ROS2

Robotic Operating System

Version 1.0

Ruotolo Vincenzo

February 16, 2022

Contents

List of Code Listings

Part I
Introduction
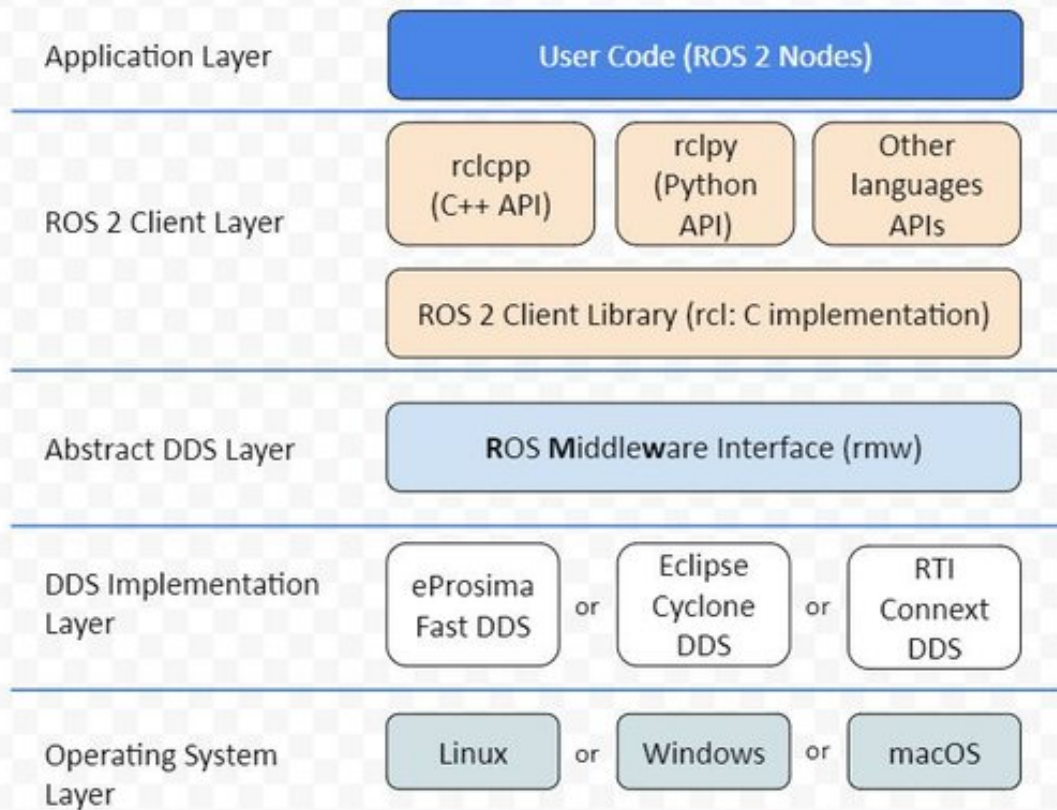
Motivation

Improvement with respect to ROS

1. Quality of Design and Implementation

2. System reliability

3. Real-time control

4. Validation, verification and certification

5. Flexibility in communication

6. Support for small embedded systems

Architecture

ROS 2 Architecture Overview

Application Layer	User Code (ROS 2 Nodes)	
ROS 2 Client Layer	rclcpp (C++ API) / rclpy (Python API) / Other languages APIs	
	ROS 2 Client Library (rcl: C implementation)	
Abstract DDS Layer	ROS Middleware Interface (rmw)	
DDS Implementation Layer	eProsima Fast DDS or Eclipse Cyclone DDS or RTI Connext DDS	
Operating System Layer	Linux or Windows or macOS	

DDS = Data Distribution Service is a decentralized, publish-subscribe communication protocol.
rmw = ROS Middleware Interface hides the details of the DDS implementations.
Use rclcpp for efficiency and fast response times, use rclpy for prototyping and shorter development time.

Part II
ROS Levels

ROS2 Graph

The ROS graph is a network of ROS2 elements processing data together at one time.

