++14")
elseif(COMPILER_SUPPORTS_CXX11)
    set(CMAKE_CXX_FLAGS "-std=c++11")
else()
    message(FATAL_ERROR "No C++11 support")
endif()

set(CMAKE_CXX_FLAGS_RELEASE "-O3 -Wall -g")


########## Dependencies ##########
# Find ROS package dependency (<depend>/<build_depend> in package.xml)
find_package(catkin REQUIRED COMPONENTS
  rospy
  roscpp
  ...
)


########## Catkin specific configuration ##########
# Libraries and headers needed by all the interfaces you export to other
# ROS packages (<depend>/<build_export_depend> in package.xml)
catkin_package(
  DEPENDS rospy roscpp ...
  CATKIN_DEPENDS rospy roscpp ...
)
```

```
# Header directories
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```

9 Data Structure

The message data structure is the specific format used to compose messages:

- .msg – topic data structure
- .srv – service data structure
- .action – action data structure

9.1 Creating message package

It is always suggested to create a separate package containing all the custom messages created:

```
catkin create pkg msg_dep -c rospy roscpp message_generation actionlib actionlib_msgs ...
```

with the following architecture:

```
catkin_ws
  /src
    /pkg_name
      /msg
        message.msg
      /srv
        message.srv
      /param
        message.param
      /action
        message.action
      package.xml
      CMakeLists.txt
```

9.2 msgs data structure

```
# defining topic message
message_type message_name
```

9.3 srv data structure

```
# defining request message
request_type request_name
---
# defining response message
response_type response_name
```

9.4 action data structure

```
# defining topic message
# goal definition
goal_type goal_name
---
# result definition
result_type result_name
---
# feedback
feedback_type feedback_name
```

9.5 CMakeLists.txt

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

# Messages data structure
add_message_files(
  DIRECTORY msg
  FILES message.msg
)

# Services data structure
```

```
add_service_files(
  DIRECTORY srv
  FILES message.srv
)

# Actions data structure
add_action_files(
  DIRECTORY action
  FILES massage.action
)

generate_messages(
  DEPENDENCIES ...
  actionlib_msgs
)

catkin_package(
  CATKIN_DEPENDS rospy roscpp message_runtime actionlib_msgs
)
```

9.6 package.xml

```
<build_depend>message_generation</build_depend>
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
<depend>actionlib</depend>
<depend>actionlib_msgs</depend>
```

10 Dependencies

10.1 ROS packages

10.2 System dependencies

Python modules and C++ libraries are dependencies that can be imported in any other package in the workspace.

For python modules:

- package.xml
  - <exec_dep>: system dependencies (rosdep, rospkg, ecc...)

For C++ libraries:

- package.xml
  - <build_dep>: packages needed for building your programs, including development files like headers, libraries and configuration files. For each build dependency, specify the corresponding rosdep key
  - <build_export_depend>: packages needed for building your programs and furthermore that exports a header including an imported dependency
  - <exec_dep>: shared (dynamic) libraries, executables, Python modules, launch scripts and other files required for running your package
- CMakeLists.txt
  - find_package: find imported libraries
  - include_directories: include libraries directories