Nodes

Each node in ROS should be responsible for a single task (single, modular purpose). There are many ways by which nodes can exchange data each others: topics, services, actions, or parameters.

Topics

Topics allows to send continuous streams of data between two or more topics:

- publisher: if data is sent (publish to a topic)
- subscriber: if data is received (subscribe to a topic)

Services

Services are based on call-and-response model, providing data only under request by a client:

- service server: who provides the service (response)
- service client: who call the service (request)

Parameters

Parameters are configuration values of a node, corresponding to passing arguments to an executable.

Actions

Actions are an advance way to exchange data for long running tasks, using the client-server model of the topics and the call-and-response model of services, with the difference that can be deleted.

- goal service: service request
- feedback topic: data about the state of the task
- result service: service result

Part III
Concepts

Execution strategies

Since ROS2 is a system that deals with asynchronous events (such as timers, subscribers, service servers, service cleints, QoS events), it is crucial understanding how to wait for these events, when deciding to act, in what the order of execution.

There are two approaches that are used: Proactor and Reactor.

	Proactor	Reactor
Implementation	ROS2 Executor	ROS2 WaitSet
Description	wait for the read/wait to be completed	wait for the read/wait to be ready
Scheduling	built-in	handled by user

This can be translated into using Executor (Proactor) or WaitSet (Reactor):

	Executor	WaitSet
Use case	Extreme latency performance	Safety and mission critical system
Throughput	Low	High
Latency	Low	High
CPU utilization	High	Low

Executors

An executor uses threads to invoke callbacks (subscribers, timers, services, actions, ...).

- **Single-threaded executor**: the main thread is used for processing incoming messages and events of a node, through "spin(node)" method. The scheduling is barely the round-robin in the blocking spin call (in the thread in which you call spin).

```
rclcpp::Node::SharedPtr node = ...
rclcpp::spin(node);
```

which is the same as:

```
rclcpp::Node::SharedPtr node = ...

rclcpp::executors::SingleThreadedExecutor executor;
executor.add_node(node);
executor.spin();
```

- **Multi-threaded executor**: creates a configurable number of threads for parallelism. The actual parallelism depends on the callback groups.

```
rclcpp::Node::SharedPtr node = ...

rclcpp::executors::MultiThreadedExecutor executor;
executor.add_node(node);
executor.spin();
```

- **Static-single-threaded executor**: enhanced version of the single-threaded executor which optimizes the runtime cost.

```
rclcpp::Node::SharedPtr node1 = ...
rclcpp::Node::SharedPtr node2 = ...
rclcpp::Node::SharedPtr node3 = ...

rclcpp::executors::StaticSingleThreadedExecutor executor;
executor.add_node(node1);
executor.add_node(node2);
executor.add_node(node2);
executor.spin();
```

Issues   The three executors suffer from different issues, making them not suitable for real-time applications:

- mixing different scheduler

- priority inversion may happen in callbacks

- no explicit control over callback execution order

Components   The Single-threaded executor is used by components in order to create multiple single-threaded executors, which is a single node that has a single thread for each process.

WaitSet

WaitSet allows to implement the "reactor" behaviour, which allows waiting directly o callbacks, instead of using Executor. It makes possible to implement deterministic, user-defined processing sequences, possibly processing multiple messages from different sources together.

Scheduling   Scheduling strategy in this pattern is all up to the user.

Issues   At the moment, the interface to implement the WaitSet makes use of the callback, even though there is no call.

Callback groups

Callbacks node can be organized in a **Callback Group**, which determines the behaviour of the callbacks inside of that group, without dealing with the threading models.
The callback group types are:

- *mutually exclusive*: ensure that the callbacks in it, will never be executed at the same time by the executor, preventing to use the same shared resources (race condition)

- *reentrant*: makes possible that a callback be called at the same time as other callbacks, and furthermore it can be called multiple times concurrently

Callbacks of different callback group can be always executed in parallel.

Components

Quality of Service

Lifecycle

Part IV
Client Libraries

/	rclcpp	rclpy
/	rclcpp::ok()	/
/	rclcpp::WallRate	/